UNIVERSITÉ DE GENÈVE

FACULTY OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

Master's Thesis

Tesseract : a proposal for a container breakout tool

Submitted by : Théo Pirkl

Supervisor : Dr. Eduardo Solana

January 2024

Abstract

Containers have tremendously simplified the job of Information Technology (IT) specialists. They offer a complete system abstraction, from hardware to networking. While containers offer powerful features, their thin isolation layer with the host renders misconfigurations dangerous, as they may leave the system or other containers vulnerable. As container use is on the up, so are attacks to break out of them. A category that is on the rise is automated attacks, which may allow an attacker to take over a system in a matter of seconds. We present a tool written in Rust to automate the exploration of containers from its environment to breakout attacks, showing what an attacker could automatically achieve, as well as mitigation advices to reduce the attack surface.
Our tool is capable of compromising very popular containers, especially with third-party provided default configurations, and shows that breakouts could be achieved in seconds. We acknowledge that big organizations have assessed the risk of container breakouts and acted upon it, but notice no signs suggesting smaller organizations have followed through. We urge organizations to implement and enforce processes throughout the lifecycle of containers to reduce breakout risks, by implementing security measures at the very start of the container design.

Table of contents

Acknowledgments

I would like to personally thank the following people:

- Professor Eduardo Solana, for his supervision and precious counseling;
- Professor Noria Foukia, for her insights;
- Professor Florent Glück, for his contributions and enhancement suggestions;
- Nicolas Paschoud, Michaël El Kharroubi for their critical analysis and their help since the beginning;
- Rémy Cassini, Adrien Neuenschwander for their support throughout the years.

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface.

ARP Address Resolution Protocol.

AWS Amazon Web Services.

BIOS Basic Input/Output System.

CP Control Program.

CPU Central Processing Unit.

CRI Container Runtime Interface.

CSP Cloud Service Provider.

CVE Common Vulnerabilities and Exposures.

CVSS Common Vulnerability Scoring System.

DNS Domain Name System.

GENEVE GEneric NEtwork Virtualization Encapsulation.

GID Group IDentifier.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HA High Availability.

IdP Identity Provider.

IoT Internet of Things.

IT Information Technology.

JSON JavaScript Object Notation.

KVM Keyboard, Video, Mouse.

LSM Linux Security Module.

MAC Mandatory Access Control.

MAC Media Access Control.

MitM Man In The Middle.

MTU Maximum Transmission Unit.

MVP Minimal Viable Product.

NIC Network Interface Card.

NIST National Institute of Standards and Technology.

NSA National Security Agency.

NVME Non Volatile Memory Express.

OCI Open Container Initiative.

OS Operating System.

OUI Organizationally Unique Identifier.

PCI Peripheral Component Interconnect.

POST Power-On Self-Test.

RAM Random Access Memory.

SDN Software Defined Network.

SGX Software Guarded eXtensions.

SMART Self-Monitoring, Analysis, and Reporting Technology.

SPOF Single Point Of Failure.

SSD Solid State Drive.

SVGA Super Video Graphics Array.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

UID User IDentifier.

USB Universal Serial Bus.

VLAN Virtual LAN.

VM Virtual Machine.

VMM Virtual Machine Manager.

VMX Virtualization Machine eXtensions.

VPN Virtual Private Network.

VXLAN Virtual eXtensible Local-Area Network.

WWW World Wide Web.

1 Introduction

Computing is nowadays used everywhere, from freighting platforms to banking, from Cloud Service Providers (CSPs) to schools, and from prisons to industrial processes. Each of those entities has complex connections between the many departments they have, either in real life or in their computer infrastructure. Monolithic structures, such as a single centralized computing entity running the business logic as a whole, are no longer considered efficient because of their underlying complexity and limits [1], and it is well-known that separating services in separate "bricks" limits complexity and that it may contribute to eliminating any Single Point Of Failure (SPOF)[2].

A brick can be defined as a single business logic unit. For instance, a web shop could have a brick that consists of running a single website, another responsible for handling outgoing mail, and another for running the database. Separating and isolating bricks from each other allows systems not only to be more scalable, but also to be maintained more easily, as each one is isolated from the other. Furthermore, this also reduces the risk of side effects as each brick has its own environment, e.g. a brick responsible of sending emails that has suffered a power loss will not necessarily stop the brick responsible for handling web traffic from working.
Moreover, bricks can be dispatched on more than one computing unit for redundancy or load-balancing purposes. In contrast, systems described as *monolithic* are usually considered inefficient and unable to meet the needs in scalability [1]. For example, systems designed as bricks could be run twice or more, allowing the workload to either be distributed or to allow the workload to be highly available. This is more difficult with monolithic systems as services were not initially meant to be separated, and this may introduce unexpected side-effects in the business logic.

This isolation process is considered by security experts as good practice, because it ensures attacks do not compromise the whole infrastructure at once as each brick is "sandboxed", and therefore is fully isolated from the rest of the resources [3]. Each brick may have its own attack surface [4], but the process of separating the business logic into bricks also allows the introduction of security mechanisms, which can ultimately reduce the overall attack surface. A previous example suggested having a brick responsible for incoming web traffic. This brick could be the only one exposed to outside traffic, and all other bricks could be only connected to the web server service. By doing so, other sensitive bricks would be protected from illegitimate traffic as long as the web server brick is not compromised.

Two main technologies are capable of creating such bricks: Virtual Machines and containers. With

the rise of CSPs, containers have become a technology that is widely used. While this has created jobs and allowed many companies to get to market quicker, it also introduced yet another layer of abstraction between the system and the administrator. Several publications show that there are new risks with containers because of this abstraction [5] [6] [7]. Furthermore, industry publications suggest that security issues within container technologies are amongst the ten key issues that IT need to solve [8].

A popular attack with containers is breakout attacks, allowing a successful attacker to access the container host. This problem gets even worse in multitenant applications such as CSPs. As containers are light, so is their isolation from the host, as they are in most cases using the host Kernel. In some cases, a breakout can extend to more than one company, particularly in the case of CSPs [6]. In such cases, it's difficult to ensure optimum security, since it's possible that a company on the same server could also be compromised, enabling an attacker to laterally attack the others.
Furthermore, organizations often run hundreds of containers at the same time [9], with each its own attack surface. As the attack surface is scattered with each container, the difficulty of managing those containers and their security is augmented. Moreover, tools to ensure proper security terms are not included directly within the tools running the containers, which then requires additional work to implement security checks.
Moreover, attacks are now mostly done automatically, with scanners finding vulnerable targets and specialized programs attacking them. Attacks that used to take hours can now be summed up in a matter of seconds.

The most optimal way to avoid breakouts is to follow existing guidelines, especially from known and recognized organizations. Large organizations have been known to deploy significant blockades to avoid breakouts, but there is little evidence this practice was passed down to smaller companies.

This work analyzes container security, focusing on automation to find vulnerabilities in containers and trying to exploit them. We intend to explore the following questions :

- Can containers be scanned to determine which attack is feasible?
- What are the risks with the current use of containers?
- What are the challenges in mitigating attacks with containers?
- What can be done to avoid breakouts on containers?

1.1 Structure of the thesis

This report consists of seven chapters. The first chapter is this introduction to our work, which outlines the basics of containers and their safety.

The second chapter is the state of the art, which studies recent publications and existing technologies,

as well as their use-case. We will first focus on VMs in subchapter 2.2. We then show possible attacks on VMs in subchapter 2.3. The same is then done for containers in subchapter 2.4, and possible attacks on subchapter 2.5. We conclude this chapter with subchapter 2.6, where existing tools to either harden or break containers are compared.

As this work focuses on container security, VMs will not be the main focus but will rather be mentioned throughout this work. Before going any further, a deep understanding of how container engines work is required, which will be the focus of chapter 3. We will quickly review Linux processes in subchapter 3.1. Then, we will shift our attention to the inner workings of containers, more specifically their kernel modules, in subchapter 3.2. We will then analyze the industry use of containers in subchapter 3.3. We will conclude chapter 3 with subchapter 3.4, where we will compare existing orchestration container software.

We will draw a proposal for a container breakout tool in chapter 4. We will first establish a roadmap for building such a tool in subchapter 4.1. Then, prerequisites will be discussed in subchapter 4.2. We will focus on information gathering in subchapter 4.3, that is what information our tool can gather, from CPU information to potential weaknesses in the container. We will then analyze in subchapter 4.4 the pros and cons of such a tool from an applied perspective: what our tool can do and what it cannot. Only then can we discuss the attack process in subchapter 4.5, the attack scoring system, as well as attacks that can be implemented in our tool. Finally, we will conclude chapter 4 with a critical analysis of what was presented.

Then, we will discuss on chapter 5 audits of our tool on sample infrastructures. We will first present different infrastructures that can be described as “typical” in container infrastructure in subchapter 5.1, then we will describe the testing protocol that was used for our measurements in subchapter 5.2. We will present the results in subchapter 5.3 along the security score that was presented in subchapter 4.5. Subchapter 5.4 will analyze the obtained results. We will generalize obtained results on infrastructures seen in the wild and analyze the potential consequences linked to the results of subchapter 5.4 in subchapter 5.5. We will conclude chapter 5 with a critical analysis of what was presented.

As our tool is dedicated for blue teams in order to fix potential weaknesses and that it is our strong belief that no tool should be dedicated for the sole purposes of breaking infrastructures, we will focus on mitigation techniques to lessen the previously found vulnerabilities in chapter 6. We will go over the challenges and goals of mitigations in subchapter 6.1. Then, we will go in subchapter 6.2 over the existing mitigation techniques or standard published by various organizations. We will propose automatic mitigations in subchapter 6.3, which will take the form of suggestions for the administrator to apply for problems that can easily be mitigated. We will then shift our attention in subchapter 6.4 on *common mitigations techniques*, potential solutions to apply in order to fix problems not yet fixed by subchapter 6.3. Subchapter 6.5 will address problems that do not fit into subchapters 6.3 and 6.4, by proposing a list of early mitigation techniques. We will conclude chapter 6 with a reflection on

containers.

We offer in our last chapter, chapter 7, a conclusion and we recommend possible future works that can be done to enhance this work.

2 State of the art

2.1 Introduction

Most actors dependent on servers would benefit from splitting their business logic into smaller bricks. This allows for the separation of each service dependency, but it also allows for proper resource scheduling, as well as allowing the reducing of latency in cases of applications used worldwide. Splitting the business into bricks requires either one of two technologies.

The first technology is called *virtualization*. It is a technology that allows to virtualize all resources linked to a machine such as its network, its hardware, or its Operating System (OS) into a Virtual Machine (VM). This technology allows for the proper separation of each resource into its own well-defined virtual space, but it also introduces complexity, such as the maintenance of the newly created VM. The first subchapter 2.1 will introduce virtualization as well as virtualization techniques, then subchapter 2.2 will focus on potential attack techniques. Considerations when creating a VM will then be introduced in subchapter 2.3.

The second technology is called *containerization*, which allows the isolation of a group of processes on the host. From the inside, this group looks exactly like a VM, but with key differences that will be presented in the subchapter 2.4 on *containers*, with a focus on differentiating VMs from containers and the components of a container engine. Attacks on containers will then be presented in subchapter 2.5. Finally, subchapter 2.6 will assess the different technologies and attacks, as well as existing security tools to audit such technologies, either to breakout from the isolation layer or to mitigate potential weaknesses.

2.2 Virtualization

In Computer Science, virtualization is a technique that allows to run a resource on a computer, as if it existed physically. Entire networks, Central Processing Units (CPUs), hard drives and many more can be virtualized.

Our work focuses on platform virtualization, which is the concurrent execution of multiple OSs on the same host [10]. The VM, or *guest*, is isolated from the rest of the resources and other VMs. This

means that a single machine could run many VMs as long as the underlying hardware has the adequate, available hardware to support it.

Virtualization may seem new, but it is in fact far from the truth. This technology, although different at first, was first seen and implemented in 1960 [11]. As personal computers were nonexistent, the notion of user and allocated resources were done on a single computer (a mainframe), by using Control Program (CP), an OS that allowed running primitive VMs. This allowed multiple users to have their own allocated space on a single machine [12]. Nowadays, virtualization is quite powerful as it can virtualize entire CPUs, RAM, and even high-end gaming setups with gaming Graphics Processing Units (GPUs) [13]. It is considered that at least one form of virtualization exists on most devices, from mobile phones to gaming devices.

The different components of a virtualized infrastructure are shown on figure 2.1. The *hypervisor* (or Virtual Machine Manager (VMM)) controls the different VMs by controlling the amount of dedicated resources, their state, and so on. Each VM has its own dedicated (although virtual) hardware, its own OS, and finally installed apps. Note that the hypervisor may run directly on hardware, called type 1 virtualization, or on the host's OS, called type 2 virtualization. Type 1 virtualization usually refers to a minimalistic OS running directly on hardware. In both types, the host OS always exists as a host OS is always required. In type 2 VMMs, the host OS is a common one, such as Linux, Apple Mac OS X or Microsoft Windows, whereas type 1 usually refers to a minimalistic OS made especially with VMs hosting in mind.

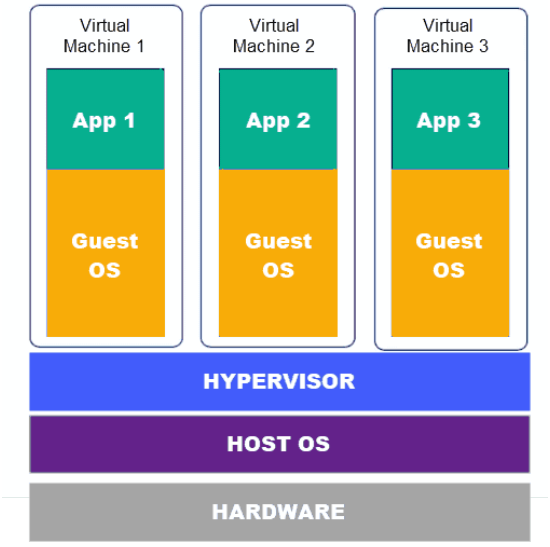

Figure 2.1: Architecture of a host running VMs on a Type 2 VMM [14]

There are three types of platform virtualization.
The first one is *full virtualization*, which will also virtualize the hardware of the VM. This type of virtual-

ization means that the underlying hardware is virtualized (including its Basic Input/Output System (BIOS)), which is in many cases slower because the VM also has to handle hardware initialization (from Power-On Self-Test (POST), to setting Random Access Memory (RAM), and so on). Full virtualization is considered to be much more portable, as everything is virtualized. Therefore, very few compatibility issues may occur. The guest OS is not aware that it is running on a VM, as nothing apart from reduced performance may point to this conclusion [15]. The most common type of full virtualization nowadays is done by the CPU, for reasons explained below.

The second one is called *paravirtualization*. It does not provide full isolation. The guest OS is slightly modified and knows it is running as a VM. Virtual hardware is "shown" to the VM, allowing all calls to the CPU to be handled correctly by the VMM. Paravirtualization is considered to be much faster. For instance, apps compiled as *unikernels* provide an OS, libraries, binaries, and dependencies for a single app. This allows an app to run in a completely isolated environment, with the kernel knowing that it runs on a VMM (and optimized for it). Performances in a unikernel are considered to be unmatched, even with other technologies.

The third one is called *hardware-assisted virtualization*. Instead of using software to enable full virtualization, hardware is used as it allows for tremendous speedup. The need to modify the guest is, as with software *full virtualization* (seen before), not required. This is the most used virtualization technique as almost all CPUs have their instruction set on how to accelerate virtualization. [16].

As most CPU sold in recent years almost all have virtualization capabilities, this work will focus on the third category of virtualization category.

Most CPUs nowadays have instructions on how to accelerate the virtualization of entire OSes. Technologies, implemented at the CPU level, allow the CPU to handle some of the virtualization workload on its own. Those technologies are known as *VT-X* with Intel, *AMD-V* with AMD, and *virtualization extensions* with ARM. This allows the CPU to be able to virtualize itself, that is, the CPU can handle being virtualized on its own. Special instructions are implemented into the CPU instruction set, allowing the notions of virtualization to be handled at great speeds. For example, Intel, in its Virtualization Machine eXtensions (VMX) implements `VMLAUNCH`, which allows the CPU to launch a VM managed by the VMM.
Hardware-assisted virtualization is not limited to processors : other technologies allow I/O (Input/Output) to be accelerated as well. For example, the VT-C technology allows for Network Interface Cards (NICs) to support VMs, allowing for high-speed communications [17].

Those technologies allow for high processing speeds, but do not implement any isolation layer between the host and the VMs whatsoever. Another technology, already used to specify the access level of processes, is used to ensure VMs cannot interfere with the host's kernel or other VMs. This technology is named *CPU rings*.

2.2.1 CPU Rings

Most CPUs run with layers (called *rings*), each giving access to specific permissions. Although the number of rings is debated [18] and the name of this technology may change depending on the CPU's vendor, Intel considers there are four rings on a CPU [19]. The outer ring, ring number three, is used for client applications, while the inner ring, ring number 0 is for the OS kernel. An example of CPU rings is shown on figure 2.2. Our work will focus on this model, for unicity and simplicity reasons.

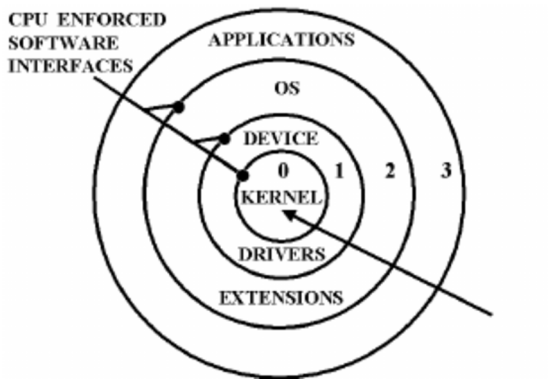

Figure 2.2: Rings on an x86 CPU[19]

This means that most applications must ask the kernel for specific rights, and generally do not have access to hardware or high-access privileges. Only the kernel decides who gets what. In fact, most applications must pass *call gates* in order to gain privileges such as writing a file, accessing a peripheral, and so on. Call gates will then decide whether or not the program can access the requested privilege.

VMs each run their kernel in what looks like bare metal (directly on hardware), but in fact, the host acts as a VMM and orchestrates calls for each VM so that each VM is fully isolated. As each VM must run its kernel and since every kernel expects to get ring 0 access, the VMM must run in ring -1[19]. This is a special mode that is above the underlying VM's ring 0, and reserved for special cases when more than one kernel is present, for example when there is a VMM. This allows the VMM to always have the upper hand and to avoid interferences from VMs [20]. By running in ring -1, the VMM can operate normally by, amongst many other tasks, redirect calls to memory, storage and peripheral requests to the specific allocated resources. Ring -1 is shown on figure 2.3.

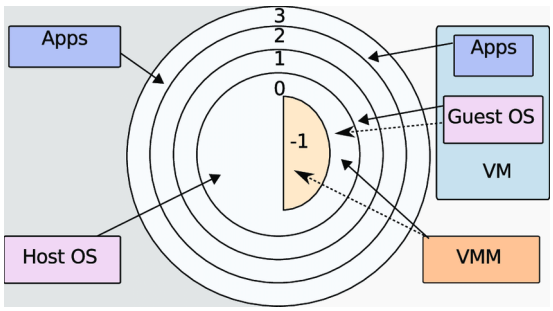

Figure 2.3: Rings on an x86 with VMs[19]

This complex ring system allows for VMs to run without any additional configuration: an OS running on bare metal will run similarly on a VM infrastructure. Two kernels (that is, the host's kernel and the guest's kernel) can coexist without any conflict on the same hardware.

2.2.2 Virtual machines considerations

VMs are considered, due to their fully virtualized nature, as "complete" machines. This implies that a certain number of measures must be taken to keep the VM and the VMM in good condition. While this work does not claim to offer a complete list, several considerations are worth mentioning for context purposes that will be used later on.

As VMs independently exist of each other, they do not depend on anything besides the VM host for power and resources. This makes each VM a complete machine. In terms of security, considering a VM as a machine is important : considering it as such means that each VM will most certainly require updates and maintenance. This is paramount to the safety of the VM.
This is one of the costs of virtualizations : while it allows for proper resource management as well as compartmentalization, it requires maintenance and monitoring, just like a physical machine would.

Considerations regarding the provisioning of resources are essential for long-term maintenance : while a VM can run with dozens if not hundreds of CPUs, it is generally a good practice to assign just enough resources required for the VM to run optimally, but not more. Doing so allows the dispatch of resources more easily, and it may also help detect bottlenecks due for example to high CPU usage.

Finally, using VMs may require to consider much more the underlying hardware. For example, whether there is enough computing power for all VMs is a question that needs to be answered. Prerequisites before implementing a network of VMs may be steeper than with other technologies, such as containers.

2.3 Possible attacks on virtual machines

VMs are complex machinery. As described before, VMs allow not only the virtualization of the whole OS, but the hardware as well. The guest OS has little to no evidence that there is another layer of management above. A few techniques have been discovered to detect potential VM execution [21] [15], which allows an attacker to detect whether the attacked machine is a VM or not.

Virtual machine breakout is the process of a program breaking out of a VM confines. Breaking out of a VM could allow the attacker not only to control the host OS, but also to potentially control other VMs on the same host.

