POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Cybersecurity assessment and host hardening policies for virtualization technologies

Supervisors:

Prof. Cataldo Basile

Ing. Rossella Lertora

Candidate:

Davide Lo Bianco

Academic Year 2021/2022
Torino

Abstract

In my thesis I selected and analyzed the virtualization technologies that are most widely utilized and created a set of host hardening policies for a Docker environment. The starting point was the research of the current state of the art of virtualization technologies that can were divided in three main categories, based on their implementation: type 1 hypervisors, type 2 hypervisors and OS virtualization. Based on the diffusion of the technologies in the automotive development and testing world we decided to analyze further two products, one classified as type 1 hypervisor since is the most common technology found in automotive embedded systems and one OS virtualization product that is very flexible and suitable for testing. This analysis was performed by studying the software in its technical details and core elements to obtain an accurate overview on how the virtualization mechanism is implemented.The products that were chosen were Xen Project Hypervisor and Docker representing type 1 hypervisors and OS virtualization respectively. These two software were then analyzed from the security point of view by performing a small scale risk assessment based on the 800-30 Nist document. Thanks to the assessment and publicly available technical documents a set of possible vulnerabilities were highlighted in both technologies, this led to the identification of some possible countermeasures to said vulnerabilities as general host hardening policies.
Due to testing and hardware availability reasons Docker was the software chosen for practical examination and based on the general host hardening policies defined in the preceding step a set of specific host hardening rules were defined to be enforced in order to increase the security of the host system. The last step was the automatization of the verification process of said rules by developing a tool. The tool was developed in the Python language and its purpose is to create a readable report containing all the informations that

can be collected from the system and present them to the user in order to highlight any possible misconfiguration or selected configuration option that could lead to security risks and at the same time to offer a possible solution to the highlighted problems.

Acknowledgements

I would like to thank the following people without whom I would not have been able to complete this thesis. First and foremost I would like to thank my supervisor professor Basile C. for his availability and patience during this long period of research and his support when problems arose. I would like to thank Drivesec and its employees Lertora R. and Rocca A. for the technical support shown in these months that helped me complete my thesis by following their directions. I would also like to thank my family for the support and patience shown in these years of attending the Politecnico di Torino, most importantly during the covid years. Last but not least I would like to thank my girlfriend, the driving force behind this achievement and without whom none of this would have been possible.

Table of Contents

List of Figures

Acronyms

VM	Virtual Machine
OS	Operating System
HW	Hardware
API	Application Programming Interface
CLI	Command Line Interface
PID	Process Identifier
IPC	Inter Process Communication
DNS	Domain Name System
HVM	Hardware Virtual Machine

PVH

Para virtualization using HVM extensions

vCPU

Virtual CPU

MMU

Memory Management Unit

DMA

Direct Memory Access

I/O

Input Output

TTP

Trusted Third Party

MLE

Measured Launch Environment

COTS

Commercial Off The Shelf

TLS

Transport Layer Security

ADT

Abstract Data Type

UTS

UNIX Time Sharing

Chapter 1

Introduction

Nowadays, virtualization is a key component in many IT fields, and its utilization is essential to improve the performance and security of many systems, this is achieved by isolating, managing access to system resources, assigning privileges and performing access control to the created virtual environments. Running virtualized environments can as said before, on one hand, improve the security level of a system, but on the other hand, it adds a new attack surface that a malicious user could exploit. This explains why there is a need to improve the security of virtualization technologies by identifying the right process to recognize and fix configuration vulnerabilities.

Virtualization products usually present a set of default configuration options enabled to ensure the correct execution of the software. These configurations are often focused more on guaranteeing general-purpose functionalities to the system than devoted to implement a security-oriented solution, this means that in most cases is up to the user to change the setting of the system to enforce security. While sets of security rules are available online, their availability is not expanded for every product. Therefore emerges the need to identify the right process to detect vulnerabilities and select the related security policies.

This thesis proposes a methodical solution for detecting configuration vulnerabilities in virtualization products and the formulation of consequent host

hardening policies. The identified process starts by analyzing the most common virtualization paradigms to classify commercial virtualization products; each product can then be studied in depth to understand its architecture, workflow, and main strategies for implementing virtualization. Based on the software structure of the product a series of assets can be selected as possible sources of risks for the system, that can be targeted by threat sources, causing a set of threat events that could affect the virtualization environment. These threat events can be used to identify a series of possible vulnerabilities present in the implementation of the virtualization software solution due to architectural defects or not security-oriented configuration options.
The subsequent step is the definition of host hardening policies to contrast the detected vulnerabilities, where policies can be applied, in fact, in the presence of structural vulnerabilities no solutions at the user level can be implemented and the software needs to be patched and tested by its distributor. Having selected Docker as a target product and following the defined general hardening rules, it was possible to define a set of practical host hardening rules together with the command line instructions required to configure them and then verify the enforcement status on a running container. The testing of the applicability of the rules was done at the host level on a clean Docker installation on a Linux personal computer, by running a container based on an Ubuntu image retrieved from the Docker standard registry. The rules were then individually enforced on the container and the result of the enforcement was verified by checking the output of status commands. Since the process of enforcement and verification took several hours to complete the need to automate the verification process where possible. To obtain the automation of such a process a tool has been developed in the Python language that, requiring minimal user input, executes the verification commands and based on their output, reports on a text file if a host hardening rule has been enforced. The development of the tool was done on a standard Kali Linux OS.
Thanks to the collaboration with Drivesec the tool was then executed to test a Docker container located in a Linux host used in the development stages by the company, this is the reason why this container was purposefully made less secure to test the tool's accuracy in detecting enforced policies.

Chapter 2

Virtualization

Virtualization is a technology that uses software or hardware modules that are capable of creating a platform to run multiple OSs in a single physical machine. This functionality makes it possible for a system administrator to create different virtual spaces to divide critical applications from non-critical ones improving performances and security by separating the configurations of these programs. Another benefit of virtualization is that it allows the user a physical machine's full capacity by distributing its hardware resources among different user instances. The result of implementing virtualization is the simplification of access and management of software and hardware resources available to users or applications. This simplification makes the underlying infrastructure transparent for the users of the system, so in case it undergoes modifications it will not be able to detect the change and will continue to operate without being affected by it.

2.1 Virtualization Technologies

Nowadays there is a large amount of COTS virtualization products available, but many of these products are based on the same virtualization paradigm and consequently present similar characteristics. It is important to understand the differences between these virtualization technologies in order to be able to classify products in the right category.

2.1.1 Hypervisors

A hypervisor, depending on the specific case, can be classified as a software, firmware, or hardware module that creates, runs, and manages virtual machines. The hypervisor controls how the guest operating systems are executed and provides them with a virtual operating environment. Based on the layer to which the hypervisor is deployed there are two possible definitions of hypervisors.

Type 1 Hypervisors

Type 1 hypervisors use hardware acceleration software to carry out computationally expensive operations and are deployed directly on top of the hardware of the system without the requirement for a native machine OS or drivers. OSs and other services are executed inside the virtual environments instantiated by the hypervisor, called guests, and the hypervisor mediates their access to physical interfaces. The type 1 hypervisor is widely acknowledged as the best-performing and most efficient of the two hypervisors. These hypervisors are more scalable because their direct resource assignment capability enables resource allocation and physical resource optimization.

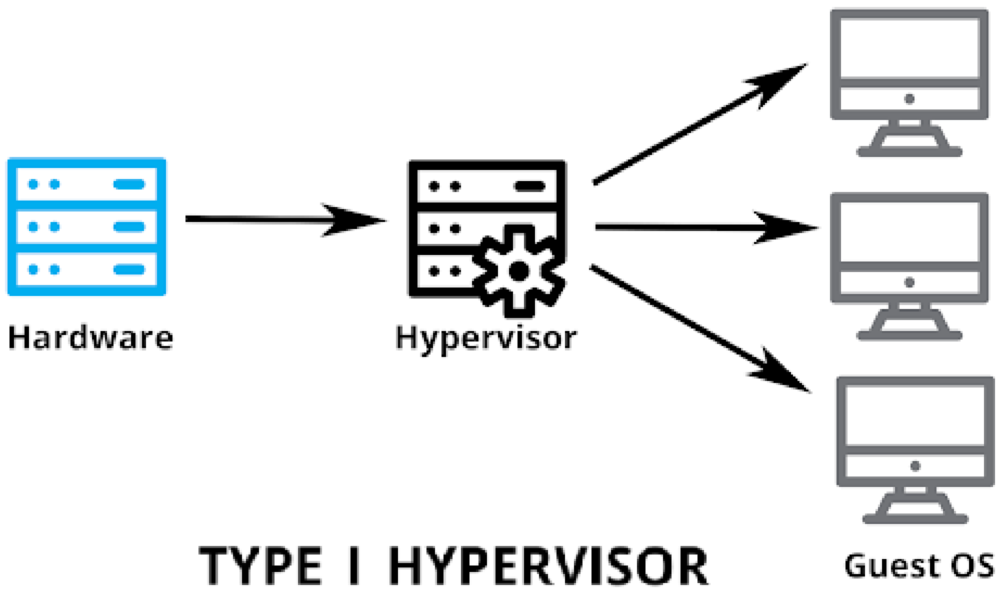


Figure 2.1: Type 1 Hypervisor architecture, [16]

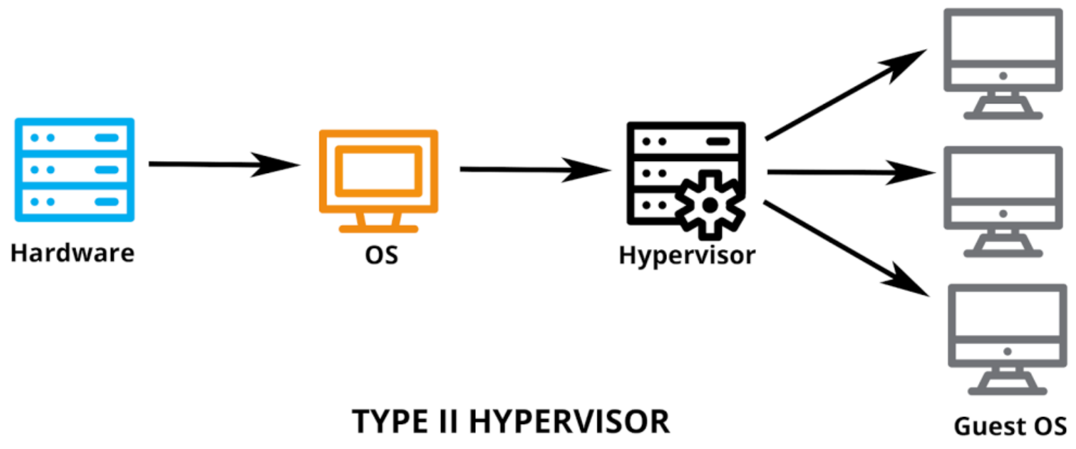


Figure 2.2: Type 2 Hypervisor architecture, [16]

One downside of this type of virtualization is that the configuration of a system executing a type 1 hypervisor needs advanced operating systems and computer architecture knowledge, therefore their primary applications are in professional areas.

Type 2 Hypervisors

Type 2 Hypervisors are built on top of the Operating System and run as a normal application inside the OS. Type 2 hypervisors support multiple virtual machines but are prohibited from having direct access to the host hardware and its physical resources. The Operating System, already present on the host machine, manages the system calls for memory, network resources, and storage. Due to not having direct access to physical resources type 2 hypervisor VMs can suffer some latency issues. This kind of hypervisor is simpler to set up than type 1 hypervisors and is compatible with almost any hardware, which is usually why this type of virtualization is mostly utilized by everyday users and for educational purposes.

2.1.2 OS Level Virtualization

OS-level virtualization is defined as an operating system capability in which the kernel permits the co-existence of several isolated user-space instances on the top of the operating system. The programs running inside those instances are not expected to see the difference between a normal

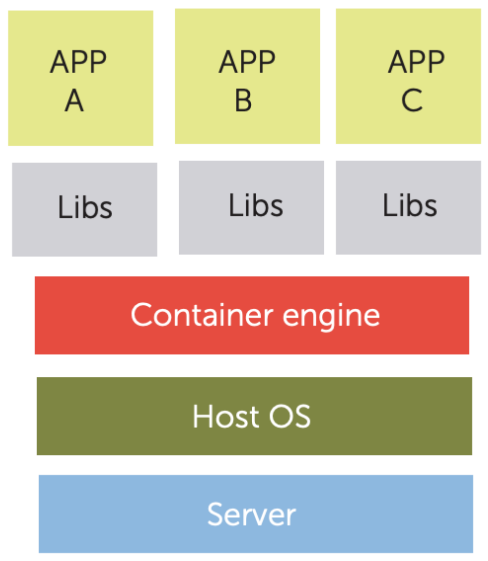


Figure 2.3: OS Level virtualization architecture, [19]

system and the virtualized one. Because OS virtualization renders access to hardware, there may be instances where the virtual OS is incompatible with the hardware. OS virtualization enables guests to execute different programs in these isolated user-space instances, which are usually called containers, each of which is assigned a specific amount of hardware resources and permissions. The software that runs inside the container sees those resources as the only ones accessible and considers itself to be running in a complete system in its own right.