Virtual machine breakouts are challenging as they present several steps, each hard to obtain. The first one is to be able to run as the guest operating system. As presented before, each user space application runs in the level 3 ring, while administrative actions run in the ring 0. This means that an attacker may need to first compromise the VM kernel. This would allow the attacker complete access to the VM resources. However, this administrative access would be limited to the VM and would not extend to other neighboring VMs. In order to continue the attack, a second step is required : escaping the VMM. This would allow the compromised VM to access resources outside of the allocated guest resources, potentially accessing the VM host system memory. Lastly, as soon as system resources are found, the attacker would have to edit system memory to gain `root` privileges. While attacks exploiting bad memory management are common and have been documented [22] [23], attacks from a VM may be significantly more difficult, due to the many indirections in between the guest and the host.

This makes virtual machine breakouts (that is, controlling the hypervisor) “not impossible, but it is very sophisticated” [24].

2.3.1 Documented cases of virtual machine escapes

As previously mentioned, VM escapes are not impossible but considered rare. In some conditions, such as peripherals (graphic cards or other Peripheral Component Interconnect (PCI) devices), calls may be directly sent to the device or require drivers to run. While drivers do not run in the user space, they still run in some cases as ring 1, which is not as high as the kernel but high enough to run some administrative tasks.

CVE-2020-3962 is a good example of a VM breakout. VMWare products distribute a Super Video Graphics Array (SVGA) virtual device, which allows to create a virtual graphics adapter. This may be useful for many applications such as Keyboard, Video, Mouse (KVM) needs, VM or software depending on Graphical User Interface (GUI), and so on.

CVE-2020-3962 exploited a bug in the virtual SVGA device (use-after-free, using a reference memory zone after it has been freed [25])[26]. An attacker with access to a VM with 3D-enabled graphics may be

able to run code on the hypervisor from the VM.

While documentation on this attack is sparse, information about what happened can be guessed. The virtual device was running on the host and was published as a generic device to all VMs having 3D-Acceleration enabled. Then, drivers were made available to the user-land by the kernel.

In this case, any application that has the right to run 3D primitives can run code to the drivers, which in turn send this code to the VMM, and finally to the virtual SVGA device running on the host. This bypasses several of the previously presented “checkpoints” : an attacker does not have to gain `root` to the VM, and does not have to escape the VMM as the attack uses a device to gain `root` on the host. Finally, the device runs as a privileged process on the host. The process was flawed (as shown by the CVE) and was allowing code that was previously freed to be rewritten (double-free attack). This typically allows for arbitrary code to be run, completely bypassing the many securities in place to ensure the compartmentalization of the VMs.

However, the amount of research required to find such exploits are nothing short of enormous. VMM source codes may not always be available, and in most cases VMM discuss with low-level components such as the CPU. Deep knowledge of the underlying processor architecture is therefore required, which may slow down the discovery of exploits. Furthermore, a single exploit may not be enough, as one may obtain `root` privileges but another one would be required to breach the VM confined.

VMs may be considered as “heavy” or “bulky” due to its many additional responsibilities, but they do generally provide a thick layer of isolation between the host and the guest. In our next subchapter, we will focus on the container technology, that provides isolation at a much higher layer, but in a lighter way.

2.4 Containerization

The first concepts of containers are more recent than the first iterations of virtualization. In fact, containers were known as *jails*, and have existed since 2000 in FreeBSD. Many projects have been published in between now and then, for example the VServer project in 2004. The most significant advancement was in 2007 [27], when the Linux Kernel included a module for containers in its system. This module, called *cgroups*, provided resource groups and allocation for processes running on the same host. This meant a process could have allocated resources for itself, but not exceed resources at its disposal.

In fact, this module led the way for technologies as we know today such as Docker, Podman and many others.

Containerization is a technology that also allows compartmentalization, and proper dispatch of resources. From a user perspective, it is the same as a VM : the user can connect to the container just the

same, packages can be installed, filesystems can be mounted and so on. However, it could not be more different from VMs. While the concept is the same, containers (what was referred as *guests* before) do not provide the same level of isolation. For instance, containers do not allow for the virtualization of hardware.

In fact, a container is a single process hosted on the host and barely more. The OS uses specific processes to ensure the "inside" of the process is isolated from the rest of the system. However, all containers run on the Host's OS, and use the same resources. Because the guest and the host share the same kernel, there is no way of tricking the guest into thinking it is running as an OS, first because the guest is a process, a simple unit inside an OS instance, and second because the isolation is too thin to allow the guest into thinking this.

The key differences between containers and VM are shown on figure 2.4.

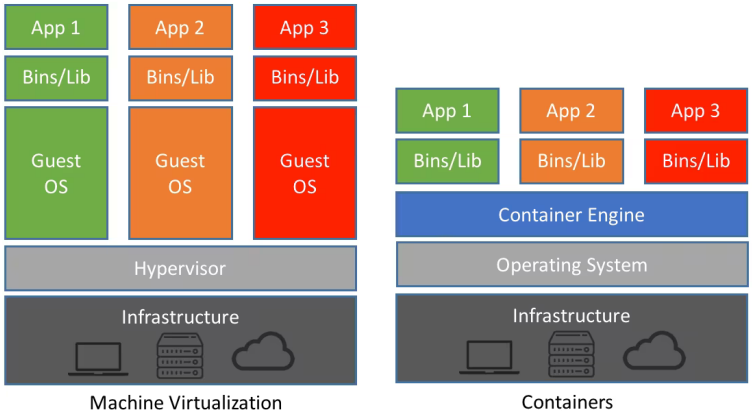

Figure 2.4: A comparison chart between VMs and containers [28]

There are two main advantages to use containers instead of VMs :

- The first one is that containers generally are extremely light : a container can start in a matter of seconds, due to the fact that a container does not have to load any driver, initialize the hardware or handle lengthy booting procedures. This also removes any requirements of virtualization extensions presented before, such as VT-X or AMD-V, as in most cases [1] there is no virtualization at hand.
- The second one is derived from the first : given a container is not an actual VM, it is not considered as a machine as such. This also allows maintenance to be lessened, but not removed.

[1] Linux operating systems do not need virtualization as they have all the required modules to run containers directly. Proprietary OS such as Microsoft Windows and Apple Mac OS X generally run a light Linux VM that bridges the host OS (Windows or Mac) and the containers.

From an executive and financial perspective, as containers are lighter than VMs, more can be hosted on the same host, and most containers will start much faster than its VM counterpart. Those two advantages attract much of the industry as it helps reducing costs. Nowadays, most OSs, if not all, are able to run containers, from Microsoft Windows to Solaris, regardless of the underlying hardware.

The main disadvantage of using containers is the downside of its greatest advantage : the lack of a complete isolation. While several mechanisms ensure an isolation between the container and the host, each of those mechanisms can either be disabled by the user or be compromised. In those situations, the host machine would be exposed.

Another disadvantage of using containers is that the OS is aware that it is not running on bare metal, as there is not enough isolation to “trick” it. Furthermore, most container engines leave traces suggesting that the current environment is, in fact, currently running in a container. This would let an attacker easily gather many information about the current environment, for example the model of CPU or to the container engine currently being used. Those informations are preferably kept confidential or away from prying eyes, as those could lead an attacker to find vulnerabilities or sensitive files or variables.

2.4.1 Use-case of containers

Containers have many uses, but one that is particularly applied is the use of the *microservice* approach. Rather than having one monolithic application, that is one that contains one database, one web server, and one system handling business logic, the microservice approach recommends creating one service for each module of the application : one service for the database, one for the web server, and so on.

This allows to untangle the many dependencies each system has, concentrating on creating a space just wide enough for each service to fit. Containers do provide the necessary tools to create environments containing little resources, networks to group services together along with systems to accommodate high availability and load balancing. The difference between monolithic and microservice-oriented architectures are shown on figure 2.5.

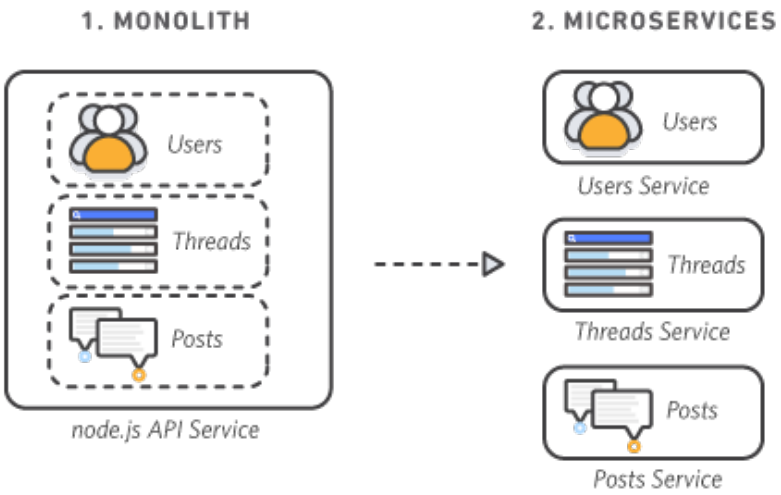

Figure 2.5: Example use of containers through microservices. [29]

As containers can be quickly created, configured and run, the microservice approach is usually preferred with containers rather than with VMs. Most Cloud Service Providers (CSPs) provide guidelines on how to implement this architecture on their platform. As this work will focus on security incursions, we will not use CSPs but rather host our own containers, isolated from any other infrastructure.

2.4.2 Security mechanisms of a container

The Linux Kernel provides a resource management system to limit what a container can actually access [10]. For instance, a container should not be able to impact the activities of other containers.

This resource management system revolves around four key technologies (also called *security primitives*), each responsible for one specific element of isolation. This work goes in-depth for each of those technologies in chapter 3, but it is worth mentioning that those *four security privimitives* work in concert to ensure a level of isolation that allows the container to run without being able to see the other processes on the host. For instance, running `ps` on a host will show the container and its children processes, but running the same command from the container will not return the process from the host. As processes are organized into a tree, containers can be considered as a leaf in the process tree of the host, with no access to its parents.

Additional security mechanisms already included with Linux and most container engines can also be configured to further restrict the container. By default, container engines provide a default set of settings, which generally allows a wide range of containers to run, while keeping an optimal level of security. While this balance is challenged by publications suggesting it is best to restrict the container

as much as possible on a use-case basis [30] [31], most attacks can be stopped by a container using the default configuration. One example of security mechanism that is included with Docker is AppArmor, which confines a program to specific resources. For instance, an AppArmor profile could force a container to be able to work only in `/data/app`, or to use specific ports.

A container running a configuration with minimal changes will most likely not be impacted by attacks, but containers running with a separated configuration set, regardless for a good reason or not, may be more impacted by attacks than one running with the standard configuration.

2.5 Possible attacks on containers

As previously mentioned, four security primitives work together to keep the container isolated. It is very important to denote that each technology has a specific area of expertise in mind, and that they barely overlap on another. While this avoids interference from one technology to another, it means there is no "*safety net*" : should one of those technologies fail for any reason, the thin isolation layer may "break", exposing the host, along with the other containers. In fact, those four security primitives are the only layer isolating the container, represented by a group of processes, from the host.

As described before, a VM has much more isolation than a container. Most attackers focus on *breaking out* of a container, as it is much easier than with a VM. A container *breakout* can occur in mainly two cases : either the container was misconfigured or the container engine had a flaw. As most security experts agree, misconfigurations are quite common, even in production environments. However, misconfigurations can endanger the safety of a container or the whole container host, as misconfigurations can remove isolation features between the host and the container. In such cases, an attacker gaining control of a container could get out of the container (*breaking out*) and gain access, albeit potentially limited, to the container host.

In cases where the container engine is flawed, the attacker does not exploit a weakness in the container configuration but exploits how the container is handled. For instance, CVE-2019-5736 [32] allowed attackers to rewrite the container runtime process (in this case, `runc`) on the host, therefore gaining `root` privileges. However, those attacks are more sophisticated as they require a specific container engine and version. Because of this, misconfigurations are usually considered the preferred way of breaking out of containers.

As with any software, there are many vulnerabilities, but the most exploited ones are the weaknesses left by misconfigurations, as they may allow trivial access outside the container. This allows an attacker to take over a system with very little time, as the container may also have access to other containers and hosts, allowing the attacker to move laterally to those other systems.

2.6 Container attacks nowadays

Container vulnerabilities require specific versions of the container engine and do not work in all cases. Misconfigurations, on the other hand, can leave vulnerabilities in the long term. They are therefore the preferred method of attack on containers [33]. This is also because testing container misconfigurations can be quickly done, should an attacker get access to a container. Automated scans and exploitations of weak, exposed machines is already a common problem, especially with the rise of Internet of Things (IoT) devices [34]. An attacker getting access to a container can, as presented before, automatically gather information about the host, but also see, if ports have been exposed, if volumes connected to the container are insecure and so on. This can be achieved in a matter of seconds, by systematically checking services, files, and networking information on the container.

Containers are popular amongst attackers because they are used in many different scenarios, especially often within development scenarios or environments with little to no security concerns in mind. For example, containers can accommodate dependencies and provide a quick and ready-to-use development environment, but with many opened ports, which could lead to security gaps.
Furthermore, as there are at this time of writing hundreds of thousands [35] of containers, most needs are already fulfilled by third parties. For instance, `nginx` provides a web server image ready to use. This allows many developers not to create their containers and rely on existing containers, but this also means that developers using those containers have little to no knowledge of the inner workings, dependencies, or potential weaknesses of the third-party container.
Moreover, many companies provide ready-to-use files or command lines to copy-paste, obfuscating the behaviour of the containers.

This ease of use is beneficial for many users, from developers to companies, but it also hides the complexity of container isolation mechanisms. This may induce users to cut corners to achieve their objectives faster, for example by removing isolations between the container and the host. For example, in order to create containers on the go, certain companies have exposed their Docker socket. While this may be very useful in many scenarios, exposing it without proper security measures gives direct access to attackers inside the company servers [36], leaving attackers with what is essentially an opened `root` shell.

Another popular attack is *typosquatting* : as many developers and users prefer existing, popular containers available on container hubs, they use those instead of having to build out their own. However, typing the name of the container wrong may be an actual security threat, as attackers have registered containers with names quite similar to the popular container names, for example registering a container named `ngixn` for a `nginx` container [7]. Downloading such a malicious container will not necessarily mean that the entire infrastructure will be compromised, but it does mean in most cases that an attacker had indirect access to all containers in the same network.

In fact, many other security risks related to containers do now exist [37], and are considered by some as an emergent field in security, whether it is in research or in its application.

2.6.1 Existing container security tools

In most security scenarios, one can categorize tools either for red teams (teams dedicated to breaching an infrastructure) and for blue teams (teams dedicated to defending the infrastructure against red teams). This is by no means different here, as both tools exist in this case : tools to break out of containers, and tools to protect them.

2.6.1.1 Red team tools

While there are existing tools that are able to scan containers for vulnerabilities or to gather information [38] [39] [40] [41], no results are delivered nor verified, and all seem to concentrate on Docker, as this is the most popular tool [42].

It is therefore impossible to evaluate the performance of either the tool or any security score of the container. Moreover, tools found on communal websites such as GitHub cannot be considered as trustworthy as they could pose a threat. However, it is interesting to denote that tools either designed to break out of containers or find vulnerabilities statically are numerous, showing a trend towards container security.

There are a number of publications on security assessments on containers made by researchers. Several publications show the use of containers in popular environments such as cloud, which allowed attackers to gather information about other tenants in specific cases, for example when the attacker and the other tenant were running on the same hardware [6] [43]. The results suggest that the isolation between containers, even when those containers do not belong to the same owner, cannot be considered as secure. In fact, many exploits have been found to be working in default container configurations, highlighting with how little security containers can run with [30].

2.6.1.2 Blue team tools

On the other side, security tools hardening container engines exist but are not widely adopted, as most container users use official releases, rarely containing additional, external security tools. Automated tools to harden seccomp profiles allow containers only to receive calls they do need to function [44], using static code analysis to generate a list of used (and therefore allowed) seccomp calls. [45]. The use of Seccomp and AppArmor technologies to enhance the overall container security is backed up by publications [46] [44] [47], proving this mean to be effective.

Big Tech industries such as Google have also rolled out sophisticated tools such as *distroless containers*, containers with just enough requirements to run their intended payload [48], but those solutions are described by third parties as [49] inadequate, mainly because they alledgedly solve problems that do not exist. For example, *Red Hat* mentions that while reducing the container image size is optimal practice, it does not necessarily reduce the attack surface. They also mention that the term *distroless* is misleading [49], as any app and container require an underlying operating system (or *distro*). It is interesting to note that security efforts, even coming from *Big Tech* are thwarted and almost considered as hostile by others.

Reports suggest that security efforts were made towards the statical analysis of containers [9], effectively reducing the attack surface of each container. Such tools will detect and potentially patch images with outdated dependencies [50] [51] [52].

While statical image analysis optimizes the coverage of optimal security practices, there are few tools made for dynamical container analysis, that is scanning the container once it is running, and not the container image in itself. While container engines published tools able to detect misconfigurations [41], there is little to no evidence suggesting this tool is popular or widely used in the container users community. Mature organizations, such as CSPs like AWS and Microsoft Azure, have started using dynamical container analysis such as Falco [9] [53], but there are no publications suggesting this is a widespread practice. Worse, the few tools able to detect anomalies in real time are policy-based, which may be circumvented by new attacks [54].

Many other security tools and practices exist : from containers running as unprivileged users to containers inside VMs, or containers running on trusted platforms such as Intel Software Guarded eXtensions (SGX) [55], many techniques allow containers to run in environments that can be described as more secure. However, a tool is only a part of the solution as a tool requires someone to use it, and tools may not be widely adopted, causing problems either with exotic configurations, or scalability when using orchestrators such as OpenShift or Kubernetes. For instance, there is no evidence Intel SGX containers are entirely compatible with orchestrators. Furthermore, SGX may require applications to be changed to fit SGX requirements, removing the technology agnosticism from the changed applications.

Tools are not the only measure taken to protect containers. A number of publications show an interest towards container security. Researchers are studying the possibility of frameworks able to thwart container attacks based on Bayesian Game theory [56] to propose a new security baseline for containers. Others have studied attack mechanisms through applied means with honeypots to build detection rules [57]. Security researchers also categorized the main attack vectors required to protect both containers and hosts from container-related attacks [37].

In contrast, many container authors usually provide ready-to-run commands or files to deploy their containers easily, with little to no security in mind. For example, containers may suggest exposing the host's network interface card instead of using a virtual one, thus exposing the host to the container,

where a port forwarding would have sufficed in most cases.

This generalized behaviour is getting more common by the minute, as most users are not interested in the installation or configuration process but usually wish for a turnkey system. This results in many software giving arbitrary commands to copy-paste or to pipe directly into the user's shell [58], which may cause catastrophic damages either by an attacker, bad network configuration, or just by bugs in the third-party script. For instance, Tailscale, a known Virtual Private Network (VPN) solution provider, suggested in early 2023 downloading a script and piping into a shell. This example is shown on figure 2.6.

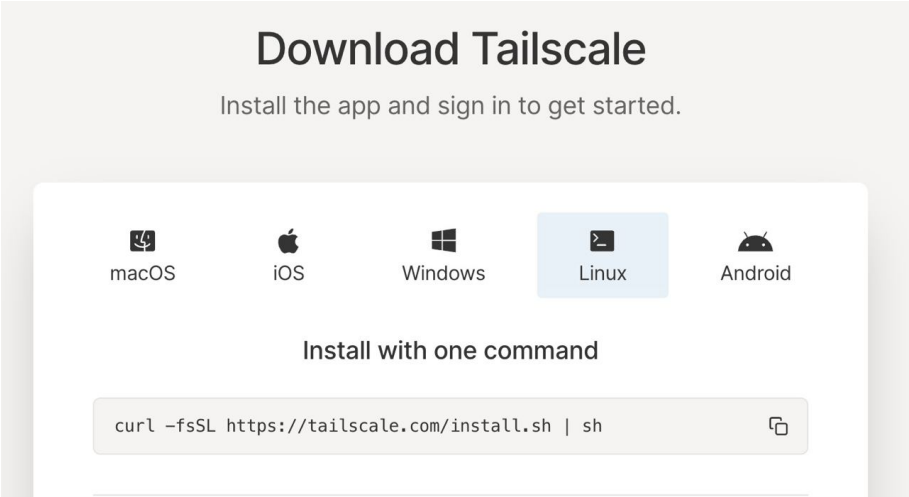

Figure 2.6: Example of a company suggesting piping an external script directly into the shell

While it is important to denote that Tailscale made efforts to allow users to see the contents of the script before running it, it is still a common practice to pipe third-party scripts directly into the terminal [59] [60] [61].

2.7 Conclusion

Monolithic infrastructures are slowly considered obsolete. Existing tools to separate concerns and services into bricks are now broadly used. Two techniques exist and have been presented in this chapter : VMs or containers. VMs are considered to be generally more secure, but they also have in most cases much more overhead than containers (as the VM generally do virtualize everything, from the BIOS to each component of the VM), which makes them slower. It also implies periodic maintenance on them as they should be considered as computers. VM that do not need to virtualize everything are considered much faster, such as unikernels or paravirtualization. Containers, on the other hand, are

considered in almost all cases light and fast, and not as a whole machine but rather like an isolated group of processes.

Both technologies are widely used and provide a way of splitting units of a service or a business into smaller bricks. However, each technology has its own weakness as well, leaving each of them with a specific set of attacks. VMs can be broken out of with sophisticated attacks, whereas containers can be broken out with less efforts due to bad configuration or poor security planning. Given that container attacks are more popular and less sophisticated, an attack on a container can rapidly lead to a breakout and potentially the host getting compromised. Many security actors are assessing the different attack vectors on containers, and many have found a significant number of exploits leading experts to declare containers to be not secure due to the kernel-sharing property [30]. To protect containers against such attacks, mature organizations and researchers have started the research of tools and technologies to protect the container against attacks, some of which will be described in the next chapter.

3 Container engines

Container engines are the core of the container technology. They orchestrate how containers run, what they can do (and most importantly what they cannot), what they can access and so on. As presented, container engines make use of existing embedded Linux technologies to allow containers to run. While each container engine has its own set of particularities, most of them work in a very similar way.

As our tool requires a deep understanding of containers, it is paramount to present and analyze how containers work in detail, as this will be relevant in the following chapters. Before presenting containers in-depth, our work focuses on the many tools and engines available on the web to run containers. Linux processes will be quickly introduced in subchapter 3.1, as containers depend on their structure. Kernel modules typically running containers will be presented in subchapter 3.2, along with their area of expertise. The use of containers in the industry is then presented in subchapter 3.3. Finally, the orchestration of containers, that is the management of containers without the worry of underlying hardware [62], will be presented in subchapter 3.4.

3.1 Linux processes

Linux processes are the core of the Linux Operating System (OS). They allow resources to be grouped together. There are many important properties in a process structure, however, our work focuses on a few characteristics.

- Links are used to refer a process to its parent [63]. The initial process, often called `init`, is the only one parentless. All other processes depend on another, from the moment they start to the moment they stop. This forms a tree, in which the root is the main process executed at startup. Containers are no exceptions : they are created as children of the program in charge of running containers (the *container runtime*). An example of process tree is shown on figure 3.1.
- Identifiers, which describe the process's permission. A process possesses four pairs of identifiers, only one of which is relevant to our work : User IDentifier (UID) and Group IDentifier (GID). The UID identifies the user that the process is running on behalf of [63]. The GID works on the same principle, only for a group instead of a user.

- The process identifier, which consists of a number starting from 1 (0, in some cases), representing the identifier of the process. Each process identifier is unique, and process identifiers are reused over time.

By the previous definitions, containers are then nothing more than a group of process. However, it is known that containers are more than that, as they are isolated from each other and the host. The mechanisms to ensure each container can live with the aforementioned characteristics are defined in the next subchapter : kernel container modules.

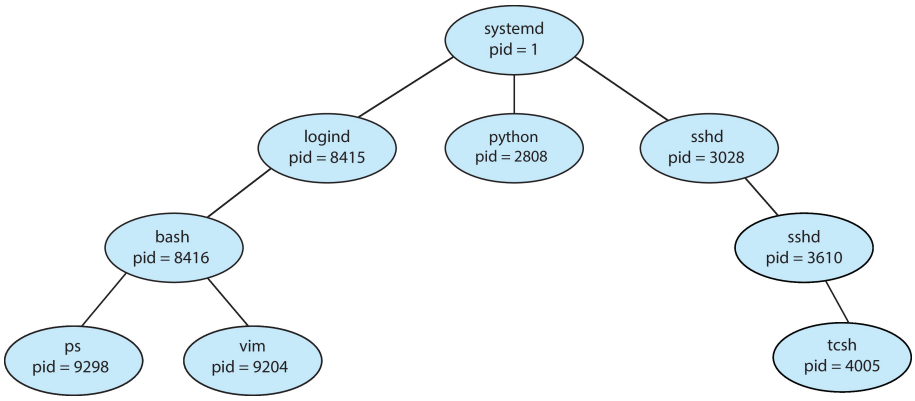

Figure 3.1: Example of a process tree [64]

Figure 3.1 shows the tree structure. This structure privileges isolation for each user, as a user will have a main process, then others stemming from it. In this case, one can quickly see that the `ps` process on the bottom left is linked to the `bash` interpreter, which is linked itself to `logind`, which is in charge of user logins.

3.2 Kernel container modules

As discussed in our state of the art, there are four modules allowing containers to run :

- SecComp, which define which kernel calls a process is allowed to do;
- cGroups, which define the limits a group of container is allowed to consume, whether it is the Central Processing Unit (CPU), Random Access Memory (RAM) or the hard drive;
- Namespaces, which create separate environments (called namespaces) for networking, user and group IDs, and many more;
- Capabilities, which allow to give specific permissions to a process instead of “lending” it the `root` account.

Additional security tools such as SELinux and AppArmor also exist, are shipped with popular container tools and allow containers to be restricted even more.

3.2.1 cGroups

Control groups, referred as *cgroups* in the Linux Kernel [65] allow to group processes together and to control and monitor the amount of resources a group of processes (called a *control group*, hence the name) is allocated.

CGroups allow the control of many resources, such as the CPU time, memory, network bandwidth, and so on.

By default, container engines do not restrict the resources made available to the containers [66]. In some cases, this may be an issue as a container with leaking memory issues or one with malicious intents may end up taking all available memory, crashing other containers and potentially the host.

It is also important to understand that while cGroups is able to restrict the amount of resources a container has, it is not able to hide the resources themselves. This is shown on figure 3.2, which shows that while cGroups has restricted the number of CPUs of the container to 1, it is still able to see the two cores the container host has at its disposal. The number of CPUs the container is allowed to use is highlighted in green, while the number of CPUs the container can see is in red.

```
labo@master:~$ docker run -it --cpus="1" debian /bin/bash
root@8b2cc91122ba:/# lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         46 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  2
  On-line CPU(s) list:   0,1
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i9-10920X CPU @ 3.50GHz
    CPU family:          6
    Model:               85
    Thread(s) per core:  1
    Core(s) per socket:  1
    Socket(s):           2
    Stepping:            7
    BogoMIPS:            7007.99
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clf
                         lush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_pe
                         rfmon rep_good nopl xtopology cpuid tsc_known_freq pni pclmulqdq vmx ssse3
                         fma cx16 pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes
                          xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch cpuid_fault inv
                         pcid_single ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow vnmi flexpriority
                          ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx a
                         vx512f avx512dq rdseed adx smap clflushopt clwb avx512cd avx512bw avx512vl
```

Figure 3.2: `lscpu` showing the whole host CPU

As `lscpu` reports, sensitive information, including the CPU model, frequency and vulnerabilities, can also be found, either by a tool or directly from the `/proc` mount. However, this is expected behaviour

from cGroups, as it does not focus on enforcing a reduced view of the resources at hand. This still means information such as the CPU model, the number of cores or the available physical memory about the host cannot be hidden.

3.2.2 SecComp

As of this writing, there are several hundred system calls in the Linux Kernel [67]. As most programs do not use all available system calls, the ones not used should be disabled for security reasons.

Seccomp is used to limit the system calls a process is able to make, effectively **sandboxing** a program. This mitigates many attacks if configured correctly. However, seccomp [45] is neither automatic nor dynamic. Default settings are applied on a recommended base by each container engine : default settings may not be sufficient or in some cases, useless. Studies suggest that generating seccomp profiles may be a way to greatly enhance the usefulness of *seccomp* [44] [47], but this requires additional and external tools, as well as an understanding of how seccomp works beforehand.

An example of seccomp is shown on figure 3.3, with a custom seccomp profile. This profile restricts the use of both the `mkdir`, `socket`, and `connect` system calls, thus blocking the creation of a new folder and/or block any traffic through networking or processes, as socket or connect are used to either listen or connect to other processed, regardless of their location.

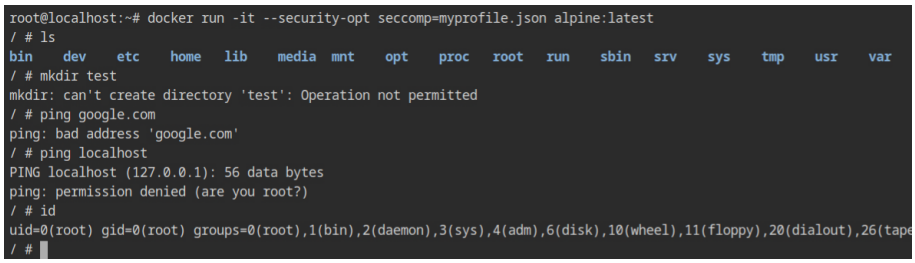

Figure 3.3: An example of a custom SecComp profile [68]

SecComp therefore allows a program to be restricted, even if `root` is the user asking for it. This is perfect for containers, as most containers simply run as `root` inside. Whoever the user is, system calls can still be restricted to the previously defined set by the container engine.

3.2.3 Namespaces

Namespaces allow for isolation of resources of many types. Linux supports many namespace types [10]. For instance :

- The *network* namespace, which allows to isolate network environments. This allows to create virtual network interfaces, separated IP routing tables, firewall rules and much more. [69]
- The *user* namespace, which allows isolation of user and group IDs. This isolation allows to specify different permissions to each process, even though they share the same user or group ID. This is very useful for containers : the root ID could be considered 0 in a container but would be actually considered 68394 in the host system.
- The *mount* namespace, which allows for "isolation of the list of mounts seen in each namespace instance" [70]. This means, from the container perspective, that each container will have its own filesystem that will be kept local to the container only. For instance, should the container unmount a drive, the other containers will not be impacted by this unmount.

Namespaces are the reason why containers cannot see their "outside", that is the container host. An example of namespace is shown with the different point of views on figures 3.4, 3.5 and 3.6.

```
labo@master:~$ pstree
systemd─┬─agetty
        ├─containerd───7*[{containerd}]
        ├─cron
        ├─dbus-daemon
        ├─dhclient
        ├─dockerd───8*[{dockerd}]
        ├─qemu-ga───{qemu-ga}
        ├─snmpd
        ├─sshd───sshd───sshd───bash───pstree
        ├─systemd───(sd-pam)
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-timesyn───{systemd-timesyn}
        └─systemd-udevd
labo@master:~$
```

Figure 3.4: `pstree` ran before running a container

Figure 3.4 shows there is no container running. By running `docker run -td debian`, a container will be created. The difference is obvious and can be seen in figure 3.5. The red square contains the container : the host can therefore see the main process and its children (in this case, as the container is recent, none are shown).

```
labo@master:~$ docker run -dt debian /bin/bash
91ab307ab4c733c5f73b42741f11f4b51223ff929d35fe761b0c94853c35a9aa
labo@master:~$ pstree
systemd─┬─agetty
        ├─containerd───8*[{containerd}]
        ├─containerd-shim─┬─bash
        │                 └─10*[{containerd-shim}]
        ├─cron
        ├─dbus-daemon
        ├─dhclient
        ├─dockerd───8*[{dockerd}]
        ├─qemu-ga───{qemu-ga}
        ├─snmpd
        ├─sshd───sshd───sshd───bash───pstree
        ├─systemd───(sd-pam)
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-timesyn───{systemd-timesyn}
        └─systemd-udevd───5*[(udev-worker)]
labo@master:~$
```

Figure 3.5: `pstree` ran while running a container

From the container, nothing can be seen, apart the main process launched by the container itself. This is shown on figure 3.6.

```
root@1f7b8a2c6760:/# pstree
bash
root@1f7b8a2c6760:/#
```

Figure 3.6: `pstree` from the container

The same can be seen from the networking point of view, as container and host do not run within the same namespace by default.

```
1 # On host
2 labo@master:~$ sudo arp -a
3 ? (172.17.0.2) at 02:42:ac:11:00:02 [ether] on docker0
4 ? (192.168.2.1) at 23:80:e0:67:1b:33 [ether] on enp1s0
5 ? (192.168.2.180) at 00:e4:6b:04:e9:99 [ether] on enp1s0
6
7 # On container
8 root@1f7b8a2c6760:/# arp -a
9 ? (172.17.0.1) at c4:1f:a2:52:f9:6e [ether] on eth0
```

In short, namespaces allow to create fully isolated “universes”, which are essential for containers. With this mechanism, not only it is possible to securely isolate the host from the containers, but container can also run in a brand new environment without any side effects.

3.2.4 Capabilities

Linux-based systems usually include two types of users : those with administrative privileges and those without. The super user `root` can do anything on a system, which usually means great responsibilities that cannot be shared with many (if any) users. Therefore, all processes that require the slightest administrative privilege would require `root` access. This is far from being ideal : should a program running with `root` privileges be compromised, catastrophic damages may incur, as `root` privileges are exempt from any restrictions.

This is what *capabilities* address : instead of a boolean access-control list (either root or not), capabilities allow certain actions to be granted while blocking others. As of this writing, 38 capabilities exist [71], each controlling a very specific aspect of the Linux Kernel capabilities. For instance, `CAP_SYS_MODULE` allows a program to "Load and unload kernel modules" [71]. Some capabilities are used much more than others, and all capabilities are not equal : `CAP_SYS_ADMIN`, while being dropped out, gives much more rights than any other capability.

As containers work on an explicit allow list for capabilities, many of them are dropped when a container starts. The capability set that is left allows all changes made to the container to remain within the container boundaries. However, giving certain privileges may break that isolation layer. For instance, giving `CAP_SYS_ADMIN` to a container would allow it to mount the host's filesystem. This could lead to catastrophic consequences as all files would be potentially exposed (the file containing the passwords, sensitive variables, other services or machines, etc.).

Container engines give *default capabilities* [72] [73] that greatly limit the capabilities of a container, allowing the container to run properly without interfering with the host.

Containers requiring additional capabilities, such as containers with specific networking needs (Virtual Private Networks (VPNs), web servers with integrated firewalls, and so on) almost always specify which capability should be added. However, some capabilities may not be required, either because it was left over, or because the container user does not need the feature requiring the capability. Regardless of the reason, adding capabilities to containers without having a good reason can be dangerous, as it *always* expands the attack surface of the container.

In general, capabilities are not enough to allow for a full breakout, and additional requirements are needed. However, adding capabilities blindly is already an extremely insecure practice as it may allow the container to perform unwanted actions. For instance, adding `CAP_NET_ADMIN`, which may be required for some networking applications, also allows the container to get out of the networking namespace and listen to traffic on the host's namespace.

3.2.5 AppArmor

AppArmor is a system working like SecComp. While SecComp allows to filter system calls, AppArmor is a Mandatory Access Control (MAC) working as a Linux Security Module (LSM), allowing to define an access control mask for a specific program. For instance, AppArmor allows to limit the access to the host devices, rendering device access from the container impossible without specific settings.

An example of AppArmor protecting the host is shown on figure 3.7. Two containers, each running with AppArmor on and one with AppArmor off, try to mount `/dev/vda1`. This drive contains sensible information, including the shadow password file of the host, as well as other files that could be used by attackers to compromise the host and the network.

```
labo@master:~$ docker run --rm -it  --security-opt apparmor=unconfined --device /dev/vda1 --cap-add=CAP_SYS_ADMIN debia
n
root@310d3b872de5:/# mount /dev/vda1 /mnt
root@310d3b872de5:/# ^C
root@310d3b872de5:/#
exit
labo@master:~$ docker run --rm -it --device /dev/vda1 --cap-add=CAP_SYS_ADMIN debian
root@c27490433e7d:/# mount /dev/vda1 /mnt/
mount: /mnt: cannot mount /dev/vda1 read-only.
       dmesg(1) may have more information after failed mount system call.
root@c27490433e7d:/#
```

Figure 3.7: `mount` ran while running containers

AppArmor is automatically deployed with popular container tools, such as Docker or Podman, but as with SecComp, a permissive profile is provided. It may be required to generate hardened profiles [74] for better or more container-specific results.

3.2.6 SELinux

SELinux, short for Security Enhanced Linux, is a LSM written in collaboration with the National Security Agency (NSA) [75]. It provides similar features than AppArmor, but is considered to be more extensive. SELinux works by assigning label to files, or processes. Labeling files allows to group data of the same label. Then, only users with access to the specific label can access said files or processes.

SELinux is considered to be way more complex than AppArmor by the system administrators community, but due to its flexibility, it is popular and often used in production environments.

A good use-case of SELinux would be in environments where servers run containers with different sensitivity levels : SELinux would allow a complete separation between the sensitive containers and the ones that are not.

The diagram showing the behaviour of SELinux is shown on Figure 3.8.

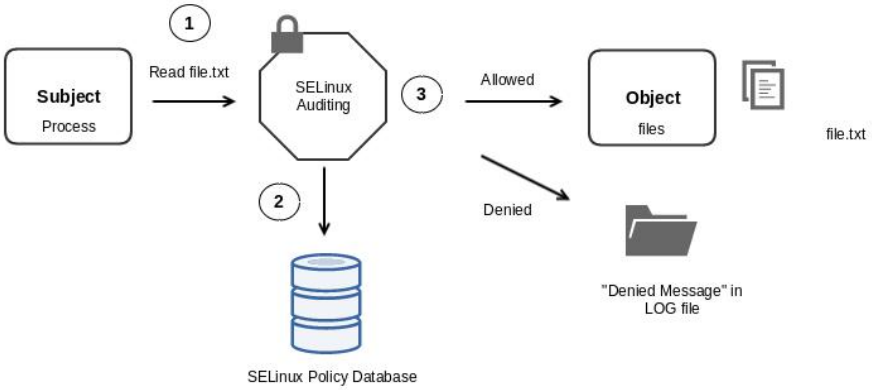

Figure 3.8: Behaviour of SELinux [76]

SELinux works in two modes : *permissive* and *enforcing*. Permissive mode allows to set up proper policies, as this mode will log everything that is happening with the tracked files. Enforcing, on the other hand, will block calls outside the policy. Both modes will log system calls : regardless of the SELinux mode, discrepancies could quickly be tracked down, should monitoring tools be deployed and configured properly.

For instance, a container that is allowed to read specific files but suddenly tries to access files outside that scope may be compromised. SELinux would therefore not be able to prevent containers to be attacked or compromised, but could certainly block further progression from attackers.

However, SELinux policies need to be written “by hand”, as containers do not provide a general-use policy nor turnkey solutions for this kind of LSM. The publications seem to indicate that it is possible to a certain extent to detect configuration errors [77], but the lack of publications on the automatic generation of SELinux policies seems to indicate that this remains a manual process.

3.2.7 OS containers and App containers

While containers in the industry mainly focuses on containers containing a service (an app), there are actually two categories of containers that exist.

- *App containers* such as the ones that can be found on the Docker Hub, are containers meant to host a single service. Docker, as well as Podman, and generally most container tools, generally provide app containers.
- *OS containers*, which do not focus as providing a service but rather an entire OS. Altough those containers are less frequent, they do provide a complete OS. This may be useful for application

that are being developed, datacenters, and so on. LXC, as well as Docker, are able to run OS containers.

3.2.8 A note on container hardening

Many containers provide configuration suggestions, recommendations for specific uses, and so on. However, little to no container editors provide made-to-measure SecComp, AppArmor or SELinux profiles with their containers. We suppose this is because the general profiles are working and that this part is exclusively considered for administrators to handle.

Regardless of the reason, there is little help from the community for custom-made profiles. This essentially means that very little users change the default profiles of the container, which is overpermissive for many uses.

3.3 Industry use of containers

Statistics show a massive use of containers throughout the industry, whether it is with Cloud Service Providers (CSPs) or in-site infrastructures [42]. In 2023, 55% of StackOverflow users responded Docker was their most used tool [78]. However, container users do not manually interact with the kernel modules, nor do they create processes manually. This subchapter focuses on the entities present in container architecture.

3.3.1 Container architecture

The main entities of the two main container tools are shown on Figure 3.9. Docker and Podman are amongst popular releases of container tools.

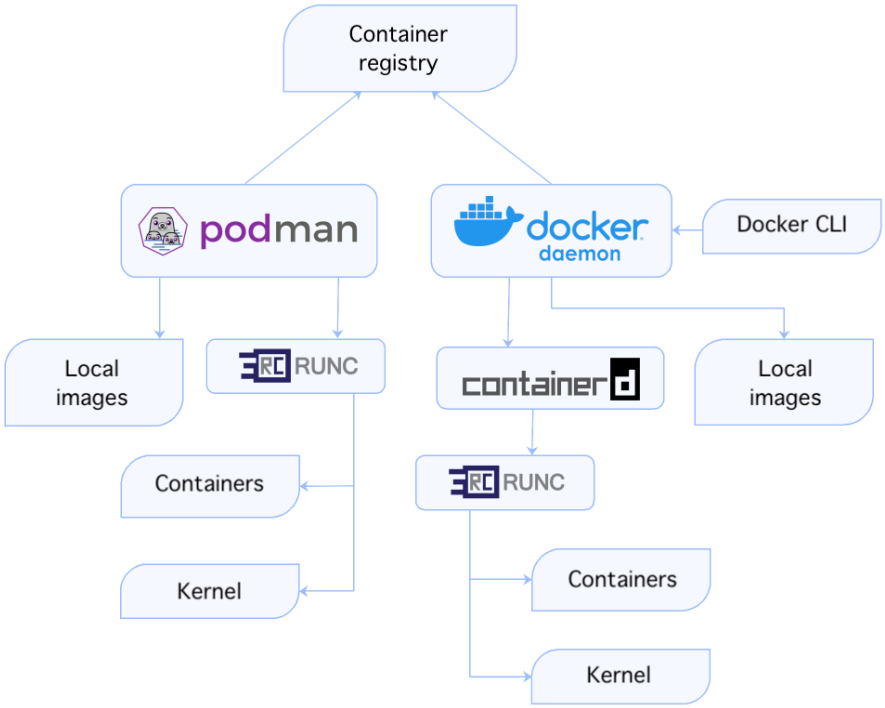

Figure 3.9: Translated container architecture diagram for two popular container tools [79]

As shown on figure 3.9, Docker and Podman are only high-level tools, which interact with *container runtimes*, such as *runC* or *containerD*. Docker[1] uses both a command-line interface and an Application Programming Interface (API) to allow users to create, maintain and destroy containers. High-abstraction container tools such as Docker depend on two repositories where containers are stored : the local directory, which usually acts as a cache, and a *container registry*. This container registry can be seen like an *App Store* where users publish containers that have not been instantiated yet, called *images*. Each image is a complete container that can be executed. The most known container registry is the Docker Hub, where hundreds of thousands of images are available to everyone.