Following this distinction between virtualization solutions, research was conducted to categorize virtualization products available to the public and to select two of these products for extensive analysis. This selection was based on features such as Open-Source software, an accessible and rich documentation, software usage popularity, and configuration options.

2.2 Virtualization Products

Having analyzed how virtualization technologies function, it is possible to categorize the most common products used for virtualization based on their behavior.

Xen Project Hypervisor
Xen Project Hypervisor is an open-source type 1 hypervisor developed by The Linux Foundation. Xen transforms the hardware present on the machine into a pool of computing resources that it can allocate to the guests it manages by creating a virtualization layer directly on top of the hardware. This hypervisor makes available to the user three types of virtualization: full virtualization, paravirtualization and a hybrid of the previous two called PVH, exclusive for Xen. The management of the system is done by controlling a privileged guest generated at hypervisor loading time called dom0. Xen can execute different OSs due to not being dependent on any operating system.

OKL4
OKL4 is a type 1 Hypervisor designed and distributed by General Dynamics Mission Systems. The OKL4 Hypervisor enables the ability to produce performance-optimized and secure environments because of its microkernel base while guaranteeing the separation of VMs by hosting guests ranging from complete OSs to device drivers in isolated unprivileged VMs. Being microkernel-based it benefits from a low-performance overhead and a small memory footprint to make it a viable solution also for resource-constrained devices. The hardware resources assignation can be dynamically scaled by the hypervisor itself by the OKL4 decision-making process so that applications and processes that execute in a dedicated operating system can access the resource they need without having to incur in lowered performances.

VMWare Workstation Player
VMware Workstation Player is an open-source type 2 hypervisor distributed by the company VMWare but currently supported only by the community. Being a type 2 hypervisor Player requires the presence of a base OS to run its software and create virtual machines, Player is the community version of software developed by VMWare called Workstation which requires a license. Originating fundamentally from the same product Player offers a reduced set of the features present in Workstation. Since Player VMs are stored on disk and updated when modifications (i.e. addition of files or directories, program installation) occur inside the virtual machine, this software offers the feature of saving a certain snapshot of an entire VM by simply saving the directory it is stored in.

Oracle VirtualBox

VirtualBox is a type 2 hypervisor developed and distributed by Oracle free of charge. One of the main positive points is the compatibility with a good number of OSs both from the software installation compatibility and from the ability of OS support for virtual machines. Oracle makes available packages for certain operating systems, usually the heaviest in terms of hardware resource consumption, devoted to improving performances. VirtualBox can manage multiple virtual machines, and configure their resource consumption, storage and emulation of their GUI. To communicate with the running VMs the host can use a variety of methods, such as shared clipboards or virtual networks, which can also be put in place for guest-to-guest communications.

Linux Containers

Linux containers or LXC is an OS virtualization tool integrated into the Linux kernel capable of running multiple virtual units, being dependent on the Linux kernel, LXC is only available for Linux OSs and capable of managing only Linux guests. The objective of LXC is to give the user the possibility of creating isolated environments, more specifically containers, inside the same Linux machine but without the need for different kernels. LXC bases its implementation on two Linux kernel modules called cgroups, used for hardware resource management and limitation, and namespaces that manage the accessibility to system resources for processes, this is used to make applications inside the containers behave like they would in a complete system. One of the advantages of this product is that a user classified as root inside a container is not seen as such outside of it.

Docker

Docker is a software that implements OS virtualization and is distributed and developed by Docker Inc. To implement virtualization Docker uses containers that can be built based on static files, called images, to improve its portability and distribution. Being based on container technology Docker makes available the kernel of the underlying host OS to further reduce the size of the container images. To turn a static image in a running container a dockerfile is needed, which is a configuration file where the execution options and configurations parameters can be stored, moreover, these parameters can be passed directly as options when managing Docker from a terminal.

Chapter 3

Virtualization Products Analysis

After a careful evaluation of commercial products, Docker and Xen Project Hypervisor were selected as the two virtualization products to be investigated in depth. This choice was based on different factors, for starters, they represent two different types of virtualization paradigms that are very common in industrial applications, Xen is classified as a Type-1 Hypervisor, which can be deployed directly on the target products in different industrial fields while Docker belongs to what is referred to as OS-Level virtualization, suitable to be implemented in testing environments or servers. Moreover, Xen is open-source software, developed by the Linux Foundation, Docker instead is developed by a private company and is available for free for private use but for enterprise use a paid license is required. For both software products, extensive documentation and support are available, both via official and community sources.

3.1 Docker

Docker is a software product implementing an OS-level Virtualization paradigm and its main purpose is to build, run and manage Linux-based containers. It allows for the packaging and running of applications in containers, which are essentially loosely isolated environments based on Linux cgroups and Namespaces, this isolation grants more security to the host and the applications running on it while giving it the ability to run many containers simultaneously, one independent from the other.
Containers include everything that is needed to run applications, exception made for some kernel modules that are provided by the OS, not having to virtualize a complete OS explains why containers are lightweight and have a lower resource consumption in contrast to type 2 hypervisors. Docker in contrast with other OS-Virtualization solutions shows higher portability since its containers can be run without needing to modify settings in the host operating system and their execution depends only on the installation of the Docker platform. Projects are very portable because of Docker container-based technology, which enables containers to execute on machines with various capacities from a developer's local laptop to cloud providers.
Because of its portability and lightweight, it is also simple to manage workloads dynamically, scaling up or down applications and services following the changing requirements of applications.

3.1.1 Architecture

Docker bases its container implementation on a client-server architecture. The Docker client communicates with the Docker daemon, which is in charge of managing the creation, execution, and distribution of Docker containers. Both the Docker client and daemon can be installed and executed on the same machine, or a Docker client can be connected to a remote Docker daemon, in this case, UNIX sockets, a network interface, or a REST API are used by the Docker client and daemon for communication.

Docker daemon

The Docker daemon is a service process that runs on the host operating system and is responsible for managing the Docker API calls, the objects to be used for the creation and execution of containers(i.e., images, volumes), and the communication with other daemons to implement online availability of the Docker environment.

Docker Client

The main tool for users to interact with the Docker environment and the containers is the Docker client, the client can be accessed via command line or with the use of a GUI called Docker Desktop. The user can send management commands or monitor container status using the Docker API, the client then sends these commands to the Docker daemon, which carries them out and provides the output of the command back to the client.

Docker Registry

A Docker registry primary function is to store and provide Docker images when requested. Anyone can utilize Docker Hub, a public registry, and by default, Docker is set up to search it for images. There is also the possibility to run a local privately owned registry, to create and store user created images.

Docker Images

Images are files containing the data for creating Docker containers. An image is frequently based on another image or created by a snapshot of

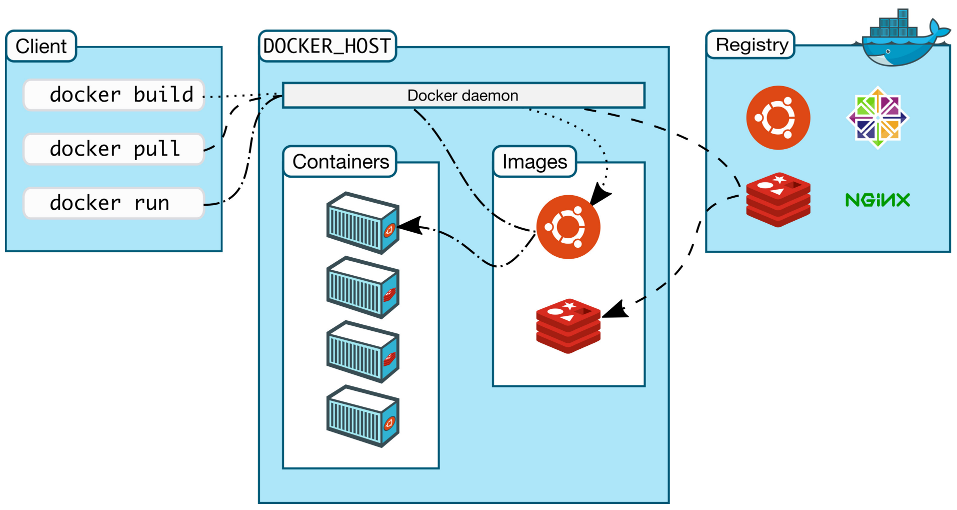


Figure 3.1: Docker architecture, [13]

an operating system in a specific status, then it can be modified to use it as a starting point. Building the image is not enough to be able to run it as a container, it is also necessary to write a Dockerfile, a file with a basic syntax, that defines the actions required to build and execute an image. Each Dockerfile instruction produces a layer in the image, and in case of modifications, only the layers that have changed are rebuilt when the Dockerfile is altered. This is the reason why images are so lightweight, small, and fast when compared to alternative virtualization technologies.

Docker Container

A container is defined as a running instance of an image. The management (creation, execution, deletion, etc.) of the container is done by using the Docker API. It is possible to attach storage units to a container, link it to one or more networks, or even construct a new image based on the state of an existing one. A container is defined by the contents of the image it is generated from as well as any configuration options that are provided to it when it is created or started, if options are not set the container is assigned a default set of rules and configuration parameters by Docker to enforce isolation from other containers and its host. The user can directly control these options with a configuration file or runtime options to define how isolated a container's network, storage, or other subsystems are from the surrounding environment. Docker offers the possibility to configure its containers with a large set of options some critical from a security standpoint, these options include but are not limited to:

PID Separation of processes on Linux hosts is provided via the PID namespace. The PID Namespace hides the system processes from visibility and permits the reuse of process identifiers, this means that two processes belonging to different PID Namespaces can have the same PID. The host's process namespace may occasionally need to be shared by the container. This effectively gives the container the ability to see every process running on the system, a possible use of this feature is the implementation of a container with debugging capabilities.

IPC IPC namespace provides separation of the access to ADT used for inter-process communication such as semaphores and message queues.

Network If not differently specified by the user or the image, containers have networking enabled and they can connect to the net or other containers. Communication with other containers is achieved over the default bridge, a network located inside the host machine and by default containers are connected to it at startup. Containers will use the same DNS servers as the host if not differently specified, in case this is not desired users can set up their DNS configurations, as an option of the command to start the container. By default, the MAC address is generated using the IP address assigned to the container otherwise the container's MAC address can be set explicitly by providing it via the *–mac-address parameter* option. This kind of configuration needs to be used carefully since it could cause network issues since Docker does not check if manually specified MAC addresses are unique.

User Memory User memory is the portion of main memory available to the user. Main memory is the memory directly accessed by the processor to read instructions and read or write data, one of the OS mechanisms implemented by OSs for memory management is memory swapping. Memory swapping enables a computer to store data in secondary storage and recover it when it is needed for usage in main memory, allowing the system to use a larger amount of memory than is physically available. By default, there is no memory limit for the container and it can use as much memory as it requires.

Kernel memory Kernel memory is different from user memory since as for kernel memory, algorithms for memory swap can not be applied. Being denied the possibility of swapping a container that gets stuck while in possession of a portion of the kernel memory could lead to a possible block of system services if the portion of reserved kernel memory is too large. Considering that kernel memory is not independent of user memory an issue in kernel memory consumption could result in issues with the main memory as well. By default kernel memory usage is not configured and a single container could claim it in its entirety.

CPU By default, all containers get the same priority when requesting access to the CPU. This is not ideal in case some processes run critical applications, this can be modified by changing the container's CPU share accessible by each container to create a priority system.

Storage The container can be given the possibility to write data on a disk. If this option is configured the container could end up taking up too much space in storage leading to reduced system performance or even denial of service. Docker has two possible options to provide disk space to a container, volumes, and storage drivers. The main difference between these options is that data in volumes is persistent while data in storage drivers are not. Although using volumes is safer since Docker assigns a 10GB default size limit if using storage drivers is necessary its implementation requires a specific configuration.

Privileges By default, Docker containers are executed without root privileges and are limited in what they can do, increasing the privilege level of a container to the host level allows him to act as a host itself. In case privileges are needed for a specific function (i.e. CAP_NET_ADMIN allows control over network interfaces) it is possible to assign the container privileges only related to that function's capabilities. Capabilities are a Linux feature that grants selected processes a set or single permission that would be usually only available in privileged mode. Docker executes the container with a certain set of capabilities enabled and the host is in charge of adding or removing them to suit his necessities. Enabling the wrong capabilities could lead to potential privilege escalation of malicious software while dropping necessary ones could lead to an incorrect execution.