Simple images exist, such as web servers, but more complex images are also available, some of which contain corporate secrets or sensitive informations [5] due to accidental publishing to the Docker Hub, or incorrect scrubbing of sensitive data.

Designing containers should, as every product should, contain the evaluation of a risk model. This would allow to perceive how much time would be required to patch containers should a security issue arise. In case where the risk model indicates that the time required is high, shifting to alternative solutions may be in the interest of the container administrator. For instance, rather than simply leaving containers, the impacted containers may be put in Virtual Machines (VMs) to limit breakouts. While not

[1] For the rest of this chapter, our work will only refer to Docker for simplicity reasons, but statements can be extended to other container tools such as Podman.

all container engines support multihost networking, VMs providers usually provide the guests with a software-defined networking service that may be of use to bridge all containers, as if they were on the same host.

3.3.2 Popular container tools and runtimes

Several container tools exist, including ones that are extremely popular. StackOverflow's 2023 developer survey lists Docker as the most used tool [78] in all respondents, with other container tools such as Podman listed way down in the list.

It is known that Docker is widely used for containers, as most infrastructures found on the web propose a *Docker Compose* file, a file defining a group of containers into an architecture. *De facto*, this file has become the standard in the industry [80], pushing other container tools to use this format as well.

Docker, which offers a wide set of features, from container port redirection to the host, volumes to link files from the host to the container and vice-versa. Docker containers run with a daemon, which itself is executed as `root`. Therefore, by transitivity, all containers are run as `root`. This is heavily criticized [79] [9], as an attacker breaking out of the container would gain a significant foothold on the container host.

This is fixed by other container tools such as Podman, which runs containers on behalf of the user running the container. Those containers are called *rootless*. However, containers sharing the host's kernel cannot be considered as fully isolated [81].
Other tools, such as Kata, which runs the container from the inside of a VM for each container. An example of Kata containers is shown on figure 3.10. Publications suggest that Kata containers are by far the most secure [82].

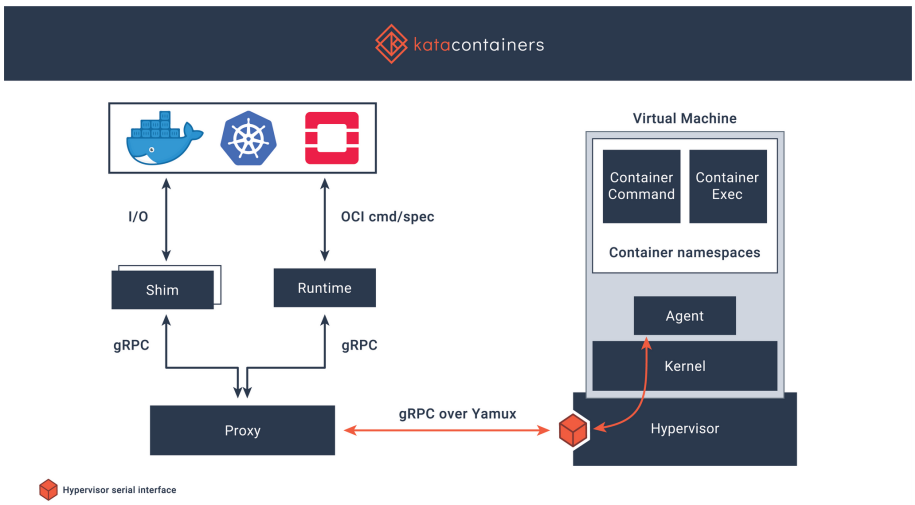

Figure 3.10: Scheme of Kata containers [83]

Other container tools, such as Nabla containers or Amazon Firecracker, also address the issue of the host kernel by adopting an attack surface reduction by using unikernels. As with unikernels, instead of including a complete OS inside of it, a library implementing most system calls is added. This results in only a tiny fraction of system calls being used.

Finally, tools like gVisor find a middle ground, where the container runtime is replaced from `runc` to `runsc`. Like with Nabla containers, system calls are captured and redirected to a specialized process. Only very few system calls actually make it to the host.

Despite all efforts made to ensure containers can be trusted as isolated sandboxes, Docker still remains the most used container tool. Technologies like Firecracker, Nabla, or Kata are mostly used by CSPs to ensure an optimal level of isolation in between each tenant. However, most containers developed or hosted in an organization will almost always be developed with Docker.

3.3.2.1 Container runtimes

Docker as well as Podman are tools designed to interact with low-level components, themselves interacting with the Kernel to run containers. In order to run containers, a *container runtime* is required. As with container tools, there is more than one tool available, as suggested on figure 3.9.

On the lowest level, `runc` is responsible for “for spawning and running containers on Linux according to the Open Container Initiative (OCI) specification” [84]. Other projects compatible with the OCI also use alternatives such as `crun`, an equivalent of `runc` written in C instead of Go. As Red Hat specifies, using a runtime that is not `runc` also allows to add experimental features, as other runtimes are less

bound by the OCI specification and thus is allowing those runtimes to be considered as playgrounds [85].

Outside of the OCI, LXC also allows the execution of containers. However, LXC focuses on OS containers, that is containers that aim to be “as close as possible as the one you’d get from a VM but without the overhead that comes with running a separate kernel and simulating all the hardware.” [86].

Other tools, considered as *high-level* can be run over `runc`, as previously shown on figure 3.9. For instance, Docker runs by using `containerd`, a daemon that is responsible of the download of remote containers from the Docker Hub; it then hands over the task of running the image container to `runc`. `containerd` is not the only high-level container runtime there is available. For instance, `cri-o` is used in another container runtime designed for orchestration purposes.

3.4 Orchestration

Most organizations will agree that one container host may not be enough, should it be for organizational, redundancy, latency or other technological reasons. As Docker and similarly other container tools have an API and they are able to be controlled remotely, it is therefore possible to create a cluster of container hosts to distribute the load of containers across said cluster. An orchestrator would control which container goes where, and so on. As of this writing, three main orchestrators exist :

- Docker Swarm, which allows a cluster of hosts running the Docker engine to run containers with load balancing, scaling, and much more. As expected, Docker Swarm only accepts Docker as the container tool running. A scheme of a Docker Swarm cluster is shwon on Figure 3.11.
- Kubernetes, which is the most used container orchestrator nowadays, whether on-site or in the cloud. Kubernetes does not limit the technology that is used below it, as long as it can run within Container Runtime Interface (CRI) specifications, such as `cri-o`. This allows to use a wide variety of container technologies, should it be `runc`, `containerd`, Kata containers, and so on. As Kubernetes introduces many terms that go well beyond the scope of this work, a typical Kubernetes schema will not be shown.
- OpenShift, which is an Kubernetes overlay with additional features on top of it.

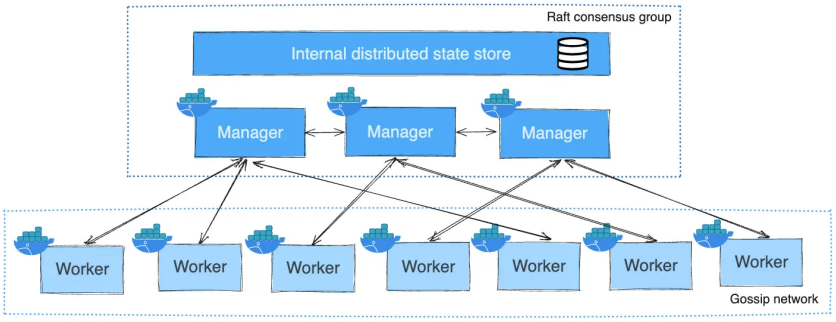

Figure 3.11: Scheme of a Docker Swarm cluster [87]

Kubernetes is the *de facto* standard in organizations nowadays. Big CSPs provide Kubernetes infrastructures ready for use [88], and organizations from various sizes also use Kubernetes [89].

While container orchestrator administrators will agree that it is essential for an organization to have orchestration over the container cluster, orchestrators usually add services on top of the container technology, which spreads the attack surface further [90].

As container orchestrators add a layer of complexity, that we will only implement attacks focused on the container and not the container orchestrator, and that it is our belief that most attacks running on containers not within a container cluster will most likely work within one, our work focuses on containers that are on a single host, running a container tool without any additional orchestrator.

3.5 Conclusion

This chapter presented the many parts present in a container environment, from the container kernel modules to its orchestrators. This knowledge is critical for a proper understanding of containers, especially in a scenario where containers are going to be scanned and tested for vulnerabilities.

The first part of this chapter focused on the container basics, mainly its kernel modules, by presenting and briefly indicating the potential threat sources that could emanate from each one of them. Our work will focus on the first part of this chapter, by scanning the container and searching discrepancies by comparing the container to a default configuration baseline. As we explained within each Kernel module category, each of those modules have weaknesses that can be introduced quickly by a container administrator simply by changing an option at runtime.

The second part of this chapter focused on the specificities that can be found in the industry, mainly the different parts present in a container platform. This part explained the different runtimes programs that could be found on the market, as well as the most popular ones.

This chapter concluded by showing the different tools at everyone's disposal to orchestrate more complex infrastructures with orchestrators, and underlining the additional notions required to run a cluster.

4 Automated Exploration

4.1 Introduction

Automated exploration refers to the automatic collection of data for later use. This process is used in many different scenarios, for instance collecting performance data in datacenters [91] to avoid the manual collection of data. In our work, automated exploration focuses on finding plausible information to determine the environment the container is running in. This process is paramount for this tool to work, as most attacks required to break out of a VM or container always require some form of known information. The automatic exploration of a container should allow our tool to determine if any attacks may allow to break out of the container.

Most information gathering about a container and its environment (its host, any neighbour, the container engine, and so on) can be collected automatically, as informations are stored in most cases in the same directories, files and structures regardless of the underlying Operating System (OS), as long as it is a Linux kernel. For instance, reliably detecting if a process is inside of a Docker container can be done by checking a couple of files and kernel structures [38].

As previously stated, containers do provide a layer of isolation in between the host and the guest (here, the container), although not as strong as Virtual Machines (VMs). This can be exploited to an attacker's advantage to either gather information or to attack the host with a knowledge of the current environment. As there are many ways to compromise a container, our tool should gather as much data as possible, without arising suspicion, which means primarily using passive detection. For instance, scanning the ports of a container might reveal useful information, but may also trigger security mechanisms as port scanning is almost always considered as an attack.

Every bit of information is beneficial to an attacker as the isolation layer is usually considered as insufficient [92]. Attackers do not have a single point of entry towards the host, as four categories of attack vectors have been considered by security researchers [37] :

- Applications inside the container
- Inter-container protection,
- Protecting the host from containers
- Protecting containers from a malicious or semi-honest host

This list suggest that not only the container as well as communications are at risk of being attacked. In some cases which we will not study, the host might also be the attacker.

This chapter studies the information retrieval process of our tool to underline how much can be retrieved and with what effort. In short, our tool tries to address the three following questions :

- In what environment is our tool working in (information gathering) ?
- What tools does the container have at its disposal (contextualisation) ?
- Can any of these tools be used to the tool's advantage (attacks) ?

Each question will be described and answered in the following subchapters. Potential solutions will then be presented in the following chapters.

4.2 Prerequisites

In order for our tool to run, several requirements must be met, as they are situated outside the field of study of this work.

- The first one may seem obvious but is required in order to run : the operating system must run a Linux Kernel. Microsoft Windows and Apple OS X are not supported, mainly because of the many changes they've imposed on their OS and the compatibility issues this incurs.
- The second one is `root` permissions. As this tool nor this work focuses on privileges escalation, the tool expects `root` privileges, but may be able in some cases to run without it. While most vulnerability tests do not require `root` privileges, exploits attacking specific parts of the underlying OSs may require `root` access. This also removes permissions concerns in the container for our tool, as this concern is outside the scope of this work.

4.3 Information gathering and OS exploration

There are many different ways to mine information in a container and its environment, as there is a myriad of ways to analyze an OS. While this tool does not claim to offer a complete list, our tool should provide enough tests to reasonably ensure that the results are not biased.

4.3.1 Informations about the container host

Most container engines reveal many details regarding the container host by default. This lets the attacker know information about the host, such as the number of cores, their frequency, available physical memory, and so on. Those informations are usually kept confidential from the public.

This also allows to predict whether or not the container runs on enterprise-grade hardware as it is for instance uncommon to see more than 64 Gigaoctets of physical memory in home computers, or by looking at the model name of the host's Central Processing Units (CPUs). The prediction, although naive, can be done easily with a threshold of available resources, or by checking the CPU's model against a list of processors found in enterprise environments such as Intel Xeon or AMD Epyc.

Apart from hardware, information about the container engine can also be detected. Container engines tend to leave clues in the container , which helps to detect if the tool is running in a container, and if so, with what container engine. For instance, Docker leaves a file at `/.dockerenv` for legacy purposes [93], but this also lets an attacker know about the chosen container technology for potentially all containers as well.

As containers do not provide enough isolation, it is also possible to determine where the container data is stored on the host with `mount`. This can become a problem if a breakout of the container succeeds, as a path for a valid program is now known. An attacker would then be able to call programs from the container but on the host, compromising the host (with, for instance, a reverse shell).

4.3.2 Informations about other neighbours

Information about other neighbours can be collected as well : Address Resolution Protocol (ARP) can be checked by our tool to see which container exists in the neighbourhood of the container. This gives our tool the ability not only to know what containers are reachable, but it may also give us a clue of their use, as it is common practice to name the container as the main service running inside of it. For instance, a container named `nginx` will most probably run the `nginx` web server. Container names may be collected by running a reverse Domain Name System (DNS) query to the container internal DNS as long as one exists, which is in most cases left on.
The ARP table may also contain information about container networks, the MAC address or the name of the container host. In specific cases, such as linking the container directly to the host network, other physical devices may be seen in the ARP table.
Using the ARP table also removes the need to scan the network for hosts as the ARP protocol is handled either by the OS or by services running on the container. Only DNS packets are sent to the internal DNS container server, which is generally unmonitored; moreover, even if the DNS server was monitored, only legitimate requests are sent towards the server, and only a slight increase in the amount of requests sent by the container (one per found neighbour) sent may draw attention.

In a similar way than with hardware, it is possible to predict whether or not a neighbour is a container based on its Media Access Control (MAC) address. Container engines will in most cases generate a Organizationally Unique Identifier (OUI) nonexistent prefix. This behaviour can be detected simply by checking if the OUI matches a container prefix group. Other MAC addresses can be considered as external devices linked to the container networks.

4.3.3 Informations about the container itself

So far, information the tool collected were about devices other than the container (the host, neighbours and other physical devices). Naturally, as the tool has access to at least a container (that is, the container in which it is running), it is also possible to collect information about it.

Many information about the running container can be collected, such as the running services, the OS, and any file contained by the container. Files that may help our tool to break out are files owned by the container's host, and shared by a volume. As volumes are shared by the mount namespace, it is easy to detect which exposed volume contains patterns of sensitive files, such as the container control socket or the host's system files. Our tool can therefore list the mounts of a container and then check which ones match a specific signature (for instance, whether system files are present). This allows to quickly find sensitive files. This method can also be used to find information about the host, for instance what services are running based on the files the container shares with the host.

Information previously collected about other containers may also give hints on the use of the container itself. For instance, web servers usually have two network interfaces, one leading to the container host and the other connecting other containers that are not exposed to the outside. The number of interfaces of a container may therefore lead an analyst towards certain conclusions.

Important information about the host can also be found in `/dev`. As this repository contains devices connected to the container, it also allows our tool to find devices connected to the host, such as Graphics Processing Units (GPUs), hard drives, and so on. Any exposed devices represent an opportunity to test breakout techniques, and although not all of them can be implemented in this work, this still represents valuable information for the analyst using this tool.

There are dozens if not hundreds other collection points, each giving valuable information about the host, the container, the security behind it, the networking, and so on. Most if not all information can usually be found in `/proc` or `/sys`, as they represent the kernel data structures. As this tool expects to be run as `root`, collection from all data points is trivial.

4.3.4 A note on determinism

All containers are different from another, and many security practices exist. Given the near-infinite permutation of settings and possibilities in a container engine let alone the host's OS, this tool cannot predict with absolute certainty what a container does or its weaknesses, features, neighbours and so on. Moreover, an evaluation is only valid at the time of execution, as resources not used at the time of evaluation can start being used at a later time, for example, and this would result in our tool not seeing specific information.

Given the multitude of environments and events a container may encounter, errors may occur in

some cases, which could cause our tool to crash. This is linked to the large variety of containers, OSs and settings. While it is normal not to be able to handle every occurring error (as most will raise at runtime and not during compilation), this is unacceptable behaviour. All tests must therefore be able to handle errors in a way that if the test fails because of an error, the test is skipped. This may reduce the overall certainty of the tool at the end but this also allows tests to continue rather than crash the tool altogether.

Our tool should be deterministic as it should give the exact same result when ran on two exact copies of an infrastructure. However, many external factors may influence the result of the tests. For instance, if the internal container DNS server is unavailable, the results may be drastically different from one analysis to another, even when ran on exact replicas of a container infrastructure. This does not mean our tool gives random results, but rather than results will ultimately need to be studied by an analyst to be of actual use.

4.3.5 Container environment scoring

As described before, all containers vary in many ways from one another. However, most information required to understand the environment of a specific container are constant, and change only if the container is re-instantiated. It is therefore possible to run a set of tests to give an *environment prediction score*. This prediction score can be interpreted as a certainty score, which becomes useful for an analyst to quickly understand what is certain and what is conjecture. Each test will return a score between 0 and 10, where 10 is the strongest certainty and 0 is the least amount of certainty. The certainty score for each test will indicate with how much certainty the test result can be trusted.

For instance, detecting if a container is running with Docker can be detected with the `/.dockerenv` file, as mentioned before. While this usually means that the container is indeed running with Docker, any user or service could have created that file. Therefore, this test cannot receive a 10/10 certainty score.

Only very specific tests will therefore be able to give a 10/10 certainty score. This will ensure that a quick test is never as valuable as one that will account for complexities.

4.3.6 Environment collection process

As previously mentioned, it is impossible to predict on which environment our tool will be ran on. This means we must consider that no tools are at our disposal. For instance, `lsblk`, `mount` or other tools commonly present in a Linux system must be considered absent, as some containers may work with only required tools, with very light containers such as `alpine` or `busybox`. Most of our information must therefore come directly from the kernel, or from tools that can be included in ours in order for the

container to work. Networking tasks are not impacted by this as we consider that all containers have a properly working networking stack since a container is of little use without one. However, handling information such as mount points, namespaces, and so on gets trickier without tools.

This in turn means that our tool cannot rely on much and that our work must implement some mechanisms itself in order to work properly. For instance, as our tool is developed in Rust, crates (modules distributed by the community and third parties) may not be used in some cases, as those crates will simply run a tool considered to be installed. This works as long as the tool is present, but as we assessed previously we cannot rely on this fact. For instance, a crate designed to list the networking neighbours may rely on the `ip` command, which is not always installed.
Furthermore, crates may have undetermined behaviour, either malicious or not, which could broaden our tool signature, effectively allowing security tools to detect the presence of our tool more easily.

Moreover, most of the data required for our tool to run can be found in the `/proc` filesystem. We can find information about Linux capabilities, ARP Tables, mount information and much more. As this filesystem is always present, we may rely on it to obtain key information about the container. This does mean that information must be processed in order to understand it, as kernel structures are never presented in a serialized format such as JavaScript Object Notation (JSON).

A subset of the information collected by our process is shown on figure 4.1. Many data collection points are taken into account, and each one is then converted to an interpretable format. All information are then collected into a *test set* : this set allows our tool to determine the environment in which it is running. The test set first tries to determine the container engine currently running. Relevant tests are grouped together by category and evaluated. As each test returns a score from 0 to 10, the test group with the biggest score is deemed the most correct. A certainty percentage is then computed by using the total number of points from the sum of the certainty scores from each test, divided by the maximum number of points from each test group. Once the container environment is guessed, the host hardware is evaluated, collecting the number of cores on the CPU, the amount of Random Access Memory (RAM), the number of Network Interface Cards (NICs), and other information about the host, such as the mount point of the container on the container host.