3.2 Xen Project

3.2.1 Architecture

The Xen Project hypervisor is classified open-source type-1 or bare-metal hypervisor and for this reason, does not require the support of a host operating system. A type 1 hypervisor enables the simultaneous execution of several operating systems on a single machine, the operating systems executed do not require to be different instances of the same OS. Xen is the foundation for many other commercial programs, ranging in a wide array of uses including server and desktop virtualization, security software, and embedded application. The Xen Project hypervisor, is executed on the hardware immediately after the bootloader and it is in charge of managing the CPU, memory, timers, and interrupts. Numerous virtual machines can be run on top of the hypervisor, and a running instance of a virtual machine is referred to as a domain or guest. The system's drivers are all located in a special domain called domain 0, which also includes a control stack and other system functions for managing the whole system. Its main features are:

- Low memory impact: It is more reliable and secure than other hypervisors because its design is based on the use of a microkernel, which has a minimal memory footprint and interfaces to the guest. The majority of deployments use Linux as the primary control stack (also known as "domain 0"), however, some implementations with other control stacks can be used (i.e. Solaris).

- Driver assignement: The Xen hypervisor gives the possibility to decide which device drivers can be run in which VM. If a driver malfunctions or stops responding, there is no need of restarting the whole system, but it is sufficient to reset the VM in which the driver is contained.

- Paravirtualization: Paravirtualization is a virtualization method that provides the virtual machine with a software interface that is comparable to the actual hardware-software relationship, similar to what happens in a normal system where the OS uses its system calls to communicate with the HW. Guests that have been fully paravirtualized are optimized to run as virtual machines and where applicable, this enables these

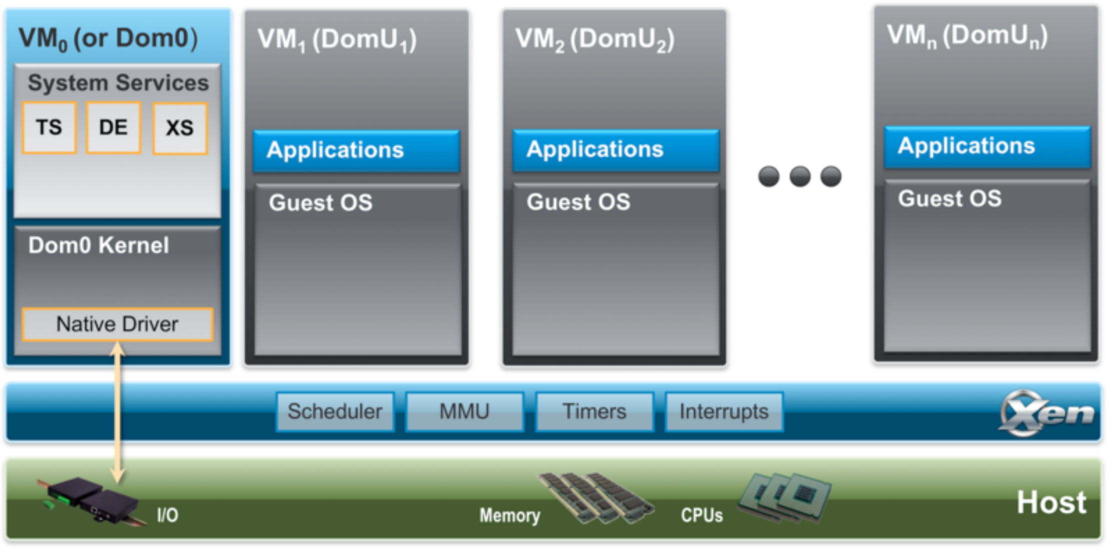


Figure 3.2: Xen Project Hypervisor architecture, [18]

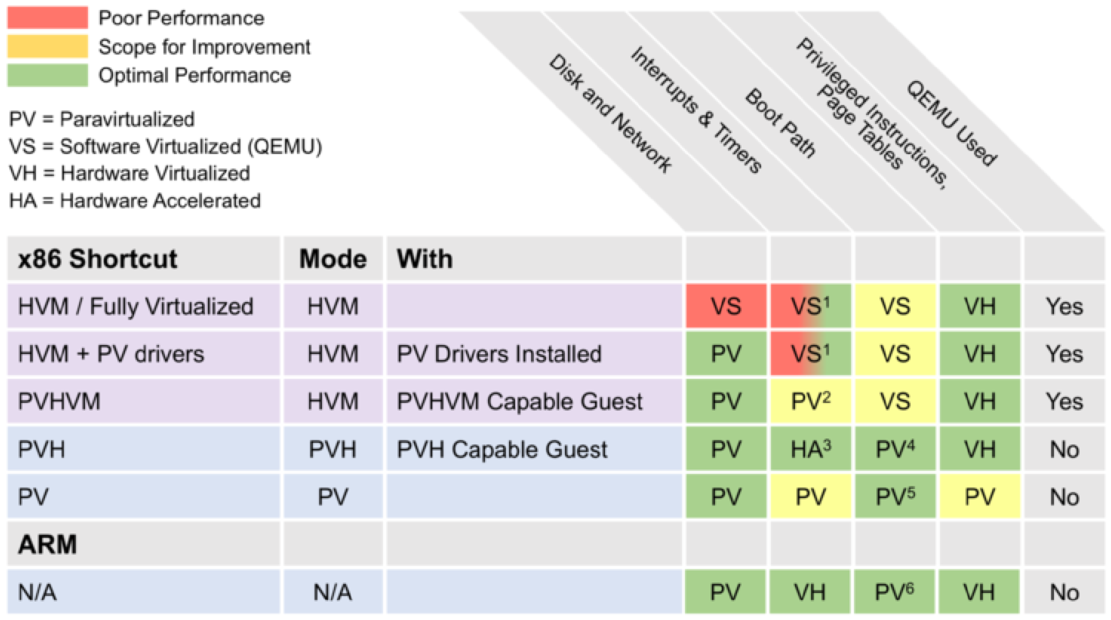
guests to operate much faster. The hypervisors that can operate with paravirtualization can also function on devices that do not support virtualization extensions.

3.2.2 Guests

- Paravirtualized Guests or PV Guests:
  Although paravirtualized guests need a PV-enabled kernel and PV drivers to run effectively without emulation or virtual emulated hardware, paravirtualization does not require virtualization extensions from the host CPU, making it perfect for running on older hardware. A Linux framework was used to allow PV in Linux kernels starting with version 2.6.24, so most Linux distributions can support PV (except for very old versions). This is why PV guests are typically used for old hardware and legacy images as well as in unique situations, such as running Xen inside of another hypervisor without the need to use nested hardware virtualization support or hosting containers.

- HVM Guests:
  Full virtualization, also known as hardware-assisted virtualization, or HVM, employs the host CPU's virtualization extensions to virtualize guests, but it also needs processor-specific hardware extensions, which are used to increase the performance of the emulation. Guests that are fully virtualized don't need kernel support to run. Due to the necessary emulation, fully virtualized guests typically operate more slowly than paravirtualized guests, but they are compatible with old operating systems that do not present PV kernel features. Xen Project software uses QEMU, a generic and open-source machine emulator and virtualizer, device models to emulate hardware, BIOS, and the necessary drivers for the system to run. If the guest has PV features HVM Guests use them to reduce latency and improve performance. Typically HVM virtualization performs best for general-purpose OSs (i.e. Windows, Linux).

- PVH Guests:
  PVH guests are the most recently introduced type of guests for Xen, they try to incorporate the best features of HVM and PV hosts by acting as HVM guests that can exploit kernel virtualization extensions usually reserved for PV guests. In contrast to HVM guests, PVH guests use



🟥 Poor Performance
🟨 Scope for Improvement
🟩 Optimal Performance

PV = Paravirtualized
VS = Software Virtualized (QEMU)
VH = Hardware Virtualized
HA = Hardware Accelerated

x86 Shortcut	Mode	With	Disk and Network	Interrupts & Timers	Boot Path	Privileged Instructions, Page Tables	QEMU Used
HVM / Fully Virtualized	HVM		VS	VS[1]	VS	VH	Yes
HVM + PV drivers	HVM	PV Drivers Installed	PV	VS[1]	VS	VH	Yes
PVHVM	HVM	PVHVM Capable Guest	PV	PV[2]	VS	VH	Yes
PVH	PVH	PVH Capable Guest	PV	HA[3]	PV[4]	VH	No
PV	PV		PV	PV	PV[5]	PV	No
ARM							
N/A	N/A		PV	VH	PV[6]	VH	No

Figure 3.3: Performance overview based on the type of guest, [18]

native operating system interfaces for virtualized timers, interrupts, and boot instead of QEMU. This is a way of combining the advantages of PV and HVM virtualization modes which was a major factor behind the birth of PVH guests. To achieve this kind of collaboration, the development process started with an HVM guest, gradually removing functionalities that were not essential and then Hardware virtualization support was implemented, a feature of PV, for memory and privileged instructions. While QEMU is not required for PVH guests to execute correctly a user-space backend can still use QEMU if it is needed for the implementation of some of its functions.

3.2.3 I/O Support

I/O is a fundamental part of any computer system and a possible performance bottleneck for virtualized systems. Xen divides its virtualization approaches for I/O devices based on the presence of PV drivers in the guest. Paravirtualization I/O is the best performing of the two approaches and is suggested for guests whose OS kernels include PV drivers, fully virtualized I/O instead while offering lower performances has its benefits in compatibility since it is the only option available when working with old operating systems.

Paravirtualization I/O

Xen Project Hypervisor provides two models for Paravirtualization I/O. In the first model, two PV I/O drivers are installed in the dom0 kernel, one for the front end and one for the back end, in a client-server schema. This model is capable of managing networking and virtualized storage.
In the second model, a raw disk interface is presented by a QEMU user space in the dom0 to its PV back end, which is then presented to the guest front-end driver. There is no difference between a back-end driver running in user or kernel space from the viewpoint of guests. Depending on the configuration option selected, Xen will automatically select the ideal pairing of front and back-end drivers.

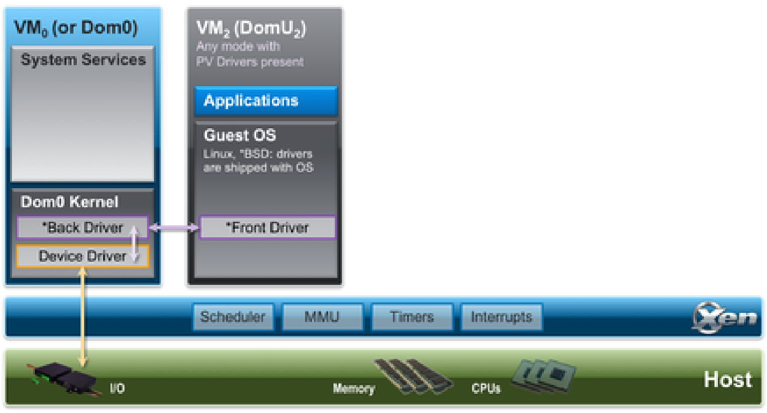


Figure 3.4: PV I/O Model 1, [18]

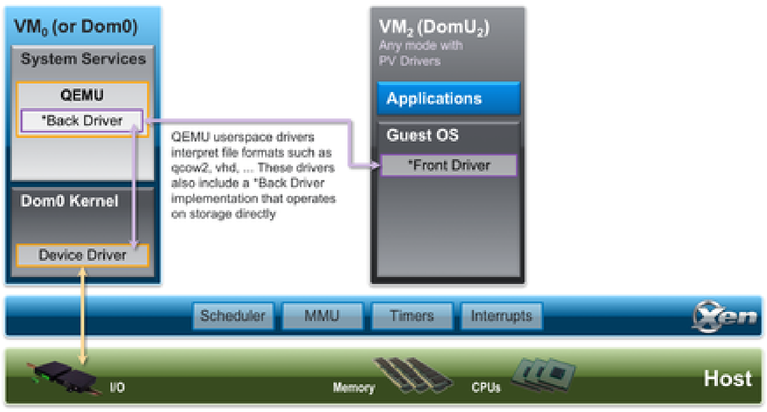


Figure 3.5: PV I/O Model 2, [18]

Fully virtualized I/O

To implement I/O for HVM guests which have no access to kernel PV drivers is used together with PV I/O Support. This kind of solution is only available to HVM guests since they can take advantage of this functionality, which is mostly used to simulate legacy devices that are required for a guest's boot procedure. This implementation is based on the presence of an emulated

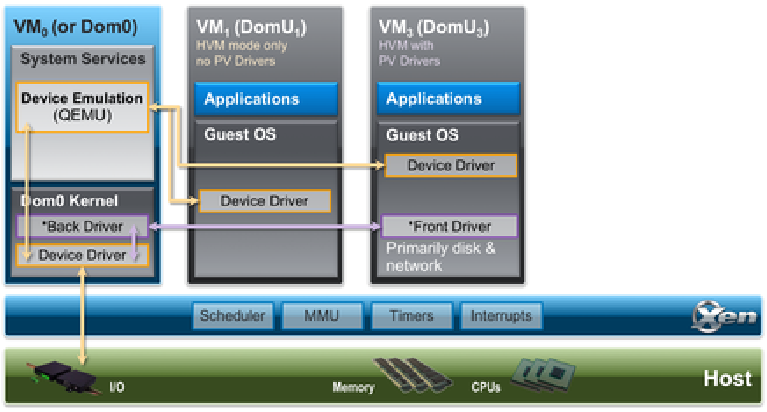


Figure 3.6: HVM I/O, [18]

device in the dom0 and then the guest has to be supplied with the related device driver to access the I/O device.