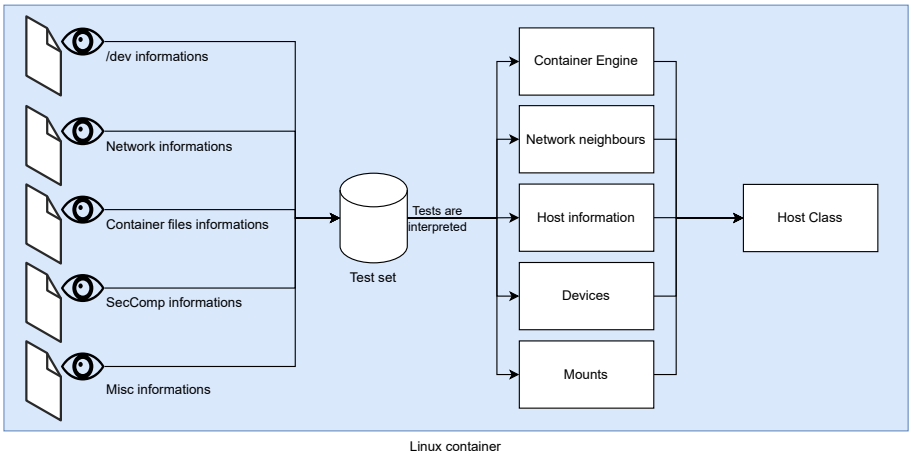

Figure 4.1: Collection process

As mentioned, the collected information will be used for attacks later on, which means all data must be checked and refined so there is no doubt about their accuracy. For instance, attacks targeting the Docker socket will not work with Podman, as Podman has a different socket.

Once complete, all information are grouped into a *host class*, containing the gathered and extrapolated information. This will be used later on by the security tests to determine the overall security of the container.

4.4 Contextualization

As previously discussed, many information can be extracted from a container, data that can be useful to direct an attacker towards potential weaknesses. In this work, exploits allowing breakouts are considered as information and must therefore be tested. As one of the goals of this tool is to guide analysts to harden containers, exploits considered as dangerous for the host or the container will not be implemented. For instance, exploits using hardware (such as Peripheral Component Interconnect (PCI) cards) with faulty instructions are considered dangerous as it may accelerate the deterioration of the hardware, which may in turn cause failures with actual physical consequences. Alternatively, using Kernel structures to break out of a container is considered acceptable, as long as there are no risks of crashing the Kernel in itself.

Once the environment collection process is over, the actual exploitation of data is the logical next step. In order to avoid attacking the container blindly, our work proposes a contextualization system, which routes any gathered information to the proper breakout test. This allows the tool to focus on areas where there is plausible information that weaknesses exist, and ignore areas where confidence is low

or nonexistent. For instance, there is little sense trying to break out of a container by trying to exploit a peripheral, if there are none mounted on the container.

Contextualization must therefore rely heavily on the information gathering process described earlier. Only a collection of useful, correct data may assist the contextualization engine into fulfilling its mission. In order to route the collected information towards the proper breakout test, each test collecting some context will be categorized in at least one category. Then, only categories exceeding a fixed threshold will be run. Theoretically speaking, our work should not try to break out of the container via a peripheral if no tests support the possibility of breaking out via this peripheral. Contexts tests are different from environmental tests as they do not possess a score but a pass/fail system : tests pass when the default setting is detected and they fail when a setting has been altered from the default value. For instance, a test checking exposed devices will fail when a GPU is exposed to a container.

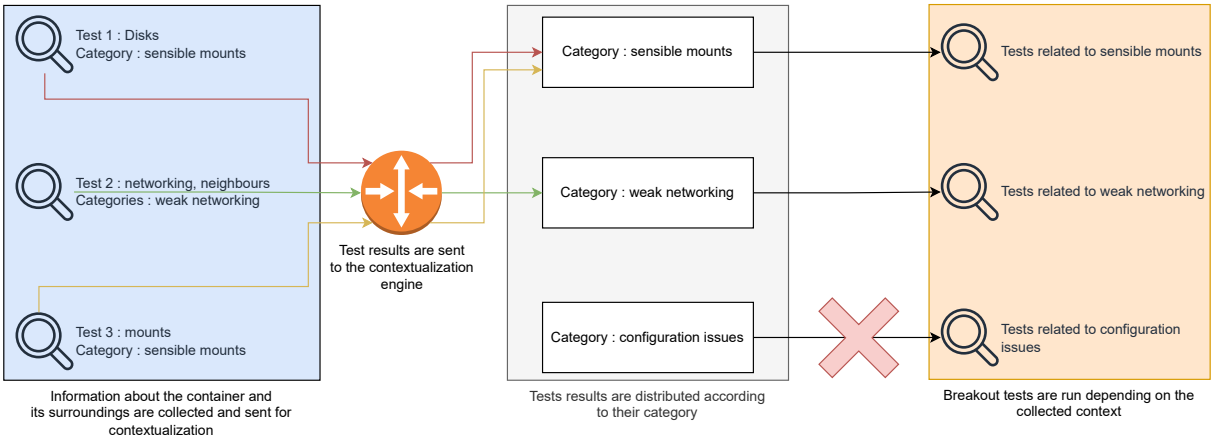

Figure 4.2: Contextualization engine

This process is shown in figure 4.2. It is important to note that the contextualisation process is predictive, like the information gathering process. In some cases, the information extracted from the context is closer to an extrapolation, and just because the contextualisation engine is based on several dozen tests does not necessarily mean that there is a weakness in the container. This is where attacks will check if the container is at-risk or not.

Some edge cases are to be considered : for instance, if the collection process fails, all categories from the engine will be considered empty, and no context can be extracted from the container. In this particular case, it is either possible to run all breakout tests or to stop the tool altogether. The latter has been chosen, mostly because running all tests would certainly be noticeable or could in some cases damage either the container, the kernel or underlying hardware.

The contextualization engine serves two purposes : the first, described before, serves the tool to orient itself in order to test breakouts on specific detected, vulnerable spots. The second is dedicated for the

analyst : a thorough report generated by this engine allows the analyst not only to see where most vulnerabilities were found, but also where they were not. This is valuable for the analyst, as this allows to see where the container may have been hardened, and where it was not.

An example of contextualization is the readings of `/proc` to compare the capabilities our tool has at its disposal. A test is run depending on the container engine, as capabilities change depending on the engine. If the gathered capabilities are not equal to the expected capabilities, the context test fails, which increases the `CAPABILITIES` category in the context engine. If the context engine has sufficient tests that have failed, all attacks regarding excessive capabilities will then be launched.

Once the context engine is loaded and populated, attacks can now proceed.

4.5 Attacks

This subchapter will focus on the attack process in its theoretical form.

4.5.1 Attack process

The attack process refers to all trials that may allow our tool to break out of the container and effectively access the host resources, including but not limited to other containers.

All attacks are executed similarly. Before the attack is launched, checks are made to ensure that the minimal criteria are met. Once this first phase, called *basic requirements* is done, *advanced requirements* are checked, checks that are too specific for the context engine to be registered, but need to be checked : as a reminder, the context engine is only there to register discrepancies and to allow specific categories of attack to run. Specificities are not registered by the context engine, which is why two phases are required.

Once the *advanced requirements* have been checked and met, the attack can proceed. The actual attack may depend on existing tools, such as shell code, or other executables. To avoid downloading files, which is usually the first step of a malware attack, files are byte-included into our tool's executable. While this makes our tool heavier, it also removes the requirement for networking and compiling files directly in the container (which could also trigger security mechanisms). In some cases, it also allows us to statically compile C files, removing the need for `libc` libraries and hence ensuring a wide compatibility over containers, regardless of the environment.

In cases where the attack depends on files, files are dropped from our tool's executable to the disk. To ensure compatibility, the files are written either to `/root` or to `/tmp`. Although unlikely, in cases where this is not possible, the attack is skipped. Finally, the attack is executed (either with the help of the dropped files, or without files if the attack does not have any file requirements).

To ensure the attack has succeeded, *final checks* are done to ensure breakout has properly occurred. For this, the `/etc/passwd` file is recovered from the host, as this will ensure the breakout succeeds with both Docker and Podman, since Podman containers are typically run as unprivileged users. Fetching `/etc/shadow` would therefore not work, even if the breakout succeeded.

If the attack has succeeded, a security score is returned, the values of which are described in the next subchapter. A security score can also be the *None* value, returned if the attack has not succeeded or if the prerequisites are not met.

The attack process is shown on figure 4.3. This process, although simple, should allow extensibility for new attacks as requirements, regardless of what they are, should be able to be integrated easily.

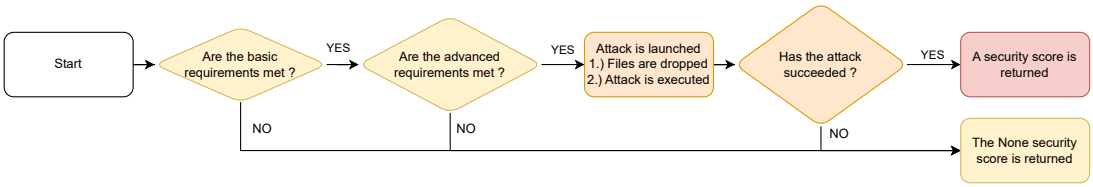

Figure 4.3: Coarse diagram of the attack process

4.5.2 Attack scoring

Each attack will then return a *security score*. In order to limit the number of possible results, a scale must be chosen. Common scores to measure the degree of vulnerability already exist. According to the Common Vulnerability Scoring System (CVSS) website, their scoring system “provides a way to capture the principal characteristics of a vulnerability and produce a numerical score reflecting its severity. The numerical score can then be translated into a qualitative representation (such as low, medium, high, and critical) to help organizations properly assess and prioritize their vulnerability management processes.” [94].

While this score may be of use, this score is supposed to highlight the criticality of a vulnerability in software. Our tool does not find vulnerabilities in software *stricto sensu*, as it mostly finds weaknesses due to bad configuration. It would be therefore incorrect and potentially unsanctioned to use the CVSS score for our needs.

Other tools used by scanners in other domains, such as the SonarQube © for static code analysis use a score derived from the academic grading in the United States, starting from A (the best grade) and ending at F (the worst grade). The grade E, precisely as in the United States grading system, has no value.

Even if the SonarQube has nothing to do with our work, this grading system seems adequate, as it provides enough possibilities to let an analyst know how critical a problem is. Our tool will therefore

output a grade based on the US Academic grading system, where grades represent the security score of a container. A container with grading F should be considered compromised, where A shows an adequate level of isolation between the host and the container to resist trivial attacks. It is important to remember that this grade will only consider specific factors implemented in the tool and that weaknesses that are not observed by the tool will not influence the grade. Many factors may be taken into account for this grade. However, all criteria must be related to the current container infrastructure. For instance, a breakout may be considered as the grade of F, but blatant security issues on the host that cannot be exploited from the container will not change the grade at all, as they do not reach the scope of this tool or its container grading system.

Each grade can be matched to an explanation as shown by table 4.1.

Table 4.1: Mapping of grades to a human description

Grade	Explanation
A	No misconfigurations were found within the container. No breakout of any form have been found, and no additional information about the host were found.
B	Additional information about the host were found. This may lead to further problems but automated attacks will likely not be able to push the issue further.
C	Information regarding the container suggests that a breakout may be possible with additional dependencies. The isolation layer of the container is thin and may break.
D	Misconfigurations that would allow a breakout have been found with the container, but automated attacks have not been successful. Manual attacks would very likely succeed.
F	Misconfigurations have been found within the container, and at least one breakout has succeeded with this container. Both the container and the host are compromised.

4.5.3 Available attacks

Given the many tools at our disposal on the Internet, there is a literal arsenal of available attacks, from Common Vulnerabilities and Exposuress (CVEs) to trivial breakouts. We have decided to implement three trivial attacks in our tool, that are the following :

- Excessive capabilities, with `CAP_SYS_ADMIN`, `CAP_DAC_READ_SEARCH`, `CAP_DAC_OVERRIDE`, `CAP_SYS_MODULE`, allowing automated breakouts or little additional requirements to breakouts
- Privileged container breakouts with `--privileged`, that allows an immediate breakout over the host as privileged containers are not containers
- Sensitive mounts, such as `/run/docker.sock`, that allow automatic complete control over the host

As capabilities and privileged containers, as well as volumes are amongst the most commonly used settings with containers, attacks relying on those mechanisms seemed logical to implement. Moreover, those settings are supported by an enormous variety of container engines. This would therefore allow to port those attacks on different container engines with little to no effort.

4.6 Strong points and pitfalls

This tool does present interesting features that may be considered as appealing to the analyst, but some others may be considered hindering or even missing from the tool.

As previously discussed, a hard requirement for this tool is the `/proc` filesystem, as there are many exploitable data available that can be used to our advantage. Instead of using external dependencies, the tool can access `/proc` directly. Therefore, much of the parsing and processing are handled internally. The main advantage of using a minimum of external dependencies is that there are very few limitations with this way of proceeding, as opposed to using a third-party crate, as a crate may not implement what our work is after or cause unforeseen side effects.

Most of the tool's requirements need to be included inside of the final binary, as this tool cannot guarantee components, libraries, executables or other requirements for this tool to work to be present on the container. It is an advantage to be able to run on most likely any OS, but this does mean a binary executable of a significant size. As of this writing, the file is about a few dozen megabytes, which is relatively large for a tool of this size. It also means that complex tools requiring several libraries may not be included and may need to either be compiled "on-site" (that is, in the container) or downloaded from the Internet. Either way, this would surely be flagged by security installations as malicious activities.

Another pitfall of this tool that will come as no surprise is that the tool only tests what it was programmed to. This means that new weaknesses will not be detected if is not kept up to date. It also means that in some cases, detection may fail to detect weaknesses and mark the container as “safe”. Inherently, this causes this tool to be vulnerable to the test of time.

In fact, most tools that were previously listed can find the same information. However, as of this writing, none of them have any significant contextual engine, nor do they have a detection system that works inside the container like ours. Most container security tools, such as Falco or Docker Security Bench, work from outside the container, which gives them a significant advantage over ours. Our tool, while disposing of less information, is still able to conduct attacks based on data and extrapolation. We do not comment on the results which will be discussed in the next chapter, but we are able to achieve what other tools do, only with less information.

The implementation of attacks may be considered bare, as many other attacks, including trivial ones, do exist. However, as our goal is not to provide every single attack but rather to assess how much effort is required to break out of containers. As the aforementioned attacks are trivial, it is certain that attackers will implement attacks that can be considered more complex. If our tool manages to break out of containers using trivial attacks, there is an enormous risk for other containers to be broken out by determined attackers. Implemented attacks may be considered meager in number, but they do concentrate on the most commonly used settings, and those attacks have yet to prove their efficency, which will be done in chapter 5.

5 Audit on sample infrastructures

In order for our tool to be evaluated, several infrastructures are hereby proposed. Each one possesses a different level of security, from one oblivious to it to another with default permissions. Infrastructures should be as generic and inspired by public repositories as possible. This would allow a proper comparison between existing infrastructures and the one in our simulation environment.

There are several goals of auditing sample infrastructures :

- The first one is to see how much data can be gathered;
- The second one is to see what can be done with the collected data;
- The third one is to determine what can be done to mitigate the detected risks.

Sample infrastructures will be run on several container engines. While it is impossible to predict whether a container engine will reveal more information than another, it is quite possible to detect whether our tool is engine-agnostic or favouring a specific container tool. Comparing the results of the two container engines will allow us to answer this question.

5.1 Infrastructures

This subchapter will list different infrastructures. For each container, a report will be generated and analyzed.

For each infrastructure, a short description of what it does will be given, along with a figure to show the different containers and networks of the infrastructure. As all containers are interconnected in a network, connections between containers of the same network are not shown.

5.1.1 Basic website

This infrastructure contains three different containers, as shown in figure 5.1.

- One handles incoming and outgoing traffic with a `nginx` container; This container hosts PHP files for a web app.

- The second runs a `php:8.2` server to handle the processing of PHP files sent by the `nginx` container.
- The third and last container handles all database-related jobs by running a `mysql` container.

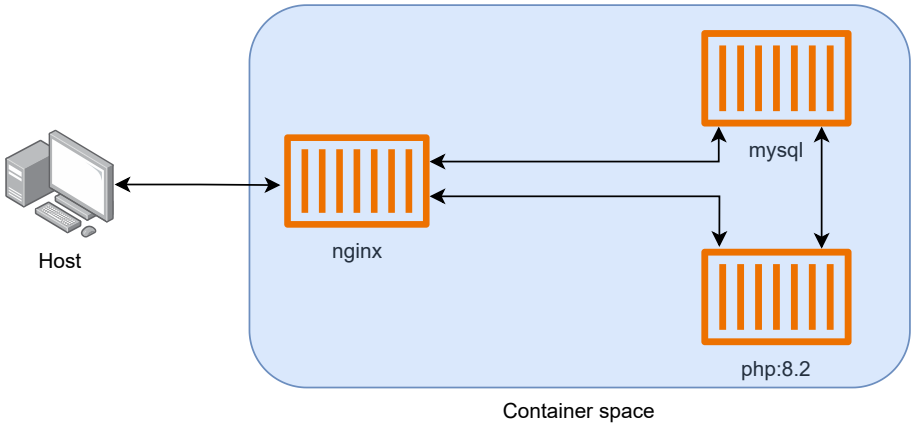

Figure 5.1: Basic website infrastructure

A likely scenario for an attacker is to compromise either the `php:8.2` or the `nginx` containers, as they are the ones processing information coming from the outside. This does include malicious payloads such as SQL injections, Cross-Site Scripting, and so on. The `nginx` container runs with two NICs : one for the container network and another connected to the host.

Only two ports are exposed : Transmission Control Protocol (TCP)/80 and TCP/443.

5.1.2 Basic website with login

This infrastructure is almost the same as the previous one. Instead of allowing users to connect directly to the website, an Identity Provider (IdP) is instantiated. All unauthenticated requests will be redirected to the IdP for authentication and authorization. The IdP runs on Authentik, which already provides containers for its infrastructure. The containers will be included with this infrastructure's containers without any changes. The infrastructure of the website is shown on figure 5.2.

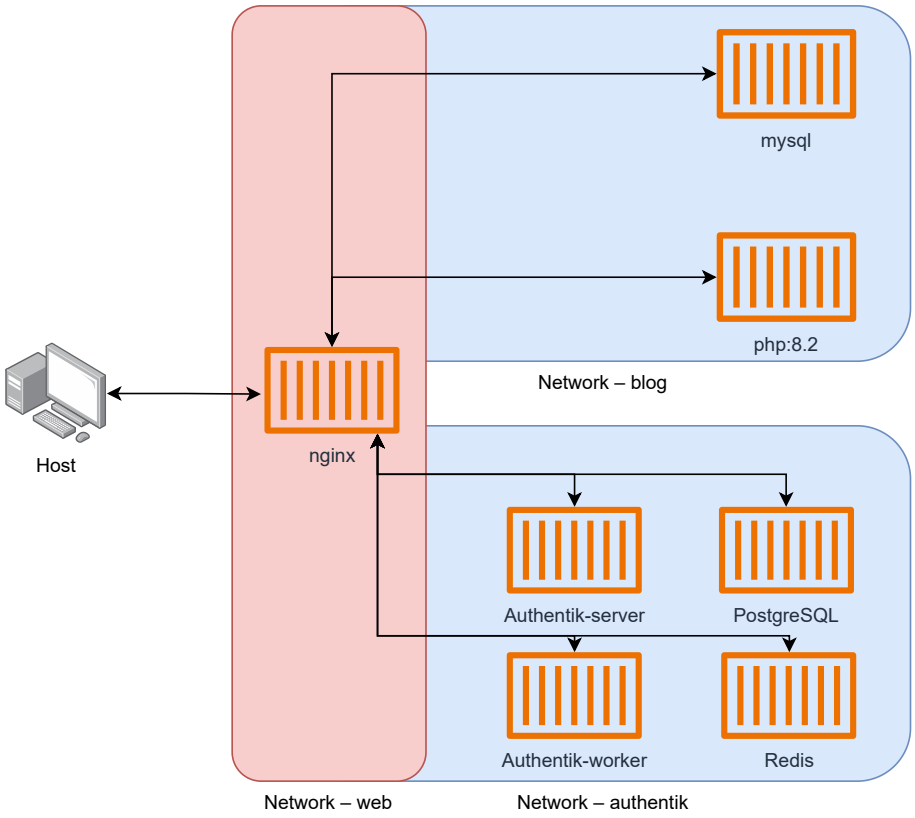

Figure 5.2: Basic website with login

This time, an attacker will have more trouble attacking the web app as it is protected by a login system. Therefore, containers that are likely to be attacked are the `nginx` and the `authentik-server`, both responsible for the login and dispatch of traffic to the IdP and web app. It is interesting to note the position of the `nginx` container, both in the `blog` and the `authentik` network as well as connected to the host. Should this host be compromised, all three networks would be impacted.

5.1.3 Storage monitoring system

In many datacenters, it is paramount to check whether or not the drives responsible for the storage of data are in good health, as all components have variable durability. The *Self-Monitoring, Analysis, and Reporting Technology (SMART)* technology allows system administrators to monitor and replace hard drives before they fail. In many cases, SMART data are not collected, which leaves hard drives unattended and unmonitored until failure. The `scrutiny` container [95] allows system administrators not only to monitor drives, but also to receive alerts on many different channels to ensure proper alert delivery and response. The infrastructure, which consists of this the `scrutiny` container, is shown on figure 5.4.