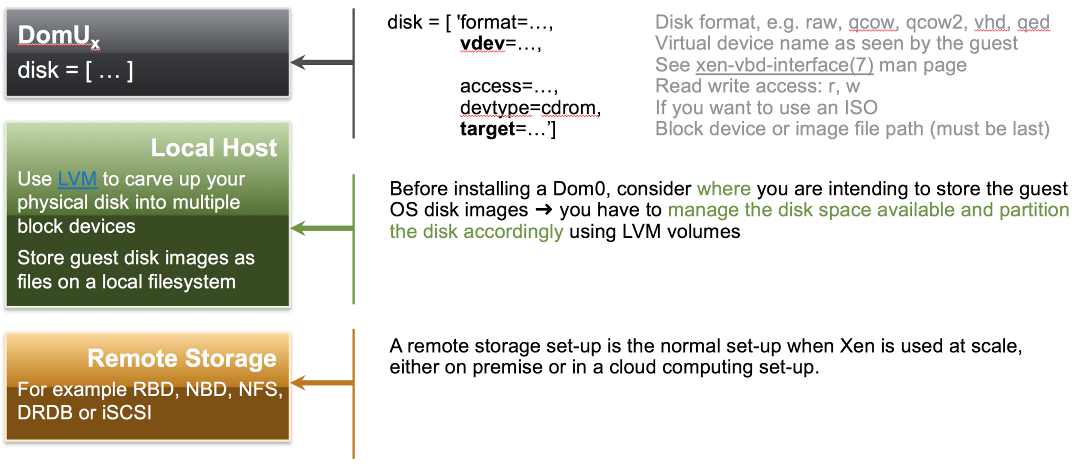
3.2.4 Storage

Xen can implement two methods for the management of storage and disks. The first possibility is to create separate partitions, using Linux Volume Manager, on the disk while operating as the dom0 and to assign them to the VM that requires it. The other possibility is the management of simulated disks via files, it is in fact possible to store disk images on the local filesystem and give permissions to VMs to access them. Another possibility for storage configuration is remote storage but its implementation and configuration is completely left to the user.

While using LVM has better performances due to driver availability using disk storage offers more flexibility in memory management, Xen gives the possibility of mixing the two approaches by creating a local volume with large dimensions used to contain disk images. Generally speaking, the use of LVM is more diffused in Xen implementations.

Logical Volume Manager

To be able to configure the usage of LVM as the policy for the implementation of guest storage there is the need to have a large space available on


DomU$_x$
disk = […]
Local Host
Use LVM to carve up your physical disk into multiple block devices
Store guest disk images as files on a local filesystem
Remote Storage
For example RBD, NBD, NFS, DRDB or iSCSI

```
disk = [ 'format=…,        Disk format, e.g. raw, qcow, qcow2, vhd, qed
         vdev=…,           Virtual device name as seen by the guest
                           See xen-vbd-interface(7) man page
         access=…,         Read write access: r, w
         devtype=cdrom,    If you want to use an ISO
         target=…']        Block device or image file path (must be last)
```

Before installing a Dom0, consider where you are intending to store the guest OS disk images ➜ you have to manage the disk space available and partition the disk accordingly using LVM volumes

A remote storage set-up is the normal set-up when Xen is used at scale, either on premise or in a cloud computing set-up.

Figure 3.7: Xen storage management, [18]

the main disk before creating the partitions for the guests. For this reason, the disk space dedicated to the use of LVM needs to be configured at the time of installation of the hypervisor.
Based on the control stack employed by the dom0 guest it is possible to use LVM to store its dedicated storage inside the LVM partition reserved for guest disk utilization.

One of the suggested divisions of storage in a Xen host is the following: up to 8GB for the dom0 guest and its memory swap requirements based on its OS and the rest of the disk made available for guest storage.

3.2.5 Networking

When configuring guests, their network parameters will be defined based on the control stack of the dom0 and the virtualization method of network interface, differentiating between PV Network Devices and Emulated Network Devices.

PV Network Devices

PV network devices despite using paravirtualization can be made available also for HVM guests, by installing the OS-compatible PV driver in the guest or in case its kernel is PV aware. The guest is then given access to single or multiple network interfaces that do not suffer from the HVM overhead of the complete emulation of the device and are capable of implementing fast network communications.

To implement PV networks there is the need for a pair of network drivers one front end and one back end. The first one will be located in the guest VM and the second is usually stored in the dom0. To the guest, the front-end device looks like a standard network interface while the back-end device situated in the dom0 is configured to contain the references to the guest.
The two devices are then connected by a virtual communication channel, and network traffic will be generated in the host, transmitted from the front-end device to the back-end device, and then routed onto the wider network usually by using Network Address Translation.

Emulated Network Devices

The other possibility to implement network access is the use of emulated network devices, which are necessary in case the host OS does not support PV. This requires the emulation of a complete hardware element dedicated to network functions, the emulated device will then run in the dom0 or in a predisposed guest as a process. In case the device is managed directly by the dom0 it is seen by it as back end device while in case it is run in another guest the dom0 sees it as a PV device and the guest needs to manage the forwarding between the HVM network device running as a program and the PV device.

In case a guest can implement PV drivers but for any reason, they do not result available the PV device can be paired with an emulated one to enable the network functions with the possibility of transferring them to the PV device if it becomes available.

Chapter 4

Vulnerabilities Discovery Process

Identifying a process that is capable of finding new vulnerabilities in a system is crucial to keep a system secure over time. To this purpose, following the NIST-800-30 document, a 4-step process has been identified: Asset definition, threat sources identification, threat event recognition, and vulnerabilities acknowledgment.
Asset definition is the first step in the creation of a process to identify new vulnerabilities. An asset can be defined as any data, device, or other components of the environment that supports the virtualization system.
Assets:

- Images A1
- Registries A2
- Containers, Guests A3
- Hosts A4
- Users A5

The second step is to identify threat sources, threat sources are those elements from which a new vulnerability that can compromise the correct functionality or the safety of any of the assets can originate.
Threat Sources:

- System administrators S1

A system administrator is a person which has full access to the administrative privileges inside a machine, capable of installing software and of changing sensitive configuration parameters.

- Hackers, external or internal attackers S2
  An attacker is an individual that has the objective of creating damage to the assets by rendering the system unusable or by appropriating confidential information.
- Software failure S3
  A software failure is defined as the malfunctioning of a software application that translates into unexpected behaviour, possibly capable of ignoring the restrictions and boundaries of privileges defined for that specific application.
- Software aging S4
  Every piece of software has an expected end-of-life being because it is superseded by products with more capabilities or better performances or being from the discontinuation of its development. The more a program stays not updated the higher the possibility of the discovery of vulnerabilities.
- OS vendors S5
  OS vendors are responsible for the distribution of OSs to the public and it is their responsibility to make sure that these products do not present faulty features that could be exploited by malicious users to implement exploits.
- Image builders S6
  When running virtualized environments the VMs or containers are generated starting from static files, these files are usually retrieved from online sources that could be trustworthy or not. The individual or the organization that distributes images to the public should be aware of the state of its components by monitoring it and deploying updates where and when necessary.
- Registry management S7
  Registry management is intended as every operation, strategy, and personnel devoted to the usage of the registry. Registries while not being a direct threat to systems can be exploited by attackers to deploy images containing malware or to intercept confidential data if the connection with the client is not secured.

The third step is needed to recognize threat events, which can be defined as events or situations that have the potential for causing undesirable consequences or impact. Every aspect of the assets needs to be considered and based on the threat sources we can define how that aspect can be compromised.
Threat Events:

- E1 - A1, potentially dangerous software, S1, S4, S6
  When building images there is the possibility that the individual managing the packaging included knowingly or not software that suffers from known vulnerabilities or in the worst cases software that contains malware. Since images are static files anyone using that specific image will be a possible target of attacks.
- E2 - A1, access control violation, S1, S6
  Image files could have embedded some configuration options related to permission and once the image is transformed in a running instance, this could lead to running virtual spaces with higher privileges than desired.
- E3 - A1, secret disclosure, S6
  When packaging images users should be careful not to include certificates, keys, and other confidential data that should remain secret. In case these sensitive files are placed inside the image anyone with permissions to access them can read their contents.
- E4 - A1, untrusted files usage, S1, S2, S6
  The origin of image files should be considered. When downloading new images the users should verify the source from which they are downloading the file and rely only on official sources. Downloading images from nonofficial sources can expose the system to risks since it is more probable that images contain defects or dangerous software.
- E5 - A2, file integrity, S1, S2, S7
  Registries should keep track of the changes applied to the files archived in their servers. This is useful in case unauthorised users modify images to include vulnerabilities, in case an altered image is delivered, the user will be exposed to risks.
- E6 - A2, out of date software distribution, S4, S7
  Registries should monitor the images they make available for users to download and remove the ones that are outdated until the patches are applied. This is because old images have a higher probability of

containing known vulnerabilities that can be exploited.

- E7 - A2, Intellectual property loss, S2, S7
  In case proprietary images are included in registries, they should be accessible only by the set of authorized users whose identity has been verified. Letting unauthorized users access these kinds of images could lead to the disclosure of company and personal secrets and the free distribution of proprietary software.
- E8 - A3, privilege escalation, S1, S3, S5
  If containers or guests do not have their permissions set correctly it is possible to exploit these weak spots in configuration to increase their privilege level to be able to access hardware or software resources that they are restricted to control.
- E9 - A3, unrestricted access, S1, S3, S5
  Users should be aware of the virtual elements in the system and should configure accesses to devices and permissions considering the functionalities that have to be implemented. Giving access to a resource or granting a certain privilege level for an instance that does not need it could expose the system to unnecessary risks.
- E10 - A3, host tampering, S1
  Only a few selected virtual elements should be able to communicate and interact with the host. A modification in the host can result in the malfunctioning of the whole system.
- E11 - A3, denial of service, S1, S2, S5
  A virtual instance that has unrestricted access to hardware resources (i.e. CPU shares, primary memory) has the possibility of monopolizing it rendering the other instances and the host unusable.
- E12 - A4, privilege escalation, S5
  In the host are generally present different users for different purposes. It is important that no user inside the host is able to increase its privilege level and act like the host. Furthermore,The host credentials should be accessible only to the few users that are in charge of managing the system.
- E13 - A4, filesystem tampering, S1, S5
  Host Filesystem integrity should be guaranteed at all times. Its filesystem contains files and directories that should not be accessible to all users

present on the machine, this is because a modification on those files could condition the correct execution of the environment.

- E14 - A4, denial of service, S5
  Both containers and VMs while being isolated spaces in many cases rely on shared resources made available by the OS or the host itself. If in any way one of these resources stops behaving correctly both the host and the virtual instances will not be able to operate correctly.

4.1 Container Technology vulnerabilities

Having defined threat events it is now possible to characterize and classify the possible vulnerabilities that could be present in Container virtualization technologies. The vulnerabilities are divided based on the asset they involve to be able to get an overview of how the asset functionalities can be affected.

4.1.1 Image Vulnerabilities

Since images are essentially archives that contain every component required to operate programs, those components embedded within the image may be out-of-date or lacking important security patches. An updated image can be free of known vulnerabilities for days, weeks, or months after being built, but if eventually weaknesses are found in one or even more image components, it will render the image out-of-date (E6).
When patching or updating containers, their execution has to be stopped and the updates have to be deployed in the image they are built on and then the container has to be restarted, unlike traditional software which is updated on the hosts it runs on. Consequently, a typical security risk in a container environment is the deployment of containers that have vulnerabilities due to the image version being not updated (E1).

Unknown origin images (E1, E4)

The utilization of untrusted software is one of the most frequent high-risk situations in any environment. Developers may be tempted to obtain images from unreliable or unofficial external sources because of the portability and simplicity of the reuse of containers. The risks associated with using this third-party images are similar to those associated with any unauthorized software, such as the introduction of malware, data leakage, or the inclusion of vulnerable components.