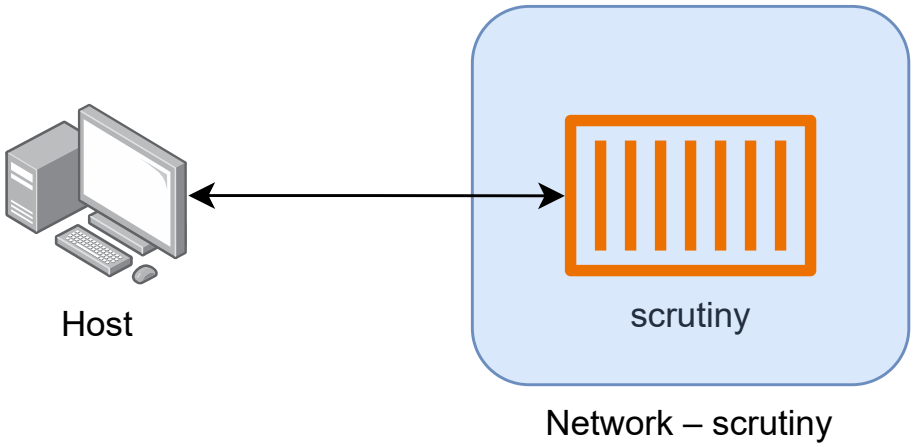

Figure 5.3: Basic monitoring infrastructure

In this case, TCP ports 8080 and 8086 are opened. The container should be able to handle hard drives and Solid State Drives (SSDs) of all sorts.

5.1.4 VPN Infrastructure

This infrastructure is an addition of the first one, as one more container is added to the container network. The container in question is a wireguard server allowing users to connect remotely by a Virtual Private Network (VPN) to the infrastructure. The `nginx` server is therefore unlinked to the host's network as only the VPN requires an outside connection. This does provide an additional layer of security from the perspective of the app, as vulnerabilities linked to the web application may only be tested if the attacker has access to the web app from the VPN. The modified infrastructure is shown on figure 5.3.

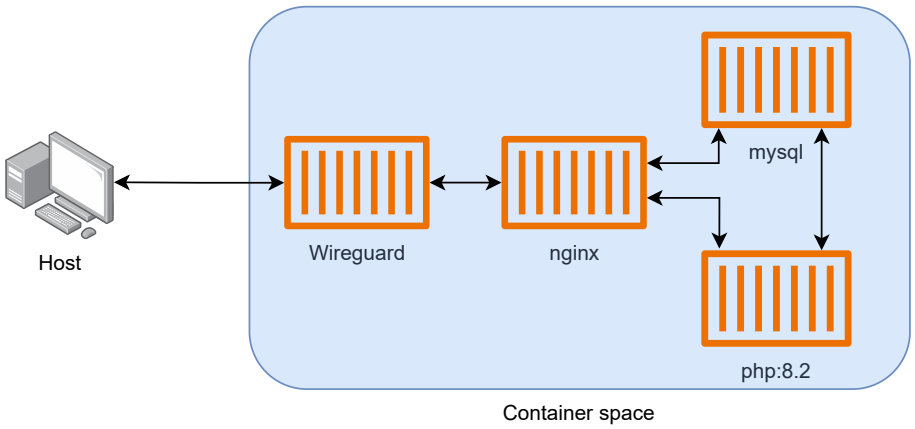

Figure 5.4: Basic website with VPN

Only port User Datagram Protocol (UDP)/12321 and TCP/80 are opened.

An attacker would have no choice but to attack the `wireguard` container as nothing else is exposed.

5.2 Testing protocol

In order to run our tool on several different container engines, Docker compose has been chosen, as it is a very popular container description format, and Podman can also use it.

All infrastructures will be run on the same host with the following hardware :

- Intel(R) Core(TM) i9-10920X CPU @ 3.50GHz (12 cores, 24 threads)
- 32 Go of RAM
- 6 To of storage

To avoid any other service disruption, all infrastructures will be run in a VM, with the following specifications :

- 4 Cores
- 8 Go of RAM
- 32 Go of storage

The VM will run on the host's network, but the containers will remain in their own namespace. To further ensure no side effects can occur, all previously described infrastructures will be ran in turn.

Finally, all containers in a given infrastructure will each run our tool, which will produce an HTML file containing the results of the analysis of the container it ran on, as well as a log file and a score. The HTML file, with the log and the score, will be kept for later analysis. All scores will initially start with the grade of A. Each test will have the ability to return a lower grade or to return no grade at all, which will allow tests that cannot run due to unfulfilled prerequisites to not pollute the overall grade.
The overall grade will reflect the worst grade obtained.

Once all containers in an infrastructure have executed the tool and that viable results have been extracted, the infrastructure is stopped. Once all infrastructures have been tested, the next container engine starts each infrastructure, repeating this testing procedure.

As of this writing, only two container engines are supported :

- Docker. For this tool, our containers will be run as `root`, as rootless containers are still experimental with Docker.
- Podman. For this tool, our containers will be run as the user `labo`, as Podman natively supports rootless containers.

5.3 Results

5.3.1 Basic website

5.3.1.1 Docker

Table 5.1: Results of Tesseract on the `simple-website` Docker infrastructure

name	grade
sw-nginx	A
sw-php	A
sw-mysql	A

The overall grade of this infrastructure is *A*. This is mostly because there is no particular configuration allowing any breakouts : only required files are passed to the container, no extra capabilities are given, no devices are exposed, and so on. Using common parameters to run containers seem to guarantee a certain level of security from our tool's perspective.

5.3.1.2 Podman

Table 5.2: Results of Tesseract on the `simple-website` Podman infrastructure

name	grade
sw-nginx	A
sw-php	A
sw-mysql	A

The overall grade of this infrastructure is *A*. As explained in the Docker test, there is no particular exotic configuration that would allow an attacker to break out of the container. The grade is consistent with the Docker measurement.

5.3.2 Basic website with login

5.3.2.1 Docker

Table 5.3: Results of Tesseract on the `simple-website-login` Docker infrastructure

name	grade
swl-nginx	A
swl-php	A
swl-mysql	A
postgresql	A
redis	A
authentik-server	A
authentik-worker	F

The overall grade of this infrastructure is *F*. This is due to the added complexity of the authentication server, asking the Docker socket to be exposed in the default configuration. While this is optional, it can be quickly disregarded by users as the configuration file is long, complex, and may throw off users unaware of the consequences of doing so. In this case, showing the Docker sock renders the *authentik-worker* container very dangerous, as this container could take control of the host if it was compromised.

5.3.2.2 Podman

Table 5.4: Results of Tesseract on the `simple-website-login` Podman infrastructure

name	grade
swl-nginx	A
swl-php	A
swl-mysql	A
postgresql	A

name	grade
redis	A
authentik-server	A
authentik-worker	D

The overall grade of this infrastructure is *D*. However, after taking a closer look in the logs, our tool was not able to access the `/var/run/docker.sock` file, as all containers are run as a standard user. Once in the container, the user running tesseract (typically `root`) tries to access the `/var/run/docker.sock`, however, the access is denied, as the file comes outside the container and a default setting from Podman restricts the access to the file. The proper overall grade is therefore *C* : should the container suddenly get access to the `docker.sock` file (which should not have been given in the first place, as it is a remnant of the Docker configuration left on purpose for this example), a breakout would surely occur. This grade is not consistent with the last experimentation, as Podman runs without `root` privileges.

5.3.3 Storage monitoring system

5.3.3.1 Docker

Table 5.5: Results of Tesseract on the `scrutiny` Docker infrastructure

name	grade
scrutiny	D

The overall grade of this infrastructure is *D*. This grade is given because the package gives the `SYS_ADMIN` capabilities to the scrutiny container, as the Linux Kernel requires at this time of writing this capability to read SSD drives, especially Non Volatile Memory Express (NVME). This may be used for many different tasks, including mounting the drives and reading their data, but SecComp blocks this from happening. An experienced attacker may cause other problems as the `SYS_ADMIN` capability may cause much damage, for instance “perform various privileged filesystem operations” [71].

5.3.3.2 Podman

Table 5.6: Results of Tesseract on the `scrutiny` Podman infrastructure

name	grade
scrutiny	D

The overall grade of this infrastructure is *D*. As described in the Docker experimentation, our tool finds that the container gets the `SYS_ADMIN` capability, effectively stripping the isolation layer. The grade is consistent with the last experiment.

5.3.4 VPN Infrastructure

5.3.4.1 Docker

Table 5.7: Results of Tesseract on the `vpn` Docker infrastructure

name	grade
vpn-nginx	A
vpn-php	A
vpn-mysql	A
vpn-wireguard	D

The overall grade of this infrastructure is *D*. This is due to the additional capability required by WireGuard. This capability allows an attacker to mount Kernel modules directly on the host, effectively running arbitrary code with total access privileges. As described before, the grade *D* is given when an attack cannot be done automatically, but the grade is very close to an *F*, as the only requirement to breakout is that the host has installed common packages required to compile kernel modules (known as the Linux header packages). Technically speaking, this may be debated to be grade *C*, but the only thing an attacker has to do is to persist a connection to this container and wait for packages to be installed.

Additionally, an attacker able to compromise the `vpn` container of this infrastructure has compromised the infrastructure as a whole, as all incoming and outgoing messages can now be captured and modified on the go.

5.3.4.2 Podman

Table 5.8: Results of Tesseract on the vpn Podman infrastructure

name	grade
vpn-nginx	A
vpn-php	A
vpn-mysql	A
vpn-wireguard	D

The overall result of this infrastructure is *D*. As described in the Docker experimentation, this is because of the additional capabilities given. The attacker simply has to wait for those packages to be installed. All observations made for Docker are valid for this experiment, and results are consistent with the Docker experiment as well.

5.4 Analysis of results

5.4.1 Docker

As in many themes, security can be kept in mind with simple infrastructure, such as the `simple-website` one. However, adding layers of other software onto an infrastructure may compromise the overall security of the containers and the host. For instance, trying to put Wireguard in a container may be considered worse than one directly on the host, as running a container WireGuard is the equivalent of running Wireguard as root, with access to the kernel modules directly from the container.

Docker provides a baseline in regard to its many settings : without proper knowledge, those settings should be left as-is. In fact, many of these settings (exposing devices, volumes, and so on) may lead to a breakout if not understood properly. In the four infrastructures presented above, three were compromised, with one remaining with acceptable levels of isolation.

It is important to remember that containers were taken verbatim from documentation, and that nothing was changed. Containers used for all measurements have been, at this time of writing, downloaded at least 1 million times, with most images downloaded at least 10 million times. Containers running with a risky configuration can be considered common, as most users will use the default configuration provided by the container third-party. Moreover, implemented attacks in our work are trivial and can be considered without doubt to be used by attackers, either manually or with scripts.

Furthermore, as containers are mostly used to deploy web applications [96] [9] and that they are in most cases exposed to the internet, exposing a flawed container directly to the Internet can be done accidentally or not in a matter of seconds. As all experts will agree, exposing a service to the World Wide Web (WWW) will very quickly result in scanners, botnets, and attackers of all sorts to scan the service. It is therefore only a matter of time before the service is compromised, and with it, the host.

As previously mentioned, Docker works by running containers with a *daemon*, hence containers are run as `root`. If an attacker breaks out of the container, `root` access is also obtained. It is estimated that 83% of containers run as `root`[9]. Security organizations comment on this number :

> "[…] Even if risky configurations are detected at runtime, teams do not stop these containers as they do not want to slow their deployment. Instead, they run within a grace period and then decide on the remediation step. Although some containers require this level of privilege to perform their intended function, this number is shockingly high and the trend is going in the wrong direction."

While mature ecosystems such as Cloud Service Providers (CSPs) seem to adopt security tools to ensure the overall security of their platform [9], nothing can be said about smaller enterprises hosting their container infrastructure on-site. For instance, Amazon implemented Firecracker, Google is using gVisor, IBM is using Nabla, and so on.

It is therefore a critical issue to review all Docker containers both before and at runtime, to ensure the permissions granted do not exceed security guidelines or what is actually required in order for the container to work.

5.4.2 Podman

Podman results are generally consistent with Docker : three out of four infrastructures were deemed misconfigured with a significant risk of breakout. However, key differences may have allowed this score to be enhanced even more. For instance, as discussed previously, Podman is a rootless container engine, the opposite of what does Docker. Running containers as a simple user would slow down an attack considerably. The attack surface could be compartmentalized by using different accounts for each container "group" : for instance, containers running a media server would run under the `mediaserver` Linux user.

Therefore, if a breakout occurs, the host would still be considered compromised, but other containers may be still be isolated from the attack.

While Podman is not the most used container engine, given 83% containers are run as `root` [9] it is reasonable to assume some of the Podman containers may be run as `root`.

If transitions from Docker to Podman are made with Podman specificities in mind, Podman may be a

viable solution to enhance the security of containers. Breakouts would still be likely, but would not render the whole host or network compromised at once.

5.4.3 Overall results

First and foremost, it is important to remember that as soon as one container is compromised, the others may be considered compromised as well. The host, on the other hand, may also be impacted partly or completely, mostly depending on the container engine used.

Results show that even default configurations can lead to insecure scenarios. The fact that the results for Docker are close to the ones for Podman suggests that the problem is not with a container tool, it is with container themselves. Only slight changes in the default setting may allow a breakout.

Only trivial attacks were implemented in our work : however, it was still possible to break out of containers even with the default configuration provided by the third-party. Advanced attacks are available for an attacker, such as SPECTRE and MELTDOWN, were not implemented. It is certain that those attacks are tested in the wild as well. With very little effort, our results already show what can be achieved directly from the inside of the container.

Furthermore, our examples were considering there is no orchestrator such as Kubernetes, and all containers were working on a single host. This may not be true in many cases, but to the best of our knowledge, there is no additional security mechanisms in Kubernetes that may prevent a breakout. Orchestrators on multiple hosts may instead be seen as a ramp for lateral movement, and further compromise of the network, as additional services present in orchestrators may alter the attack surface of the infrastructure.

Moreover, as tools warning of such security problems are all opt-in and none are enforced, unless using opt-in container infrastructures such as AWS or Microsoft Azure, it is our belief that there is a generalized risk with containers and their permissions. As most security experts will agree (as well as publications [9]), organizations are success-driven, which encourages Minimal Viable Products (MVPs). In many cases, containers that work but are not fully prepared to meet security expectations are published. This introduces many security problems, including but not limited to secrets leaking [5], excessive rights and capabilities to a container.

It would be foolish to expect processes to change. However, it may be more reasonable to introduce process pipelines for containers, the likes of which exist for software. Pipelines would include peer review, security scans, which could participate in the reduction of intrusion in containers at runtime. As mentioned before, there is a trend towards running containers as `root` : if containers are not migrated to rootless container tools such as Podman, it seems imperative for container administrators to avoid misuse, intrusions and damages.

5.5 In the wild considerations

Attacks in the wild, that is attacks on machines found on the Internet by attackers, may find two categories of infrastructures:

- Private infrastructures, such as personal computers, homelabs, and so on;
- Shared infrastructures, such as CSPs or general datacenters.

In the second case, damages caused by a single breakout may span out of control rather quickly. For instance, a breakout in a CSP datacenter may allow an attacker not only to compromise a full container infrastructure, but also the host, and even in some cases (for example, when no isolation methods have been taken before) impact other tenants running on the same host. Publications [6] suggested that isolation methods between tenants are bare and perhaps insufficient, meaning that a breakout could cause catastrophic damages to both the tenants and the tenant's users. CSP.

Furthermore, many container providers seldom enforce security guidelines. For instance, at the time of this writing, Amazon Web Services (AWS) provides written guidelines to protect containers from specific attacks [97], but any user can choose to ignore those recommendations. AWS also provides a specific service to avoid any co-residency attacks, but this service is opt-in and not as popular as AWS's Elastic Container Service, and is pricier [98].

In our tests, infrastructures were kept at their simplest. In most enterprises, infrastructures can quickly span hundreds or thousands of containers, thus rendering the monitoring of each container quite complex. Furthermore, containers may, by design, disregard or overlook security features for many reasons, accidental or not. In any case, adding containers on top of each other, such as in a microservice architecture, makes it extremely complicated to monitor the network, each container's dependencies, security gaps, and so on. Moreover, containers are very rarely made by the company using them. For instance, as mentioned in our *automated exploration* chapter, third-parties provide containers for many services (web servers, databases, and so on), but it is unclear of their inner workings. This introduces new security challenges.

The many security challenges at hand with containers cannot be ignored without a consequential cost, even in development environments : it is therefore paramount that all actors using containers establish proper security guidelines and apply them. Fortunately, instructions on how to harden a container infrastructure exist and a few will be presented in the next chapter : **mitigation**. This chapter will introduce elements on how to avoid trivial breakouts, thus keeping an acceptable level of security throughout the container infrastructure.

6 Mitigation

6.1 Introduction

So far, our work has presented the many ways an attacker has to break out of a container. In this chapter, we will present techniques to mitigate those attacks, that is to render container attacks inefficient. The goal of mitigation is to reduce the attack surface as much as possible without creating additional misconfigurations, weaknesses or exploits along the way. This generally can be done by removing features from the container, although it is not as simple as changing a few options. For instance, removing capabilities from a container may seem at first like a good idea, but certain apps may depend on them : removing capacities altogether may lead to service disruption.

The second goal of mitigation is therefore to keep the same set of features the container had before the mitigation process without any impact to its features. This does not necessarily mean the container should not be stopped : the second goal is not to ensure High Availability (HA) but to ensure that the container works exactly like before, with little delays considered as acceptable.

This chapter will show existing container guidelines published by security firms or organisms worldwide. The mitigations proposed by our tool should match what the guidelines suggest as both implement container hardening.

We will then shift our focus on proposing a solution fulfilling both of the goals presented below : in some cases, easy fixes can be applied, and in others, it may be more complex. In some others, it may be impossible to harden the container enough for it to be resistant to breakouts. With the latter, other considerations may be of use. This subchapter will then list the different mitigations there exists : automatic mitigation and common mitigation techniques for problems that can be solved easily, long-term mitigation techniques for security issues that may hard to fix, and considerations for containers for unsolvable container issues.

6.2 Existing security guidelines

6.2.1 National Institute of Standards and Technology (NIST) 800-190

The NIST created a special publication especially for containers [99] in publication *800-190*, named *Application Container Security Guide*. This publication, although extensive, lists countermeasures in response to different major container risks. As this publication has security countermeasures well outside our scope, we will only mention the ones relevant to our work.

- *Separated inter-container network traffic* : NIST recommends that apps are separated not by their use, but by their sensitivity level. Separating the networks per-app is also a viable option.
- *Mixing of workload sensitivity levels* : NIST goes one step further by recommending that low sensivity apps should not run on the same host than apps with high sensitivity. This is also valid for different environments : production containers should be kept away from development ones.
- *Vulnerabilities within the container software* : the container runtime should always be kept up-to-date, and monitored for CVEs.
- *Unbounded network access from containers* : containers on two levels of sensitivity should not be able to send or receive network traffic from each other. This introduces many challenges as the networking model in containers is fully software-defined, which may require virtual firewalls, as well as additional considerations outside the supplied Software Defined Network (SDN) options from the container engine.
- *Insecure container runtime configurations* : NIST recommends using tools to detect at-risk configurations such as Docker benchmark [41], based on the Internet Security Docker Benchmark [100]. It is noted that this process does not scale : and that organizations should “use tools or processes that continuously assess configuration settings across the environment and actively enforce them”, without mentioning a specific tool. Additionally, using tools such as AppArmor or SELinux is recommended.

This norm goes far into securing the container host. Implementations of those rules that are correctly applied should slow down a number of determined attackers in most scenarios. In those scenario, it is unlikely that automated tools would work, as they would either find nothing out of the ordinary or would be stopped by monitoring software.

6.2.2 Security Standard 011 - Containerization

The United Kingdom Department for Work & Pensions has deployed a number of standards that they, as well as suppliers or contractors, must apply. SS-011 lists recommendations for containers [101] that go in detail about the management, maintenance and handling of containers. Recommendations like NIST are made, but some are either more underlined or not present in 800-190 :

- *PR.PT-3* mentions that system administrators must “apply the default deny rule to all container capabilities, and only allow those capabilities needed through an explicit ‘Allow List’ ”.
- *PR.DS-6* mentions that “Image registries must support signed images”. Furthermore, it is suggested that container runtimes *must* be configured to enforce running signed images only.
- *DE.CM-1* and *DE.AE-1* also go as far as monitor resources for “unanticipated pikes in resource usage” that could lead to the non-availability of critical resources.

This norm, although less known than NIST's standard, goes further into securing the containers. Mentions of container designs, as well as orchestrators, are taken into account, hence seriously locking down the container host.

The NIST standard was published in september 2017, and SS-011 was published in 2022. Both those standards cover container breakouts at runtime, yet there still are breakouts that regularly happen. Thus, we may reasonably believe that there are several countermeasures not being taken. For instance, the Docker benchmark for Security [41] is a tool that is amongst the most starred repositories on the Docker organization Github, but has yet little popularity compared to the tool. It is very likely that NIST's recommendations are only taken into account by mature organizations, while smaller organizations have a trimmed-down version of it.

6.3 Automatic and semi-automatic mitigation

Automatic mitigation refers to the fact that a mitigation can be applied automatically. In our case, this is represented by a container that is at risk of being patched in a way that the risk is lessened or nulled. Semi-automatic refers to the mitigation being recommended by a machine, but applied by a human.

While our work does not offer automatic mitigation, it can assist an analyst in removing potential container risks that are trivial to solve. For instance, running a container in privileged mode is rarely required : a mitigation would be to remove all capabilities and add them again depending on the vendor’s documentation or logs.

Automatic mitigation is not black magic : at an abstract level, it analyzes “extraordinary” features seen on a container, lists attacks that succeeded, and advises mitigation techniques. Most of them consist of finding an alternative feature that is considered less insecure, or removing the container feature altogether (such as removing the `--privileged` flag when not needed).

While automatic mitigation may be able to solve trivial issues, it may also lead to other problems. For instance, automatic migration does not take into account the complexitity of the software installed in the container. In some cases, it may lead to removing required capabilities and may therefore break the app inside the container. In some others, the automatic mitigation may also introduce flaws in the

container rather than harden it. For instance, the tool may remove sensitive files from the container, but could expose others in the process.

This shows that mitigation of known problems is mostly a manual process and that it can only be done by a person or a tool with proficient knowledge of the container, network and host. Security issues are, as always, easy to create, but hard to solve. Automatic mitigation may be done for very trivial tasks, but should be kept as a verbose option, and not as a tool that automatically applies mitigation techniques. Tools able to mitigate problems automatically do exist [102] [103], but focus solely on the detection and mitigation of a problem in known environments, while our work is designed to be able to work on a container of unknown contents and environment. It can therefore be argued that in our case, automatically applying mitigations without breaking the container is, in itself a work on its own, as it would require understanding the context of each container in great lengths, so that applying mitigations does not break the application inside.

In our case, and as of the time of this writing, it seems preferable to apply mitigations semi-automatically, firstly to properly understand what a mitigation might mean, and secondly to avoid any further service disruption.

6.4 Common mitigation techniques

In this chapter, we define common mitigation techniques as mitigations that can be applied quickly and easily. This does not take into account the complexity of the contained app, but those techniques do not necessarily require extensive configuration updates, host reconfiguration or any process that can take extended amounts of time. This chapter lists several of those techniques which will be developed in this chapter.

6.4.1 Capabilities

In general, additional capabilities are a bad sign. While there are numerous good uses of granting capabilities to a container, many derived uses may lead an attacker to a trivial breakout. It is usually best to consider carefully when adding a capability, as the default ones are sufficient for most uses. When adding a capability, avoid any `CAP_SYS` capability, as they will let an attacker access the host. Additional capabilities do mean adding `root` privileges to a group of processes, and this should never be done without a proper understanding of the consequences. The Linux Kernel documentation provides a good oversight on what capability grants [71].

The best way of getting rid of this security issue is to remove capabilities when not required. For instance, a `wireguard` container requiring `CAP_SYS_MODULE` would be more secure if it was running on a VM, or on an appliance. If moving the container is not possible, ensure the container is not accessible

directly from the outside (that is, any traffic not coming from the container network). Otherwise, apply strict firewall rules to ensure that only legitimate traffic is allowed. As NIST publication 800-190 recommended, isolation of such apps may be required.

6.4.2 Namespaces

All namespaces allow for proper isolation between the host and the container, whether it is volumes, the network, the user IDs, and so on. It is therefore dangerous to remove those isolations as they are the most important security system in the container. Those isolation allow the container to live in its own environment, and removing the namespaces would allow the container to reach the host and vice versa.

For instance, containers using many ports prefer not to bother with the port redirection between the host and the container, and ask the user creating the container to bind the container directly on the host's network. This immediately allows a breakout, as the container could poison key elements of the network such as ARP, and proceed to Man In The Middle (MitM) attacks. As a container has little isolation and is essentially a group of processes, removing the isolation layer is as if a guest was allowed to run a group of processes directly on the host.

Therefore, using options to remove the namespace isolation is generally a terrible idea. Options like `--network=host` are extremely dangerous. Volumes, which are essential for the mount namespace to work, are also considered risky, depending on what is exposed. No filesystem in its entirety or system files should ever be exposed (`/proc`, `/var/run`, `/dev` and so on) as they may allow the container to breakout. For instance `/proc` has files linked to the temperature controller of the kernel. A malicious container could overheat the host, potentially damaging the hardware.

It is therefore a safe practice to avoid tweaking namespaces as much as possible :

- For any networking setting, avoid exposing the host's network and redirect ports instead.
- If volumes are desired for a container, the containers' data should be on a separate partition, away from system files. For instance, storing container files in `/media/docker` where `docker` is a mount on the other partition would be an acceptable practice.
- Others namespaces (User IDs, Cgroups, and so on) should never be tweaked, except for extremely precise reasons.

Luckily, most containers only tweak namespaces with volumes : it is therefore trivial to isolate the container files from the host's. If any networking options are exposing the host's network, an inventory of all ports used by the container must be done, then it is possible to remove the container from the network by redirecting only the required ports from the inventory.

6.4.3 CGroups

While CGroups does not allow a breakout in itself, an attacker could crash other processes by using all the RAM at its disposal, or all the CPU, or storage space. Most containers are run without any cgroups settings, which mean they have unlimited access to the host's resources. Should one container fill the host's resources, containers may start crashing, stop responding or behave in unforeseen ways.

As of this writing, no container configurations were found even suggesting using a particular cgroups profile. Furthermore, documentation about cgroups is sparse, and since many users are content with the vendor's default configuration, it is our belief that cgroups is an option that is rarely used. Although this may not be the case in organizations with proper knowledge (such as multitenant infrastructures like AWS or Microsoft Azure), many smaller organizations may be quick to overlook those settings.

Popular container tools such as Kubernetes or OpenShift do recommend using cgroups, but do not enforce it. As stated before, as most users do use the default configuration, the cgroup settings may be forgotten or ignored. Worse, the first version of cgroups has known vulnerabilities, which may be used to break out of the container.

Our work focuses on escaping the premises of a container. As disregarding cgroups may lead to being able to influence other containers, it is our belief that adding clear limitations on resource consumption for each container is a good practice.

6.4.4 SecComp

Seccomp allows to filter system calls made by a certain process. Container engines do include a SecComp profile for containers [104], but this option can be disabled by using `--security-opt seccomp=unconfined`. This would in turn remove any limitation on system calls. While many system calls would not be available due to the lack of capabilities in a container by default, this would certainly diminish the overall security of the container.

As of this writing, no container configuration did propose an alternative seccomp profile, which suggests the default is adequate for most uses. Customized seccomp files should be avoided as much as possible, as the Linux Kernel is complex and allowing additional system calls may not be an adequate solution in terms of security.

It is therefore advised not to add system calls to the seccomp profile, and rather whitelist only those that are required. Publications have proposed different tools for system calls enumeration in order to generate seccomp profiles automatically [31] [47] [44], which all show optimal results from a security perspective. Other publications suggest that establishing a baseline to enhance anomalies in system calls also proves to be effective [105].

6.5 Long-term mitigation techniques

Long-term mitigation techniques are mitigations that are considered to take a significant amount of time to apply them. This type of mitigation is not the exact opposite of common mitigations as they are not especially rare or common, but are different in the time they take to be applied.

6.5.1 CVE-2022-0492 Carpediem

The CVE CVE-2022-0492 provides a good example of a long-term mitigation technique. This vulnerability impacted one of the core security primitives of containers, cgroups. With this CVE, an attacker in a container could call a file made by cgroups v1, which would in turn be run as `root` on the container host. This CVE provided trivial escape of the container, but the mitigation was recommending cgroups to be upgraded to v2. However, this upgrade required numerous changes on the container host [106], which would have required stopping all container on the host, and in some cases changing the business logic on software running in containers. This may require days or even weeks of work, with disruption to the quality of service.

This CVE is a good example of long-term mitigation techniques : sometimes, the container in itself may not be the only one responsible for the security issue. As soon as other components (such as the host) are involved, it may take a lot of time to properly patch and address issues. In the meantime, security incidents may still occur.

Many examples of security issues that cannot be mitigated quickly can be given. While this work does not claim to offer a complete list, the following subchapters provide examples with potential solutions of security issues with long-term mitigation consequences.

6.5.2 Container engines

Container engines can also be the target of CVEs. In cases where an exploit is found, an upgrade must be done. In most cases, upgrading the container engine will disrupt all containers, as the software running the containers itself will need to be updated and restarted. In the worst scenarios, the update process does not offer proper backward compatibility, which may break containers. For instance, should Docker remove the `--network` option, many containers would stop working. This is also true for peripheral software such as Docker Compose or Podman Compose, created so that a whole container infrastructure can be defined in a single file. If the update process forces the administrator to upgrade to a newer container description format, there is no guarantee that all containers will behave the same after the update.

As keeping a flawed version of a container engine is certainly not a viable option, a user should generally try to patch the host as quickly as possible, while still making sure not to introduce any other security flaw. In some environments, this may take a significant amount of time, in particular with exotic options applied to containers. In most cases, the time required to update from a flawed version to a patched one should be insignificant, as long as updates are kept in order. In other cases, the time required may be exponential, as breaking changes may occur, significantly increasing the time required to properly update the whole container infrastructure.

For instance, CVE-2020-14298 [107] was letting an attacker to bypass security restrictions, effectively accessing the container host. In those cases, if updates were not done properly, patching the CVE may have proven difficult due to the many changes to the container engine. The update process to mitigate CVE-2020-14298 required updating the runc executable, which meant updating the container engine as well, and potentially the host. It is well-known that updates may disrupt running services, and since this mitigation process involves at least two different services, emergency patching would have seriously disrupted all services. Moreover, emergency patching may quickly introduce other problems, including security gaps.

It is therefore recommended to ensure the container engines running all containers are kept up-to-date, as well as the host's kernel and other packages.

6.5.3 Network dependencies

Containers rely not only on the host's hardware, but also on its network. For instance, incoming or outgoing container traffic will require the host's NIC, to properly route the data towards its destination, regardless if the destination is another container or not. The settings of the NIC, as well as the network's settings for the host must be taken into account independently of the host's configuration.

An example of dependency from the container to the host is the host's NIC Maximum Transmission Unit (MTU). As containers have their own network, they can specify any MTU in their network. However, when switching to the host's own NIC, the container engine must convert each data packet from the container's network MTU to the host's MTU. Otherwise, packets will be dropped and connectivity will not work properly. This may happen with protocols typically for datacenters, such as Virtual eXtensible Local-Area Network (VXLAN) or GEneric NEtwork Virtualization Encapsulation (GENEVE) that use part of each packet for network isolation across tenants.

Another dependency might be Virtual LANs (VLANs), as each VLAN will usually have its own role in the network. For instance, the management network, which usually contains all administrative interfaces usually found on servers, routers, switches, firewalls and other equipments. Should a container have access to this VLAN, this might cause the network as a whole to be compromised, as management interfaces are considered highly sensitive and not considered attack-resistant [108].

While network dependencies may not represent much complexity at first, it will impact containers significantly : network security issues may require thorough patches to be applied, and could cause other security incidents. Service disruption may be the least significant factor of importance in those cases : should the wrong VLAN be applied to the host's NIC, this could cause important damages once a container is compromised, as the container will have access to the aforementioned VLAN.

It is therefore recommended to create specific network zones (VLAN, VXLAN, etc.) for containers (depending on the sensitivity), and consider this zone to be already compromised. Setting up zero trust policies on this network zone by authorizing just enough privileges, ports and resources to the network area, which will significantly slow down an attack. In multitenant or multiuser environments, consider adding firewall and intrusion detection software to ensure foreign traffic is flagged, to detect attacks early.

6.5.4 Hardware dependencies

Handling hardware with containers can prove to be complicated but may be required. For instance, containers handling complex matrix computation or converting media files will require hardware acceleration, for which a GPU will perform with optimal performance. However, exposing a hardware device to a container does complexify the interactions between the host and the container. Exposing hardware to a container is different than with a VM : in a VM, a device is fully allocated to the VM, whereas many containers can share devices. This is because containers are simply a group of processes and they share the host's kernel. This means that if any exploit is possible using the exposed devices, the host will be exposed as well.

Hardware dependencies are simple to fix, but only as an emergency measure. If containers do depend on the device to work properly, simply removing it from the container may not be optimal in the long term. Media conversion software will not be able to work as well without GPU at its disposal, and performance will be degraded. In other cases, removing the exposed device is simply not possible, as software may depend on hardware to check for a license or to simply function.

Furthermore, exposing devices to a container may not be sufficient for it to interact with the device, as the process representing the container may not have the proper capability set to do so. In those cases, the container administrator would have no choice but to add the capabilities to ensure the container is working properly. However, this grants the container more permissions than what is recommended, and lessens the isolation between the container and the host. A determined attacker may well be able to break the separation between the two and get root access to the host.

There is little room for maneuver with hardware dependencies as host devices and peripherals simply are exposed or not. In cases where a device needs to be exposed and the device is more than a Universal Serial Bus (USB) device, consider creating a VM to group all containers depending on the device in

one. In some cases, like CVE-2020-3962 [26], described in our State Of The Art, an attacker may still be able to access the VM host, although VMs escapes are known to be rare. It is our recommendation to privilege VMs over container for those use-cases.

6.6 Considerations for containers

Containers are a powerful tool as they allow proper isolation if ran correctly. As soon as exotic configurations are considered, it is paramount to remember the thin layers of isolation in between the host and its containers. To ensure that this isolation remains intact or effective, it is generally a good idea to remain near the common configurations or the default ones.

As containers do tend to abstract many different aspects of system administration, such as networking, security, and maintenance, it is important to remember that container engines are mostly tools made to simplify the lives of their administrators, but that the many complexities are still present. Those abstractions may lead to forgetting key differences between VM and containers, such as the weak level of isolation between containers and the host. In the worst cases, it may lead users to directly run containers regardless of the options passed at runtime, and strip restrictions as they go.

Therefore, the container maintainer must understand all options passed to it at its creation, as one option can quickly render the container dangerous for the whole infrastructure. Mitigation guides [100] [109] do exist, which provides optimal security for containers, and may be a good starting point for container administrators to ensure a good level of isolation. In any case, any option the container maintainer does not know about should be considered a risk and documented, understood and avoided if possible.

Default configuration is also a security risk in itself, depending on who is providing said configuration. For instance, the container publisher may have limited experience or knowledge about containers, which may lead the publisher to provide at-risk configurations. In some cases, publishers have been known to ask for wide permissions to be granted, to avoid long and complex manuals [110]. This is the same with containers, as some publishers have been known to strip the isolation layer much more than required. An example of such a container, downloaded more than 500 million times, is shown on figure 6.1.

The abstraction of topics linked to containers coupled with an “App Store” for containers, such as the Docker Hub, has tremendously simplified software developments, deploying new projects, and software infrastructure. However, it has also created new security issues, such as the ones described in this work. Nowadays, container users should not rely on the container or its default configuration without proper understanding of what the container is working with, that is its capabilities, its volumes, its networking, and so on.

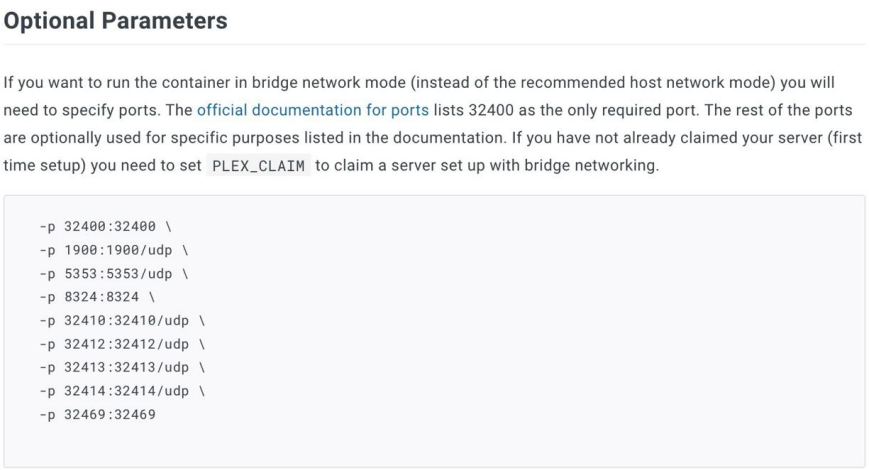

Figure 6.1: A popular container removing the much-needed layer of isolation between the host and the container [111]

As container issues can lead to security ones, and may consequently lead to compromising the entire host or cluster, containers, while disposable, are still to be considered as a process that can receive or emit malicious traffic. While designing security for containers may take significant amounts of time, patching containers for security issues once they are deployed may take much more time to patch than to design [112]. While emergency patching is almost inevitable, a container administrator may save time by checking containers for weaknesses before running them, by comparing the containers with security guidelines and security tools suggested in this work.

Should all of the above solutions be inadequate, then it may be best to consider VMs instead of containers. Switching from containers to VMs is switching from one paradigm to another, but it drastically reduces the chances of a breakout. When VMs are not possible, for example in Kubernetes or OpenShift clusters, the use of Kata containers may be of use, as containers are replaced by VMs, ensuring an optimal security layer between the VM and the host. Other solutions are available depending on the requirements. It is important to denote that implementing a tool fitting security needs may slow down the container [113], but would certainly cost less than having to patch security holes in an emergency.

In fact, more generally, there surely is a solution with containers that will fulfill each need. The most important point is to ensure a consistent and optimal level of security to avoid breakouts, for example by using VMs or by using containers with hardened configurations. As containers can be started in a matter of seconds, scalability is most probably the biggest issue with this technology, as containers are seen as disposable. More than 70% run for less than 5 minutes [9], and since 61% of organizations run more than 250 containers at the same time, it is very hard to keep track of all containers. This also explains why containers are targeted by attackers and why breakouts are favoured over many other entry points : given the constant new containers ran, it is only a matter of time before one vulnerable

container appears.

In our opinion, in view of the above, it is truly critical for any organization using containers to use the security tools at their disposal, establish a well-established security policy as well as deployment process, and retroactively review current containers. A real security risk is present and cannot be ignored without considerable subsequent losses. As container technology is still very much on the up, ignoring its security could lead to heavy losses for both operators and container users.

7 Conclusion

Security is a primary concern for any organization, as a lack of it will certainly lead to damage to both the organization and its users. Containers have proven useful to separate concerns, services and entire departments without the overhead of Virtual Machines (VMs). Their usage have tremendously increased in recent years, to the point that many programs only come pre-packaged for containers. The main reason for their popularity is their simplicity, as one command can launch a full infrastructure in a matter of seconds. As turnkey solutions are popular within organizations, this pushed containers to the top of preferred technologies [42] within the developer community. However, the main downside is that users are less aware of how containers work and their inherent risks. Good practices have not been set universally, which may lead developers, administrators and other container users to disregard or ignore security risks linked to containers, let alone their inner workings. Organizations such as the National Institute of Standards and Technology (NIST) quickly underlined the importance of good practices and the most frequent risks. Furthermore, containers are still gaining popularity, and so are security tools, either created to harden containers or to break out of them.

While big organizations and Cloud Service Providers (CSPs) have quickly adopted an optimal security posture, data suggest smaller organizations as well as environments for development or ones that are at-risk such as Internet of Things (IoT) devices are, in many cases, left exposed.
This work proposes a tool to analyze containers and underline security risks linked to containers from the inside. Our work explores the security aspects of containers and also studies their security implications.

In our introduction chapter, we have asked four questions. We have answered the four as follows :

- *Can containers be fully scanned to determine which attack is feasible ?* We have established in chapter 4 that, as long as a binary can be executed with some form of privilege, it is possible to determine the container engine, the potential weaknesses of the container and the attacks that could breakout of the container, all while inside of it. We have shown in chapter 5 that standard configurations found on the web are sufficient to allow trivial attacks to pass, regardless of the container tool used.
- *What are the risks with the current use of containers ?* We have proved in chapter 3, 4 and 5 that containers are not secure due to their lack of isolation, and we have also proved that breakouts can easily occur with few requirements. We have then showed in chapter 5 and 6 the risks of

using containers carelessly or with little understanding of them, which could cause extensive damage. We hence have proved that there are significant risks of using containers without a proper understanding of them and how they run.

- *What are the challenges in mitigating attacks with containers ?* We have answered this question in chapter 6 by analyzing the cost of mitigation after deployment, and highlighted the cost of emergency patching in comparison to early precautions. We also have listed potential roadblocks that could slow down mitigation of containers, and have suggested solutions for each of them.
- *What can be done to avoid breakouts on containers ?* We also have answered this question in chapter 6, by underlining existing guidelines that would not only allow to minimize the risk of breakouts. Both guidelines went a step further by ensuring that incursions cannot compromise the infrastructure as a whole. We have also suggested using several tools to harden containers as if they were VMs, for example by using Kata containers.

Changing any process is a tenuous task, as most humans quickly get accustomed to how they work. Containers are no exception : from what was seen, an enormous challenge is ahead for many organizations. Changing the ways of working for all users, including junior employees, will surely represent a challenge. However, administrators in charge of infrastructure may be able to impose certain guidelines, the lines of which should follow well-established ones, such as NIST and so on.