Image packaging risks

Images may potentially be injected with configuration vulnerabilities. Image configuration options are defined at image build time. If for instance, a container is being run with administrator privileges or with a namespace shared with the host and an image is generated based on the container whoever will run the image will inherit such runtime options that could lead to security issues (E2, E8, E9). Moreover, some applications use secrets for asymmetric cryptography. These secrets can be included directly in the image file system when an app is packaged as an image, but doing so poses a security concern because anybody who has access to that image can freely analyze it and acquire those secrets(E3). As said before Images can be seen as collections of files, hence harmful files may be inserted into them knowingly or unknowingly. These malware might go undetected to a superficial analysis and once “unpacked” they can target other hosts or containers in the environment because they would share the same permissions as the container they belong to (E1, E4).

4.1.2 Registry Risks

Connection to registers over insecure channels (E3, E7)

Images frequently include information classified as private or as company internal, such as personal data or proprietary software. If the image is downloaded or uploaded without the use of a secure connection its content could be vulnerable to cyber-attacks such as man-in-the-middle or spoofing.

Image archive management (E6)

Images are usually stored in registers for both users’ and companies’ utilization, if not properly managed this library of images may accumulate a large number of outdated, therefore possibly insecure images over time. The fact that these possibly vulnerable images are stored in the registry does not directly affect the user in its activities, but, if for any reason, one of these images is retrieved from the registry, it raises the possibility of compromising the system where it is utilized.

Insufficient authentication and authorization restrictions (E5, E7, E2)

Inadequate authentication and authorization enforcement can result in the loss of intellectual property and reveal important technical information about an application to an attacker because registries may store images used to execute proprietary apps, access confidential data, or run sensitive software. Even more significantly, as registries are frequently relied upon as a source of authentic, certified software, the breach of a registry raises the possibility of compromise of hosts and containers further down the chain.

4.1.3 Container Risks

Vulnerabilities within the runtime software (E8, E12)

Although they are generally rare, container runtime software vulnerabilities can be one of the most harmful vulnerabilities if they enable container escape. The possibility of accessing physical resources or/and data belonging to other containers and in worst cases the host, this kind of vulnerability is tightly related to privilege escalation.

Unrestricted network access (E11)

By default in most container environments, individual containers can contact each other and the host OS over a local network called the default bridge. Accepting this network traffic could put other containers or even the host at risk if a single container is compromised. For instance, an malicious container could be used to scan the default bridge to which it is connected for the discovery of other vulnerabilities in the configuration that the attacker could exploit. Since a significant portion of the connection between containers is virtualized, traffic on the network from one container to another looks different from normal traffic since many packet fields like source and destination addresses are not easily identifiable. This makes managing egress network access more difficult in a containerized environment. Security tools and IDS systems that are not aware of containers or are unable to manage this kind of special traffic are unable to examine these packets or detect their threat level.

Insecure container runtime configurations (E9, E10, E11)

Administrators are often given a lot of configurable settings by containers, and if these settings are configured incorrectly, the system's security may decrease. On Linux container hosts, for instance, the number of permitted system calls is frequently restricted by default to only those needed for the safe and correct functioning of containers, but if this number is increased, it may subject containers and the host OS to an elevated risk from a compromised container. Similarly to this, if a container is launched in privileged mode, it has access to every component on the host, enabling it to effectively function as the host OS and have control over every other container that is running on it.

Packaged software vulnerabilities (E1, E3, E11)

Since containers appear as complete systems to applications that are executed inside of them they are vulnerable to these software vulnerabilities like every OS. This is simply the indication of common software faults in a container environment, not an issue with containers per se.

Unregulated containers (E9, E11)

Rogue containers are unplanned or overlooked containers left in a running state in a system. This can happen frequently, especially in contexts where developers are testing their code by launching multiple containers. These containers may be more vulnerable to attacks if they are not put through rigorous testing or vulnerability scanning and correct configuration, which they usually aren't since this is usually the case in development scenarios. Rogue containers hence present a further risk to the environment, particularly if they continue to operate without the knowledge of the security administrators and development teams.

4.1.4 Host OS Risks

All host OSs have an attack surface, which is the set of points on the boundary of a system, a system element, or an environment where an attacker can try to enter, cause an effect on, or extract data from, that system, system element, or environment. The attack surface can be expanded, for instance, by the presence of any network-accessible service that offers a potential point of entry for attackers.

Shared kernel resources (E11, E13, E14)

Compared to general-purpose OSs, container-specific OSs have a substantially reduced attack surface. For instance, they lack the libraries and package managers necessary for a general-purpose OS to execute database and web server applications directly. This forces the container to use the libraries stored in the host creating a possible single point of failure. Despite the fact that containers offer software-level resource separation, using a common kernel always creates a wider attack surface than hypervisors, even among OSs designed specifically for containers. In particular core system components, like the cryptographic libraries needed by connections by remote connections to be authenticated and the kernel functions fundamental for general process initiation and management, are provided by host OSs, even container-specific ones. These components are susceptible to vulnerabilities just like any other software, and since they are located so low in the architecture of container technology, they can heavily affect the whole system.

Improper user access authorization (E12)

Considering interactive user login in the system should be uncommon, container-specific OSs are often not tailored to enable multiuser scenarios. When users control containers by logging in directly to hosts rather than through an orchestration layer, the system can be exposed to risks. for example, a user that solely has to manage the container for one particular app may be able to have a significant impact on many others because it may be able to consume all the resources available or gain access to the entire container.

Host filesystem manipulation (E13)

As stated before insecure container settings increase the threat of file manipulation on host volumes. For instance, a container may be able to alter files in sensitive host directories if it is permitted to mount them. Containers shouldn't typically modify the file system of the host OS and hardly ever alter locations that regulate the host OS's fundamental operations (e.g., /boot or /etc for Linux containers, C:\Windows for Windows containers). If a compromised container is given access to modify these directories, it might be exploited to escalate privileges and attack both the host and other containers that are being executed on the host.

4.2 Hypervisor Technology vulnerabilities

Hypervisor virtualization is a technology based more on the hardware, more loosely or strictly based on the specific cases, than OS virtualization. This introduces vulnerabilities in the modules in charge of managing those hardware resources, which can be tested by analyzing the code of the hypervisor, but this is out of the scope of this thesis. Due to this architectural characteristic, hypervisor vulnerabilities have been roughly divided into two categories: structural vulnerabilities and deployment vulnerabilities. The structural vulnerabilities are due to faulty implementations of the hypervisor code and can be detected by appropriate testing of the software, but, no user solutions can be implemented and a patch has to be deployed to solve the issue. On the other hand, deployment vulnerabilities depend on the configuration choices done by the system administrator and have to be monitored and validated to avoid critical security faults.

4.2.1 Structural threats to VM process isolation

Virtual Machine data structures management (E9)

The register states have to be managed correctly to plan a particular VM's work, for example, vCPU tasks, as each guest VM is assigned several virtual CPUs. A data structure is used by the hypervisor to allow the saving and loading of each vCPU's state if this data structure is implemented improperly it could lead to hypervisor memory leaks.

Sensitive instruction handling(E12)

On hardware platforms that do not support virtualization, a software procedure should be put in place to detect sensitive instructions, submit them to the hypervisor, and before actually executing these instructions on the hardware, substitute them with safer instructions. Any failure to catch these sensitive instructions or an inaccurate translation could lead a guest OS to execute privileged instructions.

4.2.2 Deployment threats to VM process isolation

Memory management unit (E8, E12)

Since guest VMs cannot be given direct access to the hardware-based Memory Management Unit (MMU) because it may potentially allow them to access memory assigned to the hypervisor and other VMs, the hypervisor employs a software-based MMU that generates a shadow page table for every VM. However, a flawed software-based MMU implementation could result in the leakage of data in unpredictable address spaces, including memory sections assigned to the hypervisor and the VMs located on it, breaching memory isolation.

DMA (E8, E9, E10)

To enforce memory isolation for device drivers and programs employing direct memory access (DMA), the hypervisor makes use of the hardware I/O Memory Management Unit. The enabling of this capability is managed by a firmware switch integrated into the hypervisor. If left unused, this feature might create a vulnerability where one VM could use the DMA as an attack vector to replace the contents of physical RAM addresses used by the other VMs that are being run in the hypervisor.

Resource allocation (E11)

Good isolation management necessitates that denial of service is avoided by allocating to each VM the appropriate memory and CPU resources required for its hosting applications. It’s crucial for system stability and, consequently, security to provide sufficient memory through proper memory allocation options configuration and proper virtual CPU allocation with suitable vCPU assignment options setup.

4.2.3 Structural Device Emulation and Privilege Escalation Threats

Emulation of storage and networking devices (E9, E10, E14)

The hypervisor is in charge of both the emulation of storage and network devices and control of the accesses to them from the various VMs, which is handled by a kernel-based code. Since guest VMs normally cannot access the physical devices directly until they are given access to it, the hypervisor kernel intercepts any I/O call from a guest VM and forwards it to this code. This code is responsible for the emulation of devices, the mediation of access to them, and also multiplexes the actual device. In case this code is not properly checked and its integrity and error absence verified it could lead to severe security issues.

Execution of privileged operations (E12, E14)

By employing mechanisms like VM Exits, which perform actions specific to processor architecture, or Hypercalls which are hypervisor-specific functions similar to system calls, the hypervisor can perform some privileged actions, like Memory Management. The entire virtualized host could crash if certain activities weren't properly validated, for example, if a VM's Control Block was allowed to be fully accessed or if input checking was not being performed. This is a design flaw that can only be fixed by thoroughly testing the hypervisor code.

4.2.4 VM Lifecycle Management Threats

VM images monitoring(E1, E4, E5, E6)

As already discussed for Docker when image repositories are not monitored continuously old images can still be available for deployment, this can lead to the usage of possibly vulnerable software or OS versions. Moreover, images can be retrieved from non-standard sources and if these images are not properly analyzed from a security point of view some malicious software can be introduced into the system without the system administrator knowing. Any image created as a snapshot of the system at a certain point in time and then restored could diverge from the standard because of insufficient monitoring, and missing updates that might lead to any of the platform-level threats.

Management of the Hypervisor (E11)

The management of the hypervisor is intended as the general administration of a hypervisor host and it is typically carried out through virtual consoles. This kind of management is core to safeguarding the host hardware resources. One possibility for the implementation of this function is by utilizing the dom0 as a monitor for the resource utilization of all the other virtual machines. This kind of solution implies that the dom0 is analyzed for vulnerabilities offline before the execution of the other VMs and maintained secure throughout the hypervisor life cycle. When hosting a hypervisor, several traditional security solutions might not be reasonable. For instance, if a network attack occurs on a system that is not virtualized, the problem can be fixed by simply turning off the port on which the attack is being directed or the targeted network interface. However, a hypervisor host cannot deploy this strategy since multiple operating VMs may share a single port on the actual network interface card of the hypervisor host. Rather, a particular security solution is required, such as turning off the virtual network interface of VMs that utilize those ports.

Chapter 5

Host Hardening Principles and Policies

Once the risks that could affect a virtualization product have been identified it is possible to define the host hardening practices to be applied to mitigate their impact. This was done by identifying general host hardening practices that could apply to the specific product and considering them as a starting point to define the practical host hardening policies based on the practical commands that can be used in the setup of the software.

5.1 General Xen Hypervisor host hardening practices

The first step to the safe execution of a Xen hypervisor is the verification that its components meet integrity requirements, this can be done via a verification process based on the hardware of the host. This is required since the hypervisor operates in direct contact with the low-level architecture of the system and can be subject to rootkit attacks and the execution of unauthorized code. To guarantee this component integrity the hypervisor and the underlying hardware need to be able to support what is defined as a measured launch environment (MLE). An MLE is a set of hardware, firmware, or software resources that are integrity verified and have the capability of starting up a system. This can be obtained if the host processor supports a hardware module that is capable of guaranteeing its integrity and therefore classify it as the starting element of the chain of trust that starts from the hardware through the BIOS and then to the hypervisor components. Moreover, a pre-kernel module is necessary for the hypervisor code since the hypervisor executes as the first module and it is the one in charge of launching the secure boot process. The purpose of this pre-kernel module is to ensure that the appropriate component is selected between the hardware to enable the systematic evaluation of the hypervisor modules or any other software running on that hardware. The integrity of the launched hypervisor components will be guaranteed by enforcing authentication employing cryptographic algorithms; this authentication also makes sure that the system only executes authorized code. The integrity verification of the hypervisor components is not sufficient for the safe execution of the hypervisor, functionalities such as process isolation, device emulation and access control, and VM lifecycle management need to be guaranteed to increase the system security level.

5.1.1 Process Isolation

Process isolation can be implemented in a hypervisor environment by following a set of selected rules. Firstly, not all privileged instructions coming from the guest VMs can be directly executed on the CPU but they need to be managed by a privileged entity that can be trusted, it being the hypervisor or a special guest that acts as a monitor. Memory management has to be implemented and supported to reduce buffer overflows attacks and VM escape attacks, this is when a VM accesses a memory location that belongs to another VM, to the minimum. Algorithms have to be put in place to ensure the correct assignment of system resources such as primary memory, CPU shares, disk access, etc. this guarantees the correct execution of the guests inside the machine addressing issues like denial of service and resource starvation.

Hardware Assisted Virtualization

Hardware-assisted virtualization is a platform virtualization approach that enables efficient virtualization using help from hardware capabilities. Full virtualization is instead used to emulate a complete hardware environment, or virtual machine executing in complete isolation.

Instruction Set Virtualization Instruction Set Virtualization is available only in specific processor architectures that have two possible operation modes: root and non-root. Each of these modes has four privilege levels going from Level 0, the highest, to Level 3, the lowest. The hypervisor can be increase its protection level from instruction set-type attacks by guests by executing in root mode and the guests operating systems executing in non-root mode regardless of the privilege level they are assigned. This solution addresses privilege escalation and VM escape at the local level but is it possible that a virtual machine that can be controlled via remote connection could perform such actions by networking protocols.

Memory Virtualization Hardware-assisted memory virtualization is provided when the hardware enables the mapping of the Guest OS's physical addresses in their respective page tables to the host's physical addresses using hardware-based page tables instead of hypervisor-generated shadow page tables. This brings a reduction in the size of privileged code executed providing security advantages.

Hardware-assisted virtualization functionalities can also be provided by some software solutions, which is the only possibility in case a system does not support virtualization extensions. Being implemented at the software level these products do not carry the same guarantees and performances as the hardware-based ones. In particular, exclusively hardware-based solutions have these characteristics:

- Better memory management controls
- Better isolation of I/O devices, if direct assignment of I/O devices is supported it eliminates the need for providing emulated device driver, reducing the size of untrusted code
- Guest OS code and hypervisor code execute at different permission levels
- Privilege-level isolation provides better protection
- If full virtualization is supported, COTS versions of OSs can allow for easier patching and updating
- The hypervisor size will be reduced, enabling faster security testing

Resource Management

VM primary memory allocation The hypervisor is in charge of managing the primary memory assignment to VMs to satisfy the memory requests for each of them during its lifetime. Generally speaking, a VM does not need the entire amount of memory that it is assigned at configuration time, this allows the hypervisor to assign to VMs a total quantity of memory larger than the one physically available. This is done to dynamically allow VMs to use the memory they require without getting limited to a level that could hinder its performance, this principle, called overcommitting, can improve the performance of a system and avoid resource starvation and denial of service. On the downside, it may negatively affect memory-sensitive processes and it is not the optimal solution when a VM requires heavy memory utilization.

Another element to consider when configuring memory parameters for VMs is the ratio of the physical RAM size to kernel swap size. Setting this value to a low threshold could lead some applications to not run properly, effectively blocking some programs. Setting minimum guaranteed memory to each VM is helpful in case overcommitting is utilized to avoid that memory being monopolized by a single guest and guaranteeing the possibility of execution to all the other VMs running in parallel. Configuring a priority system for memory assignment to manage guests that are running applications that expect immediate memory availability could allow these VMs to access the resource they need if at that moment they are being utilized by a low-priority virtual machine.

VM CPU allocation Similarly to memory allocation, CPU allocation should have as objectives the definition of upper and lower bounds of CPU computing power utilization. Failing to set up the correct configuration of these thresholds could lead to resource starvation or denial of service in case a VM starts behaving uncontrollably due to errors in its current processes or in the worst case if it has been compromised. Usually, the management of the division of clock cycles is done by considering an average of the consumption of the VMs present on the system (i.e. if a system has 4GHz available to be used by VMs and the average is 800MHz the system could ideally run five VMs). The possibility of using a priority scheme is applicable also for CPU resource allocation.

5.1.2 Devices Emulation and Access Control

In case an application performs an I/O call, the request is redirected to the emulated device assigned to that specific virtual machine. The emulation code is executed either in a special VM or in the hypervisor kernel as a kernel module and the role of this code is to call the I/O device drivers present in the hypervisor to access the actual physical device. Usually, these drivers do not necessitate running in privileged mode and consequently, in case malicious code is included in their implementation their ability to interfere with the normal flow of execution of the system is reduced. In some particular cases, there is the need of executing such drivers in privileged mode, that is the same mode in which the hypervisor runs, and in these situations, those special drivers should undergo a formal verification process to guarantee their safe functioning. If device access is not properly configured some VMs that should not normally be able to communicate with certain devices could gain access to them. This can be avoided by configuring access control lists at the device level in order to only allow selected VMs to interact with them or by setting up a whitelist for accesses at the hypervisor level defining which guest has access to which device. This solution is valid when talking about fully virtualized guests, in case para-virtualization is used these VMs can directly access the hardware devices without having to request it from the hypervisor, for this kind of user is required a different approach. A possibility to face this issue is the setting of a bandwidth limit for each authorized VM to access the I/O and network virtual interfaces, to avoid DOS attacks.

5.1.3 VM Lifecycle Management

In a hypervisor environment, VMs are created starting from images which are static files and can be considered templates to be customized based on the user's needs. To guarantee that images are trustworthy a series of security rules should be applied to the libraries in which they are stored. When a user needs to access a library, authentication should be requested to guarantee that only authorized users can access and modify its contents. If the library happens to be located on a remote server the connection needs to happen over secure channels to avoid tampering and the possibility that a malicious user could intercept the image if transmitted in clear. While it might not look like a critical issue, often images contain proprietary apps or confidential information that should not be disclosed outside of a company environment. As a consequence of this, integrity is one of the key security properties that should be guaranteed to safeguard the security of systems. Image integrity can be verified in case a digital signature is appended to the image for validation purposes.

5.2 General Docker host hardening practices

In essence, containers are wrappers around Linux control groups (cgroups) and namespaces. Cgroups are used in the Linux kernel for monitoring and restricting resources among a group of processes (i.e. two processes inserted into the same cgroup are applied the same limitations). Namespaces determine which information about the system a process can access (i.e. the PID namespace restricts which processes can be seen within a container).
Containers are isolated from one another by default configuration, but each container shares with the host its kernel creating a possible single point of failure, in fact, if a malicious container manages to compromise the host all the containers being executed are at risk. Similarly, some shared libraries in the host could be used by multiple containers, this could pose a risk since theoretically there is a possibility that one container could gain access to the other by exploiting weaknesses in the library.

5.2.1 Registries

Containers should only connect to registries over encrypted channels. The key goal is to ensure that all data exchanges with a registry occur between trusted endpoints and is encrypted in transit, this guarantees confidentiality and integrity for the images. Registries should be monitored continuously either from a trusted software or by a designated person and rules should be put in place to manage the obsolescence of the images contained. Old OS versions, outdated applications or libraries, should be detected by periodically scanning for version numbers in the image registry, and in case an out-of-date image is found set a flag to warn any possible user of its status.

5.2.2 Images

Before building a container from any image, it is preferable to check and validate image configuration settings, to avoid any intentional or unintentional misconfiguration that could negatively impact container and system security. Secrets such as private keys or proprietary software and data should be stored outside of images and should be provided dynamically at runtime. The distribution of this sensible material should be managed by access lists configured by system administrators, this approach guarantees that only the necessary information is provided to containers, and in case of an attack the affected resources are easily identifiable and the proper remediation actions can be enforced. In the development stages of the creation of a containerized environment is not unusual to use different versions of the same image that maybe are retrieved from different registries. This behavior could lead to the acquisition of images from unreliable sources, this can be avoided by creating a set of trusted registries and images to work with.

5.2.3 Host

Docker Engine and OS

Before operating any system is good practice to ensure that its operating system is updated to the latest release and the same consideration can be made for the any software that has to be executed. Not installing updates could leave known security flaws open increasing the possibility of attacks. Since the OS is the largest attack surface making sure the system is up to date drastically reduces the probability of a successful exploit.

Docker Daemon Socket

The Docker daemon socket is a Unix network socket that facilitates communication with the Docker API. By default, this socket is owned by the root user, if any other user obtains access to the socket, they will be able to operate at the same permission level as the root user, it is crucial to set the correct Linux permissions to the socket directory and file. In case the Docker container requires to be accessed by a remote controller it is possible to associate the daemon socket with a network interface. Enabling this kind of access needs to be implemented with the utmost care, generally if this is not needed the feature should be disabled.

Filesystem and permissions

A simple and effective security practice is to run containers with a read-only filesystem. This can prevent malicious activities such as deploying malware in the container.
Many files stored on the system generated by the installation and configuration of the Docker environment contain sensitive information. It is imperative to assign the correct permissions to those files to avoid unauthorized users or attackers accessing or modifying their contents. In case the Docker Client and the Docker daemon are located on different machines there is the need of implementing a secure connection with the TLS protocol, the Docker host, that is where the daemon is running, will have to act as a server and manage certificate and keys. These new files and directories need to be protected from tampering.

5.2.4 Containers

Privileged Containers

Docker provides a privileged mode for containers, which lets a container execute with root privileges on the local machine. Running a container in privileged mode provides the container with some capabilities that should be assigned exclusively to that host, the most dangerous from a security standpoint are:

- Privileged access to every system device (i.e. network interfaces, disks)
- Ability to interfere with Linux kernel security modules
- Ability to install a new instance of the Docker platform, using the host's kernel capabilities, and run Docker within Docker.

These features can lead to the tampering of system resources, other containers, and the OS itself. Therefore, the use of privileged containers is not recommended, especially in the production phase of a product where the possibility of an attacker gaining control over this kind of container would generate the most damage.

Container Network configuration

Docker containers need to manage two types of network connections, the first one is the one configured between containers to allow them to communicate with each other, and the second one is the connection to the internet via IP address. By default Docker configures a newly created container to connect to the “internal” network called the default bridge, this behavior is not security compliant. Only the containers that should be able to communicate have to be placed on the same network, but it is not recommended to connect them all to the default bridge, instead, a custom bridge can be created to better isolate the communication. If a Docker environment contains many containers as many networks as needed can be created to configure inter-container communication.

Container resource management

When a container is compromised or behaving uncontrollably, it may consume a physical resource such as primary memory or network bandwidth in its entirety, causing the other containers on the system to fail or not work correctly. Setting up containers with usage limitations to the hardware resources is one of the countermeasures to minimize the impact of resource exhaustion.

Container isolation

Development teams should create an optimized environment when talking about container isolation, to ensure that containers are not capable of interfering with the other containers and the host. One of the functionalities that a host operating system should provide is the assurance of protection of its kernel from attacks coming from containers, this is one of the most critical issues since the kernel is shared by all the elements on the host. Configuring the following Linux security features can greatly increase container isolation resulting in a more secure system:

Linux namespaces The Linux kernel from version 2.4.19 offers a feature called namespaces that divides the OS environment in such a way that processes assigned to a certain namespace can access only a defined set of system resources. This is how applications running inside containers do not detect differences from running in a complete system.

Capabilities When running processes, the root user typically receives privileged consideration. When this user ID is detected, the kernel and programs are typically configured to ignore the restriction of some actions. This user is therefore free to do almost everything. A process can be set up to access a portion of the root privileges through Linux capabilities. This feature successfully divides root privileges into discrete entities more manageable and flexible. Then, independent access to each of these entities can be given to processes. By doing this, the set of privileges assigned to processes is narrowed, lowering the possibility of exploitation.

Cgroups A Linux kernel feature known as control groups, or simply "cgroups," enables processes to be grouped into hierarchical units so that the utilization of different types of resources may be restricted and evaluated.

SELinux The Linux kernel security module known as Security-Enhanced Linux (SELinux) offers the means for providing access control security policies. SELinux is essentially a collection of user-space utilities and kernel modifications that have been included in multiple Linux distributions. Its architecture reduces the amount of external software required for security policy enforcement. Its usage improves and expands the namespaces restrictions.

AppArmor AppArmor is a Linux kernel security module that allows the creation of application profiles by an administrator to limit certain functionalities such as network access and disk permissions. These profiles are loaded into the kernel at boot time and can be assigned two modes of operation, enforcement and complain, while enforcement mode will actively stop processes that try to violate the profile rules complain mode will report the violation attempt.

Seccomp Seccomp is a Linux kernel security feature that, similarly to AppArmor, allows or denies applications to execute a certain set of system calls based on a profile. For Docker, its default seccomp profile is enabled for all new containers, which enforces basic security principles, if modifications are implemented to this profile it could lower its effectiveness or indicate a possible security breach.

5.3 Docker specific host hardening practices

Having pointed out the objectives and the available tools available for the implementation of host hardening in a Docker environment it is now possible to define the practical rules that enforce the general Docker host hardening rules. These rules that are ordered based on the asset they affect have almost a one-on-one correspondence with a command or an option to be set when configuring the host and the container.

5.3.1 Host

/etc/default/docker file ownership is set to root:root
The /etc/default/docker configuration file should be owned by the root account and group since it contains parameters that could change the way the docker daemon behaves, imposing this restriction ensures that unprivileged users cannot modify its contents.