Contrary to new technologies, there is enough material with containers to mitigate common and uncommon attacks in a way that satisfies both the container maintainers and the security guidelines of an organization. While it is true it is impossible to account for future attacks, most current attacks can be thwarted with simple fixes.

Ensuring a container is secure is a known process as it is similar to traditional hardening methods, such as VM. A machine, regardless if it is virtual that has excessive rights (such as exposed unused ports or wide access to company resources) or not, can be patched efficiently so that the service has exactly enough rights to function. As containers are groups of processes, they fall into the same category and hence can be patched in the same way.

This work concludes that, without proper security mechanisms in place, whether it is at the organizational level with processes or at a technological level with a security baseline, containers will continue to be vulnerable. As attackers have adopted automated scans, it is only a matter of time, if it is not already the case, that attackers will adopt automated attack tools designed to break out of containers and many times more powerful than our implementation.

Although the situation is critical, this kind of security hole is unfortunately common with new technologies. It is urgent to implement a policy aimed at hardening containers rather than keeping them open, at the risk of an initial drop in productivity. This should be temporary, however, and gradually be adopted by all, leading to a more secure container environment. Responsible container mainte-

nance right from the service development stage will minimize the risks of intrusion and unauthorized access.

7.1 Future works

The basis of our tool relies on its attack implementations. As containers have many interactions with the container host, there are many attacks that can still be implemented to assess the security of a container, from trivial ones to more complex ones, relying on the Central Processing Unit (CPU) architecture, or more generally using a specific version of the container runtime. As containers are getting very popular, many organizations and individuals are searching for vulnerabilities, either due to misconfiguration or mistakes in the container engine. Those vulnerabilities could be included in our work easily, as dependencies can be included in our tool's binary and the context engine allows for proper checks of the environment. Both can be extended at will, allowing new attacks to be added. Furthermore, instead of developing our attacks, attacks found on community websites dedicated for exploits and vulnerabilities, such as *The Exploit Database* would further allow our tool to have a wide set of attacks.

As security organizations are also focusing on containers, ensuring that attacks can go as undetected as possible would allow our tool to run even in high-security cases. The feasibility of such a feature could be observed by checking if our tool is detected as an anomaly by security tools such as Falco.

We believe more and more automated attacks will occur, as attackers are quickly adapting to new paradigms. It is also our belief that optimal security practices will continue to be pushed forward by major technological investors like Microsoft, Amazon and Google as they depend on security infrastructures, as they depend on it. As new technologies for containers appear very frequently, it is very probable new security procedures for containers will also become common with time, even in small organizations, at the cost of some paying the toll for not implementing them properly.

8 References

[1] F. Ponce, G. Márquez, and H. Astudillo, “Migrating from monolithic architecture to microservices: A Rapid Review,” in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, Nov. 2019, pp. 1–7. doi: 10.1109/SCCC49216.2019.8966423

[2] “What is a single point of failure (SPOF) and how to avoid them?” *Data Center*. https://www.techtarget.com/searchdatacenter/definition/Single-point-of-failure-SPOF.

[3] “What Is a Sandbox Environment? Meaning & Setup | Proofpoint US,” *Proofpoint*. https://www.proofpoint.com/us/threat-reference/sandbox, Feb. 2021.

[4] “What is an Attack Surface? | IBM.” https://www.ibm.com/topics/attack-surface.

[5] M. Dahlmanns, C. Sander, R. Decker, and K. Wehrle, “Secrets Revealed in Container Images: An Internet-wide Study on Occurrence and Impact,” in *Proceedings of the ACM Asia Conference on Computer and Communications Security*, Jul. 2023, pp. 797–811. doi: 10.1145/3579856.3590329. Available: https://arxiv.org/abs/2307.03958

[6] S. Shringarputale, P. McDaniel, K. Butler, and T. La Porta, “Co-residency Attacks on Containers are Real,” in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, in CCSW’20. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 53–66. doi: 10.1145/3411495.3421357

[7] G. Liu, X. Gao, H. Wang, and K. Sun, “Exploring the Unchartered Space of Container Registry Typosquatting,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 35–51.

[8] Kasey Panetta, “Gartner Top 10 Security Projects For 2019,” *Gartner*. https://www.gartner.com/smarterwithgartner/gartner-top-10-security-projects-for-2019.

[9] SysDig, “Sysdig 2023 Cloud-Native Security and Usage Report.”

[10] Florent Glück, “Flg_courses / virtualization / virtualization_pub_jour · GitLab,” *GitLab*. https://githepia.hesge.ch/flg_courses/virtualization/virtualization_pub_jour, Apr. 2023.

[11] S. Conroy, “History of Virtualization,” *I Don’t Know, Read The Manual*. https://www.idkrtm.com/history-of-virtualization/, Jan. 2018.

[12] R. J. Creasy, “The Origin of the VM/370 Time-Sharing System,” *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, Sep. 1981, doi: 10.1147/rd.255.0483

[13] R. Dillet, “Shadow revamps its cloud computing offerings in Europe,” *TechCrunch*. Jun. 2023.

[14] S. Simic, “Containers vs Virtual Machines (VMs): Critical Differences to Understand,” *Knowledge Base by phoenixNAP*. https://phoenixnap.com/kb/containers-vs-vms, Apr. 2019.

[15] N. Rin, “Virtual Machines Detection Enhanced.”

[16] “Virtualization and Protection Rings (Welcome to Ring -1) Part I,” *Zeros & Ones*. May 2009.

[17] “Intel® Virtualization Technology for Connectivity.” https://www.intel.com/content/dam/www/public/us/en/doc
briefs/virtualization-technology-connectivity-brief.pdf.

[18] diagprov, “Answer to "What is protection ring -1?"," *Information Security Stack Exchange*. Jul. 2016.

[19] J. Reid and W. Caelli, “DRM, Trusted Computing and Operating System Architecture,” *Conferences in Research and Practice in Information Technology Series*, vol. 44, Jan. 2005, doi: 10.5555/1082290.1082308

[20] “Virtualization and Ring Negative One,” *Coding Horror*. https://blog.codinghorror.com/virtualization-and-ring-negative-one/, May 2006.

[21] N. M, M. H, and K. T, “Virtual Machines Detection Methods Using IP Timestamps Pattern Characteristic,” *International Journal of Computer Science and Information Technology*, vol. 8, no. 1, pp. 1–15, Feb. 2016, doi: 10.5121/ijcsit.2016.8101

[22] “Zero Day Initiative — CVE-2022-26937: Microsoft Windows Network File System NLM Portmap Stack Buffer Overflow,” *Zero Day Initiative*. https://www.thezdi.com/blog/2022/6/7/cve-2022-26937-microsoft-windows-network-file-system-nlm-portmap-stack-buffer-overflow.

[23] “CVE - CVE-2008-4395.” https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2008-4395.

[24] A. A. Ali, “Virtual machine escapes,” Apr. 2013.

[25] “CWE - CWE-416: Use After Free (4.12).” https://cwe.mitre.org/data/definitions/416.html.

[26] “CVE-2020-3962.” https://www.cve.org/CVERecord?id=CVE-2020-3962.

[27] Linus Torvalds, “History for kernel/cgroup.c - torvalds/linux.” https://github.com/torvalds/linux/commit/ddbcc7e8e50aefe467c01cac3dec71f118cd8ac2.

[28] D. Jones, “Containers vs. Virtual Machines (VMs): What’s the Difference? | NetApp Blog.” https://www.netapp.com/blog/containers-vs-vms/, Mar. 2018.

[29] “Monolithique et microservices : différence entre les architectures de développement logiciel-AWS,” *Amazon Web Services, Inc.* https://aws.amazon.com/fr/compare/the-difference-between-monolithic-and-microservices-architecture/.

[30] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, “A Measurement Study on Linux Container Security: Attacks and Countermeasures,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, in ACSAC ’18. New York, NY, USA: Association for Computing Machinery, Dec. 2018, pp. 418–429. doi: 10.1145/3274694.3274720

[31] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, and Q. Li, “SPEAKER: Split-Phase Execution of Application Containers,” doi: 10.1007/978-3-319-60876-1_11

[32] Brandon Edwards and Nick Freeman, “A Compendium of Container Escapes.” 2019.

[33] gmcdouga, “Docker Images: Why are Many Cyber Attacks Originating Here?” *Check Point Blog*. https://blog.checkpoint.com/securing-the-cloud/docker-images-why-are-many-cyber-attacks-originating-here/, Jul. 2023.

[34] Florent Faisendel, “Honey-pot de haute interactivité SSH.” https://gradechelor.hesge.ch/api/studentFiles/2228/ISC_SE_memoire_diplome_Faisandel_Hoerdt_2023.pdf, 2023.

[35] “Docker Hub Container Image Library | App Containerization.” https://hub.docker.com/.

[36] “A hacking group is hijacking Docker systems with exposed API endpoints,” *ZDNET*. https://www.zdnet.com/article/a-hacking-group-is-hijacking-docker-systems-with-exposed-api-endpoints/.

[37] S. Sultan, I. Ahmad, and T. Dimitriou, “Container Security: Issues, Challenges, and the Road Ahead,” *IEEE Access*, vol. 7, pp. 52976–52996, 2019, doi: 10.1109/ACCESS.2019.2911732

[38] “GitHub - PercussiveElbow/docker-escape-tool: Tool to test if you’re in a Docker container and attempt simple breakouts,” *GitHub*. https://github.com/PercussiveElbow/docker-escape-tool.

[39] “CDK - Zero Dependency Container Penetration Toolkit.” cdk-team, Jul. 2023.

[40] “Amicontained.” https://github.com/genuinetools/amicontained, Jul. 2023.

[41] “Docker Bench for Security.” Docker, Dec. 2023.

[42] “Stack Overflow Developer Survey 2022,” *Stack Overflow*. https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022.

[43] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, “ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds,” in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2017, pp. 237–248. doi: 10.1109/DSN.2017.49

[44] C. Canella, M. Werner, D. Gruss, and M. Schwarz, “Automating Seccomp Filter Generation for Linux Applications.” arXiv, Dec. 2020. doi: 10.48550/arXiv.2012.02554. Available: https://arxiv.org/abs/2012.02554

[45] “Seccomp security profiles for Docker,” *Docker Documentation*. https://docs.docker.com/engine/security/seccomp/, Aug. 2023.

[46] Z. Guo, Z. Lv, N. Li, T. Yuan, X. Gao, and Z. Yuan, “Comprehensive defense scheme against container escape related to container management procedure,” in *2022 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, Oct. 2022, pp. 263–266. doi: 10.1109/CyberC55534.2022.00051

[47] N. Lopes, R. Martins, M. E. Correia, S. Serrano, and F. Nunes, “Container Hardening Through Automated Seccomp Profiling,” in *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*, in WOC’20. New York, NY, USA: Association for Computing Machinery, Jan. 2021, pp. 31–36. doi: 10.1145/3429885.3429966

[48] “"Distroless" Container Images.” https://github.com/GoogleContainerTools/distroless, Sep. 2023.

[49] “Why distroless containers aren’t the security solution you think they are.” https://www.redhat.com/en/blog/why-distroless-containers-arent-security-solution-you-think-they-are.

[50] “Gestion des vulnérabilités des conteneurs et sécurité des charges de travail Kubernetes,” *Snyk*. https://snyk.io/fr/product/container-vulnerability-management/.

[51] “Aquasecurity/trivy.” Aqua Security, Jan. 2024.

[52] “Anchore/grype.” Anchore, Inc., Jan. 2024.

[53] K. German and O. Ponomareva, “An Overview of Container Security in a Kubernetes Cluster,” in *2023 IEEE Ural-Siberian Conference on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT)*, Yekaterinburg, Russian Federation: IEEE, May 2023, pp. 283–285. doi: 10.1109/USBEREIT58508.2023.10158865

[54] C.-W. Tien, T.-Y. Huang, C.-W. Tien, T.-C. Huang, and S.-Y. Kuo, “KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches,” *Engineering Reports*, vol. 1, no. 5, p. e12080, 2019, doi: 10.1002/eng2.12080

[55] S. Arnautov *et al.*, “SCONE: Secure Linux Containers with Intel SGX,” doi: 10.5555/3026877.3026930

[56] W. He, K. Yun, L. Bai, and K. Li, “Research on Defense Strategy of Container Image-Attack Based on Bayesian Game,” in *2021 IEEE 6th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, Apr. 2021, pp. 432–436. doi: 10.1109/ICCCBDA51879.2021.9442517

[57] Y. C. Jadhav, A. Sable, M. Suresh, and M. K. Hanawal, “Securing Containers: Honeypots for Analysing Container Attacks,” in *2023 15th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, Jan. 2023, pp. 225–227. doi: 10.1109/COMSNETS56262.2023.10041276

[58] Filip Borkiewicz, “Piping curl to s(hell).” https://0x46.net/thoughts/2019/04/27/piping-curl-to-shell/.

[59] “Ohmyzsh/ohmyzsh.” Oh My Zsh, Jan. 2024.

[60] “Getting started.” https://www.rust-lang.org/learn/get-started.

[61] “Docker/docker-install.” Docker, Jan. 2024.

[62] “What Is container orchestration,” *Google Cloud*. https://cloud.google.com/discover/what-is-container-orchestration.

[63] David A Rusling, “Chapter 4 - processes.” https://tldp.org/LDP/tlk/kernel/processes.html.

[64] John Saraille, “Processes - 10th ed. Chapter 03.” https://www.cs.csustan.edu/~john/Classes/CS3750/Notes/Chap

[65] “Cgroups(7) - Linux manual page.” https://man7.org/linux/man-pages/man7/cgroups.7.html.

[66] “Runtime options with Memory, CPUs, and GPUs,” *Docker Documentation*. https://docs.docker.com/config/containers/resource_constraints/, 09:42:25 +0200 +0200.

[67] “Syscalls(2) - Linux manual page.” https://man7.org/linux/man-pages/man2/syscalls.2.html.

[68] “Basics of Seccomp for Docker,” *tbhaxor*. https://tbhaxor.com/basics-of-seccomp-for-dockers/, Jun. 2022.

[69] “Network_namespaces(7) - Linux manual page.” https://man7.org/linux/man-pages/man7/network_namespaces.7.html.

[70] “Mount_namespaces(7) - Linux manual page.” https://man7.org/linux/man-pages/man7/mount_namespaces.7.html.

[71] “Capabilities(7) - Linux manual page.” https://man7.org/linux/man-pages/man7/capabilities.7.html.

[72] “Moby/oci/caps/defaults.go at master · moby/moby,” *GitHub*. https://github.com/moby/moby/blob/master/oci/caps/defaults.go.

[73] “Common/docs/containers.conf.5.md at main · containers/common,” *GitHub*. https://github.com/containers/common/blob/main/docs/containers.conf.5.md.

[74] F. Loukidis-Andreou, I. Giannakopoulos, K. Doka, and N. Koziris, “Docker-Sec: A Fully Automated Container Security Enhancement Mechanism,” in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, Jul. 2018, pp. 1561–1564. doi: 10.1109/ICDCS.2018.00169

[75] P. Loscocco and S. Smalley, “Integrating Flexible Support for Security Policies into the Linux Operating System,” in *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, 2001. doi: 10.5555/647054.715771

[76] *Introduction to SELinux - Enterprise Cloud Security and Governance [Book]*.

[77] D. Pahuja, A. Tang, and K. Tsoutsman, “Automated SELinux RBAC Policy Verification Using SMT.” arXiv, Dec. 2023. doi: 10.48550/arXiv.2312.04586. Available: https://arxiv.org/abs/2312.04586

[78] “Stack Overflow Developer Survey 2023,” *Stack Overflow*. https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023.

[79] “Podman vs. Docker – secure containerization,” *EuroLinux*. https://en.euro-linux.com/blog/podman-vs-docker-secure-containerization/, Jan. 2021.

[80] “Docker Compose Specification.” https://compose-spec.io/.

[81] J. Chen, “Making Containers More Isolated: An Overview of Sandboxed Container Technologies,” *Unit 42*. Jun. 2019.

[82] O. Flauzac, F. Mauhourat, and F. Nolot, “A review of native container security for running applications,” *Procedia Computer Science*, vol. 175, pp. 157–164, Jan. 2020, doi: 10.1016/j.procs.2020.07.025

[83] “Kata Containers - Open Source Container Runtime Software.” https://katacontainers.io/.

[84] “Opencontainers/runc.” Open Container Initiative, Jan. 2024.

[85] D. Walsh, “An introduction to crun, a fast and low-memory footprint container runtime.” https://www.redhat.com/sysadmin/introduction-crun; Red Hat, Inc., Aug. 2020.

[86] “Lxc/lxc.” LXC - Linux Containers, Jan. 2024.

[87] "How nodes work," *Docker Documentation*. https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/, 09:15:29 +0000 UTC.

[88] "Service Kubernetes géré – Amazon EKS – Service web Amazon," *Amazon Web Services, Inc.* https://aws.amazon.com/fr/eks/.

[89] "Kubernetes @ CERN." https://kubernetes.web.cern.ch/.

[90] C. le roy, "Brompwnie/botb." Jan. 2024.

[91] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson, "Automatic exploration of datacenter performance regimes," in *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, in ACDC '09. New York, NY, USA: Association for Computing Machinery, Jun. 2009, pp. 1–6. doi: 10.1145/1555271.1555273

[92] N. Yang, C. Chen, T. Yuan, Y. Wang, X. Gu, and D. Yang, "Security hardening solution for docker container," in *2022 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, Oct. 2022, pp. 252–257. doi: 10.1109/CyberC55534.2022.00049

[93] "Moby/daemon/execdriver/lxc/init.go at v1.9.1 · moby/moby," *GitHub*. https://github.com/moby/moby/blob/v1.9.1/daemon/execdriver/lxc/init.go.

[94] "Common Vulnerability Scoring System SIG," *FIRST — Forum of Incident Response and Security Teams*. https://www.first.org/cvss.

[95] "AnalogJ/scrutiny: Hard Drive S.M.A.R.T Monitoring, Historical Trends & Real World Failure Thresholds." https://github.com/AnalogJ/scrutiny.

[96] C. Boettiger, "An introduction to Docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, Jan. 2015, doi: 10.1145/2723872.2723882

[97] "Task and container security - Amazon Elastic Container Service." https://docs.aws.amazon.com/AmazonECS/latest/bestpracticesguide/security-tasks-containers.html.

[98] "My Estimate - Calculateur de prix AWS." https://calculator.aws/#/estimate.

[99] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide." National Institute of Standards and Technology, Gaithersburg, MD, Sep. 2017. doi: 10.6028/NIST.SP.800-190

[100] "CIS Docker Benchmarks," *CIS*. https://www.cisecurity.org/benchmark/docker.

[101] Department for Work and Pensions, "Security Standard – Containerisation (SS-011)." 2022.

[102] E. Filiol, F. Mercaldo, and A. Santone, "A Method for Automatic Penetration Testing and Mitigation: A Red Hat Approach," *Procedia Computer Science*, vol. 192, pp. 2039–2046, Jan. 2021, doi: 10.1016/j.procs.2021.08.210

[103] "PHILOS: Real-time Detection and Automated Mitigation of BGP Prefix Hijacking Attacks | PHILOS Project | Fact Sheet | H2020," *CORDIS | European Commission*. https://cordis.europa.eu/project/id/790575.

[104] "Moby/profiles/seccomp/default.json at master · moby/moby," *GitHub*. https://github.com/moby/moby/blob/master/profiles/seccomp/default.json.

[105] A. El Khairi, M. Caselli, C. Knierim, A. Peter, and A. Continella, "Contextualizing System Calls in Containers for Anomaly-Based Intrusion Detection," in *Proceedings of the 2022 on Cloud Computing Security Workshop*, in CCSW'22. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 9–21. doi: 10.1145/3560810.3564266

[106] "Embracing Cgroup V2: Best Practices for Migrating Kubernetes Clusters to AlmaLinux," *CNCF*. https://www.cncf.io/blog/2023/06/30/embracing-cgroup-v2-best-practices-for-migrating-kubernetes-clusters-to-almalinux/, Jun. 2023.

[107] "CVE-2020-14298 : The version of docker as released for Red Hat Enterprise Linux 7 Extras via RHBA-2020:0053 advisory included an incorrec." https://www.cvedetails.com/cve/CVE-2020-14298/.

[108] A. Bonkoski, R. Bielawski, and J. A. Halderman, "Illuminating the Security Issues Surrounding Lights-Out Server Management," doi: 10.5555/2534748.2534761

[109] J. Shetty, "A State-of-Art Review of Docker Container Security Issues and Solutions," *American International Journal of Research in Science, Technology, Engineering & Mathematics*, Jan. 2017.

[110] "Nintendo Support: How to Set Up a Router's Port Forwarding for a Nintendo Switch Console." https://en-americas-support.nintendo.com/app/answers/detail/a_id/22272/~/how-to-set-up-a-routers-port-forwarding-for-a-nintendo-switch-console.

[111] "Linuxserver/plex - Docker Image | Docker Hub." https://hub.docker.com/r/linuxserver/plex.

[112] Bromium, "The Hidden Costs of Detect-to-Protect Security."

[113] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The True Cost of Containing: A gVisor Case Study."