/etc/default/docker file permissions are set to 644 or more restrictively
The /etc/default/docker configuration file should have its permissions set to rw-r–r– since it contains parameters that could change the way the docker daemon behaves, imposing these permissions ensures that unprivileged users cannot modify its contents.

/etc/sysconfig/docker file ownership is set to root:root
The /etc/sysconfig/docker configuration file should be owned by the root account and group since it contains parameters that could change the way the docker daemon behaves, imposing this restriction ensures that unprivileged users cannot modify its contents.

/etc/sysconfig/docker file permissions are set to 644 or more restrictively
The /etc/sysconfig/docker configuration file should have its permissions set to rw-r–r– since it contains parameters that could change the way the docker daemon behaves, imposing these permissions ensures that unprivileged users cannot modify its contents.

daemon.json file ownership is set to root:root
The daemon.json configuration file should be owned by the root account and

group since it contains parameters that could change the way the docker daemon behaves, imposing this restriction ensures that unprivileged users cannot modify its contents.

daemon.json file permissions are set to 644 or more restrictively
The daemon.json configuration file should have its permissions set to rw-r–r– since it contains parameters that could change the way the docker daemon behaves, imposing these permissions ensures that unprivileged users cannot modify its contents.

docker.service file ownership is set to root:root
The docker.service configuration file should be owned by the root account and group since it contains parameters that could change the way the docker daemon behaves, imposing this restriction ensures that unprivileged users cannot modify its contents.

docker.service file permissions are to 644 or more restrictively
The docker.service configuration file should have its permissions set to rw-r–r– since it contains parameters that could change the way the docker daemon behaves, imposing these permissions ensures that unprivileged users cannot modify its contents.

docker.socket file ownership is set to root:root
The docker.socket file should be owned by the root account and group since it contains parameters that could change the way Docker remote API behaves, imposing this restriction ensures that unprivileged users cannot modify its contents.

docker.socket file permissions are set to 644 or more restrictively
The docker.socket configuration file should have its permissions set to rw-r–r– since it contains parameters that could change the way the Docker remote API behaves, imposing these permissions ensures that unprivileged users cannot modify its contents.

Docker socket file ownership is set to root:docker
Since the Docker daemon runs as root in case a unprivileged user is able to gain ownership over this file it might enable him to have control over the docker daemon therefore to interact with containers. Moreover when installing Docker, the installer configures a docker group, the system administrators

can add or remove users to it, being included in this group grants read and write privileges to the Docker Unix socket. This privileges are given to any group that share the ownership of the socket. For these reasons the Docker socket file should be owned by the root account and group owned by the docker group.

Docker socket file permissions are set to 660 or more restrictively
The Docker socket configuration file should have its permissions set to rw-rw—- since only root and the users part of the docker group ought to be permitted to read and write

/etc/docker directory ownership is set to root:root
The /etc/docker directory should be owned by the root account and group since it contains certificates and keys, imposing this restriction ensures that unprivileged users cannot modify its contents.

/etc/docker directory permissions are set to 755 or more restrictively
The /etc/docker directory should have its permissions set to rwxr-xr-x since it contains certificates and keys, imposing these permissions ensures that unprivileged users cannot modify its contents, but can still use them.

Docker server certificate file ownership is set to root:root
In case the Docker host is acting as server over a secure connection, there is the need of securing the certificate it uses for authenticating to avoid any unsanctioned modifications. The server certificate file should be owned by the root account and group to ensure that unprivileged users cannot modify its contents.

Docker server certificate file permissions are set to 444 or more restrictively
In case the Docker host is acting as server over a secure connection, there is the need of securing the certificate it uses for authenticating to avoid any unsanctioned modifications. The server certificate file should have its permissions set to r–r–r– since no user should be able to modify its contents but it has to be readable for providing authentication.

Docker server certificate key file ownership is set to root:root
In case the Docker host is acting as server over a secure connection, there is

the need of securing the certificate key it uses for its digital signature to avoid any unsanctioned modifications or readings. The server certificate key file should be owned by the root account and group to ensure that unprivileged users cannot modify its contents.

Docker server certificate key file permissions are set to 400
In case the Docker host is acting as server over a secure connection, there is the need of securing the certificate key it uses for authenticating to avoid any unsanctioned modifications or readings. The server certificate key file should have its permissions set to r——— since no user except the root should be able to read its contents.

TLS CA certificate file ownership is set to root:root
In case the Docker host is acting as server over a secure connection and its CA certificate has been issued by a certification authority, there is the need of securing it because of its role in the authentication process to avoid any unsanctioned modifications. The server CA certificate file should be owned by the root account and group to ensure that unprivileged users cannot modify its contents.

TLS CA certificate file permissions are set to 444 or more restrictively
In case the Docker host is acting as server over a secure connection and its CA certificate has been issued by a certification authority, there is the need of securing it because of its role in the authentication process to avoid any unsanctioned modifications. The server CA certificate file should have its permissions set to r–r–r– since no user should be able to modify its contents but it has to be readable for providing authentication.

Docker is allowed to manage iptables
iptables is a firewall located in the Linux kernel that filters incoming IP packets based on tables. Docker needs to have access to iptables in order to configure egress and inter-container communication for containers, if the option is enabled Docker configures iptables automatically after container creation. This setting is useful to reduce the workload of the system administrator and to avoid configuration errors that could cause containers to not be able to communicate. Potentially this option can be turned off but the network configuration of containers for the host side must be done manually.

Containers are restricted to connect to the default bridge
The default Docker configuration does not impose any restriction on network traffic between containers. To improve the security of the system only those containers that require to communicate with each other have to be connected by the creation of a custom bridge tailored for them. Allowing all container to be connected to the same network could lead to sensitive information leaks or secret disclosure. Therefore there is the need to restrict containers from connecting to the default bridge.

Docker daemon TLS authentication is setup
In case there is the necessity of controlling containers remotely Docker gives the user the possibility of making the Docker daemon available over a TCP port, but any user that has access to that port might obtain complete control over the Docker daemon. In this cases it is good practice to configure TLS authentication for the Docker daemon effectively reducing the number of users capable of contacting the daemon to a closed set.

Logging level is set to 'info'
The Docker daemon has the capabilities of recording events related to containers. This logging can be set to acquire different sets of information based on the option selected. Setting the log level to info guarantees that in case of container crashes enough information can be recorded.

Monitoring of the docker group
Docker allows the user to mount host directories inside containers, this gives the possibility to mount the '/' directory inside a container. In this case the container would be able to modify the host file system as a privileged user. So if an unprivileged user is part of the docker group it could act as a privileged one by just being part of this group.

Monitoring of Docker disk usage
Docker uses a single directory in the host file system for storing all its files including the images. In case this directory size is not checked regularly it could grow its size at the point of rendering Docker and the host in a not responsive state. To avoid this occurrence it is possible to configure Docker to store these files in a different directory, but, if this is not possible, the directory has to be monitored and emptied of unused files regularly.

SELinux is enabled and profiles are configured
SELinux adds an extra layer to access control policies enforced by namespaces, if a user or process requires access to a certain file SELinux will check their security level and then decide to grant or not the access to the file. If SELinux is active in the host, after having defined a security policy this can be enforced on containers to further increase their isolation.

5.3.2 Container

Docker socket is not mounted inside containers
In case a container has the Docker socket mounted in its file system it would allow it to run Docker commands and consequently to be able to control the Docker daemon and all the containers running in the host.

Containers are restricted from acquiring additional privileges
Docker containers can be run with the option: *–security-opt=no-new-privileges* that denies that process the ability of gaining new privileges. This sets a bit in the kernel that ensures that any child process generated by the parent and the parent itself is restricted of acquiring new privileges. Setting this option makes sure that the processes created in the container stay unprivileged if the container is.

Privileged containers are avoided
Docker provides the *–privileged* flag as option to assign to a specific container all the Linux capabilities, overriding the ones belonging to the cgroup to which it belongs, effectively allowing a container to run at the same privilege level as the host. It is recommended that containers are assigned single capabilities tailored to their functionalities instead of enabling them all and creating security risks.

Container health check is enabled at runtime
Container health is a general indicator of the status of a container. Enabling the option *–health-cmd* will make possible to check the container health status at runtime. In case the check highlights problems further analysis has to be done to restore the container in a healthy condition.

Restart policy is set as 'on-failure' to a count between 3 and 10
In case a container terminates its execution it will check its restart policy option to decide if restart or not. This option should be set as *on-failure* in

order to restart the container in case the container crashes. It is necessary to select a number of maximum restarts because if the container is being crashed by an attacker it could try to restart an infinite amount of times possibly leading to a denial of service. Moreover restarting a container indefinitely without checking the exit status does not create the premises for fixing the issue that is causing the restarts. It is suggested to limit the containers restarts between the count of three and ten.

CPU shares are set on containers
By default, containers have the same priority to access hardware resources. If some containers need to have the precedence of execution over others Docker offers the possibility to assign a priority to containers by assigning CPU shares. This blocks lower priority containers to utilize CPU computing power that is needed by high priority containers, that are able to access the CPU time they require, moreover it limits less secure containers in case they are compromised and try to consume all the resources. This option needs to be used cautiously since it could lead to resource starvation scenarios for low priority containers.

PID limit is set
If a container is compromised or a bad piece of code is executed a fork bomb could be launched. A fork bomb is a type of cyber attack in which a process repeatedly duplicates itself to exhaust the system's resources, causing the system to crash or become unresponsive to the point of requiring a host restart. By limiting the number of PIDs inside a container the execution of fork bombs can be avoided.

Containers memory availability is limited
By default a container has access to the entire amount of primary memory available in the host. Limiting the amount of memory accessible by each container can avoid intentional or unintentional resource exhaustion events on the host. On the other hand setting this value too low could lead to containers to not execute properly.

Docker default seccomp profile is Enabled
By default Docker enables its default seccomp profile for all its containers. Seccomp is a Linux kernel security feature that, in case of Docker, whitelists a set of system calls for use. Most system calls are not used by users and processes and from a security point of view is advantageous to reduce the

set of accessible ones since it reduces the attack surface of the kernel that is a possible single point of failure of the whole system if compromised.

Docker aufs storage driver is not used
The aufs storage driver is the oldest storage driver available on Linux systems. The aufs driver is known to cause serious kernel crashes and it has only legacy support within systems using Docker. Most notably, aufs is not a supported driver in many Linux distributions using latest Linux kernels. This classifies the aufs storage driver as a legacy module optimised neither for performance nor security, therefore it should be avoided.

Container belongs to the Docker cgroup
System administrators have the ability of creating cgroups suited to their necessities. By default containers are run in the Docker cgroup but at runtime containers can be assigned any cgroup by the host thus granting or removing additional privileges. Using a custom cgroup for containers can create a security risk, if a container with elevated privileges obtained by a particular cgroup membership is compromised it could prove much more dangerous than the default option.

Container does not have access to the host IPC namespace
IPC namespace provides isolation of ADT structures such as semaphores and message queues used for inter process communication. By not sharing the host IPC process isolation is guaranteed between containers and the host. The processes inside the container would be able to see all communications on the host system in case the host's IPC namespace was shared with the container. Any attacker that could manage to control a container with a shared IPC would be able to access any communication happening on the host between processes. It is imperative that the namespace is not shared.

Container does not have access to the host network namespace
When a container shares the network namespace it fundamentally uses the network stack of the host separating the container from its networking part. A container that has enabled the option *–net=host* has full access to the physical network interfaces having the ability to open reserved ports. Such behaviour needs to be avoided since it could significantly lower the system security level, in case it is compromised an attacker would have enough permissions to shut down the host completely.

Container does not have access to the host process namespace
The PID namespace is used for process isolation in Linux machines. Its purpose is to create different process id spaces in order to allow the creation of two separate processes with the same process identifier on the same host. It is used to mask system processes from user processes by "hiding" them in another PID space, if a container is given access to the host PID namespace it would not only allow the container to see all this processes but could even restart their execution indefinitely or kill them completely, just as the root user. It is clear that a container with such privileges should be avoided if not strictly necessary.

Container does not have access to the host user namespace
The user namespace is used to restrict root processes in the container. If the user namespace is not shared a privileged process executed in the container will correspond to a non privileged process in the host. Having access to the host user namespace removes the separation between container user and host user leading to possible privilege escalation exploits.

Container does not have access to the host UTS namespace
UTS namespace is used to mask the hostname seen by processes. A process in a container does not require to be given the actual hostname, in case the namespace is shared a process inside a container coul potentially change the hostname of the Docker host.

AppArmor Docker default profile is enabled
AppArmor is a Linux kernel security module available by default in some Linux distributions. AppArmor, similarly to seccomp, protects the OS and applications by assigning profiles to processes that implement a set of security rules and restrictions. Docker has its default AppArmor profile that should be left enabled for its containers, but if a custom profile is required it should be configured carefully to not be too restrictive or too permissive.

Container root file system can not be modified
The container should not be allowed to modify its own file system. Since containers can be created from snapshot of the state of a previous container. Giving the possibility to a container to modify its own file system could be potentially dangerous in case the container is reused in the future but with different security restrictions.

Sensitive OS directories can not be modified by containers
Since as of the writing of this thesis there is not a Docker option to separate the container from the host mount namespace one possible solution to avoid containers to write in the host filesystem is to mount sensitive directories it as read only. Some host OS directories contain sensitive information and files that should not be available inside containers, in case one of these directories has to be mounted it is imperative to avoid read-write mode. Generally these directories include but are not limited to: '/', '/boot', '/dev', '/etc', '/lib', '/proc', '/sys', '/usr'.

Linux capabilities are configured for the container
By default, Docker containers are assigned a limited set of Linux capabilities at startup. Based on the function to be executed by the container this set can be enlarged or reduced, with the objective of minimizing the number of capabilities assigned to each container while still guaranteeing its correct execution. Leaving containers a larger set of capabilities from what they require could lead to potential security risks since capabilities are essentially root permission split in smaller units to be assigned.

Containers are restricted to use privileged ports
Usually ports numbers below 1024 are labeled privileged ports. Docker by default maps container ports automatically by assigning them to a free port in the range 49153-65535 of the host well away from the privileged ports. Generally unprivileged users are not allowed to use privileged ports, but Docker does and if a user sets up manually the port mapping he has the possibility to assign container ports to host privileged ports. This should be avoided since security sensitive data are usually exchanged on them and the host should manage this data and distribute it to the containers that need it.

5.3.3 Image

HEALTHCHECK monitoring setting is included in the image
The HEALTHCHECK instruction should be added to the docker images before execution in order to ensure that health checks are executed while containers are in running state. Adding this instruction to the container image guarantees that Docker periodically checks the running containers to make sure that containers are still operating correctly. If this is not the case, the Docker engine could try to restart or even terminate the "unhealty" ones.

Image has been tested for vulnerabilities
Docker has made available a tool for testing docker images for known vulnerabilities. While the tool is not available for free use it is the most effective way to analyze images before their execution.

A user for the container has been created
Many images once executed present only the root user in the container, it is a good security practice to avoid this by defining a user in the Dockerfile before building the image, in order to give access to the root user only to authorised users. This is particularly useful when the container is used from different people of which just a restricted set needs to operate with root privileges.

5.3.4 Registries

Private insecure registries are not used
Docker classifies private registries based on the availability of TLS authentication or in presence of a valid registry certificate, if any of these two elements is present Docker will consider the private registry as secure. In case Docker has classified the registry as secure it will store its certificate in the host /etc/docker directory. Insecure registries could distribute tampered images, leak some private data or be subject to man in the middle attacks, due to these reason the use of these registers should be avoided.

Registry certificate file ownership is set to root:root
The /etc/docker/certs.d/<registry-name> file should be owned by the root account and group since it contains the certificate of the registry that should not be modified, imposing this restriction ensures that unprivileged users cannot modify its contents.

Registry certificate file permissions are set to 444 or more restrictively
The /etc/docker/certs.d/<registry-name> file should have its permissions set to r–r–r– since it contains the certificate of the registry that should not be modified, imposing these permissions ensures that no users can modify its contents.

Chapter 6

Host Hardening Report Generator Tool

Having defined the set of specific host hardening rules in the previous chapter, a testing phase started. A Docker community edition environment was setup on a personal computer running Kali Linux to test firstly the practical implementation of each specific rule and secondly to find the correct way to test its enforcement. This was done by running a standard Ubuntu Bash container obtained from the Docker Hub, and then the default status was checked for each applicable rule before the implementation of each host hardening policy, then if necessary the container was restarted by including the right option in the docker run command to enforce the rule, and in the end the status of the rule was checked again to compare the differences of the two outputs. After having defined the options and commands necessary for the enforcement and verification of rules the process of verification was repeated and took several hours to complete. This result showed the necessity of automating such a verification process for it to be applicable in a real development environment of a company, intending to reduce time consumption and user errors.
The process automation was obtained by collecting all the verification commands in a single Python script and executing them by making use of the subprocess library that provides functions capable of running commands and capturing their standard output and error contents in strings. Immediately after the execution, the script requests the user to insert some information

such as the location on the disk of the image from which the container was created and the root password to be able to execute certain commands. The script then starts to send one command at a time and waits for its output, after having collected it the script evaluates the string and based on its contents detects if a security rule has been enforced or not. Once the decision process is terminated the script reports the result of the verification of each rule on a text output file. The output file can be considered a report of the security status of the assets defined in chapter 3, and it lists the security properties based on the asset they affect. The script generates a new report each time it is executed since it names the report file based on the date and time of execution, this is done purposefully to keep track of the security status of the Docker environment over time. The tool has some limitations too, one of them is that it requires that only the container to be tested is running in the system, which is not optimal for development environments where multiple containers could be running. Furthermore, due to testing limitations, not all the rules defined in section 5.3 were enforced.
After having produced the first stable version of the host hardening tool, I was able to test it on a Linux machine configured by Drivesec. The company provided a Docker environment and a container that was stripped of many security properties on purpose to test the accuracy of the script in detecting the enforcement of the host hardening rules and the execution time in presence of many missing policies. After several hours of testing and implementing changes following the suggestions given by the company, an improved second version of the tool was drafted.

Host hardening report

The latest version of the tool, produced the following report as a result of its execution in the Docker environment provided by Drivesec.

```
DOCKER HOST HARDENING STATUS REPORT 11/11/2022 14:08:00

Docker Version is up to date...................................................WARNING!

The tool could not check for docker updates
--HOST--

/etc/default/docker file ownership is set to root:root..........................OK!
etc/default/docker file permissions are set to 644 or more restrictively........OK!
/etc/docker directory ownership is set to root:root.............................OK!
etc/docker directory permissions are set to 755 or more restrictively...........OK!
/etc/sysconfig/docker file ownership is set to root:root........................SKIPPED!

Could not locate the file on the system, the check will be skipped
```

```
/etc/sysconfig/docker file permissions are set to 644 or more restrictively.....SKIPPED!

Could not locate the file on the system, the check will be skipped
daemon.json file ownership is set to root:root.................................OK!
daemon.json file permissions are set to 644 or more restrictively..............OK!
Docker socket file ownership is set to root:docker.............................OK!
Docker socket file permissions are set to 660 or more restrictively............OK!
docker.socket file ownership is set to root:root...............................OK!
docker.socket file permissions are set to 644 or more restrictive..............OK!
docker.service file ownership is set to root:root..............................OK!
docker.service file permissions are set to 644 or more restrictive.............OK!
Docker is allowed to manage iptables...........................................OK!
Containers are restricted to connect to the default bridge.....................WARNING!

Network traffic between containers is unrestricted, to avoid this restart
the docker daemon with the option --icc=false

Docker daemon TLS authentication is setup......................................OK!
Logging level is set to info...................................................WARNING!

 Restart the docker daemon with the option --log-level=info or edit the
 daemon.json file to include the option "log-level" : "info"

Users allowed to control the Docker daemon:
user1,user2
In case one of the listed users is unrecognized use the command:
gpasswd -d <user> docker

--CONTAINER--

Container running.............................................................OK!
Docker socket is not mounted inside containers................................OK!
Container is restricted from acquiring additional privileges..................WARNING!

Restart the container with the option: --security-opt=no-new-privileges

Privileged containers are avoided............................................WARNING!

Restart the container without the option: --privileged

Container health check is enabled at runtime..................................OK!
Restart policy is set as 'on-failure' to a count between 3 and 10.............OK!
CPU shares are set on containers..............................................WARNING!

The container has unlimited access to the cpu, to configure it restart the container
with the option (x=1024: max, x=1: min): -cpu-shares x

Containers memory availability is limited.....................................WARNING!

The container has unlimited access to the ram, to configure it restart the container
with the option: --memory xGB

PID limit is set..............................................................WARNING!

The container can create an unlimited number of processes, to avoid this restart the
container with the option: --pids-limit x
```

```
Docker default seccomp profile is enabled......................................OK!
Docker aufs storage driver is not used.........................................OK!
Container belongs to the Docker cgroup.........................................OK!
Container does not have access to the host IPC namespace.......................OK!
Container does not have access to the host network namespace...................WARNING!

Restart the container without the option: --net=host

Container does not have access to the host process namespace...................OK!
Container does not have access to the host user namespace......................OK!
Container does not have access to the host UTS namespace.......................OK!
AppArmor Docker default profile is enabled.....................................WARNING!

If a custom AppArmor profile has been configured you can ignore this message.
In case no configuration was setup and this warning is active please enforce the
default profile by using --security-opt="apparmor:docker-default"

Container's root file system can not be modified...............................WARNING!

Restart the container with the option: --read-only

Sensitive OS directories are not mounted inside containers.....................WARNING!

The following directories were mounted on the container:

/lib
/
/dev
The following capabilities were added to the container:
[ALL]

The following capabilities were removed from the container:
[]

Containers are restricted to use privileged ports..............................OK!


--IMAGE--

No image path has been given, Image policies will be skipped

--REGISTRIES--

Insecure registries are not used...............................................OK!
Registry certificate file ownership is set to root:root........................SKIPPED!

Could not locate the file on the system, the check will be skipped
Registry certificate file permissions are set to 444 or more re-strictively....SKIPPED!

Could not locate the file on the system, the check will be skipped

Number of enforced policies regarding A3 (container):11/20
Number of enforced policies regarding A4 (host):14/16
Number of enforced policies regarding A2 (images):0/0
Number of enforced policies regarding A1 (registries):1/3
```

```
END OF REPORT
```

Each security rule verification test has 3 possible outcomes: OK, WARNING, and SKIPPED. If the rule verification test result is reported as OK it means that the policy was correctly configured on its target and the tool was able to verify its presence status. In case a rule verification test result is reported as WARNING it means that the tool was able to execute the test and obtained a status but it is different from what is expected. Usually, this signifies that a rule is not enforced and a fix should be applied, but for specific implementations, some security rules cannot be configured, for example, if a container needs to communicate with another there is the need of enabling inter container communication, and therefore the check of that specific rule will never be marked as OK. The last possible output is SKIPPED, if this result appears as the outcome it means that the tool was not able to check the enforcement of the policy. The tool is based on a standard installation of Docker and configuration files and directories have specific paths in the host filesystem, if these files are moved and the tool is not able to find their new location it will report it, by defining the test as skipped due to not being able to find its target.
One of the report's characteristics is that its contents do not list any private or proprietary information that could be somehow linked to a specific container or environment, because in case elements like the container's IP address or OS version were to get leaked this could lead to possible security breaches. In case the report detects that a security recommendation has not been enforced it also outlines basic suggestions on how to implement the security option and increment the system's security. Moreover, at the end of the file, a brief recap of the number of rules enforced is reported to give the user a general overview of the results.
Executing the script periodically during the development phase of a Docker project can help monitor the security status of each container to deploy fixes and patches during an early stage exposing the final product to a reduced number of security risks. The script could also be useful at the deployment stage since keeping track of these reports could help highlight unwanted changes to the container configuration possibly identifying a compromised container.

Chapter 7

Conclusions

The purpose of this thesis was the creation of a process that aims at creating a set of rules to be applied at the host level to harden its security status in a virtualized system. The study started with the analysis and definition of the virtualization principles used nowadays, this was done to classify virtualization software products commercially available. Consequently, a choice had to be made by selecting two elements from this set of software products to be thoroughly studied and their working mechanisms explained, thanks to the help of Drivesec the set of software was reduced to two products Xen Project Hypervisor and Docker.
Having selected the software, the following step was to understand their workflow to identify the most critical assets used from a security standpoint, recognize the threat sources associated with them and point out the possible threat events that those threat sources could create to such assets. The process of identification of assets, sources, and events, implemented by following the NIST Guide for Conducting Risk Assessments document, led to the classification of vulnerabilities based on the asset involved and the threat event caused. Having pointed out the vulnerabilities it was possible to conduct a research to find possible countermeasures to said vulnerabilities for the two products. Due to different reasons such as the testing environment, the feasibility of testing, and Drivesec requirements the study of countermeasures was conducted differently.
For both products were identified a series of general host hardening policies, but for the reasons stated before the creation of specific host hardening rules

and consequent tool implementation has been done exclusively for Docker. These specific host hardening rules are based on the available commands and options found in the Docker command line reference documentation and follow the same structure as the general rules by being classified based on the asset they affect. The set of specific rules is quite large thus the necessity emerged of automating the enforcement of the verification process to reduce errors and time consumption, this need was fulfilled by the creation of a tool that produces a report text file containing the outcome of the verification for each rule and to offer possible practical solutions to enforce the missing rules. The tool was then tested in an ad hoc Docker environment provided by Drivesec, to test the accuracy and execution speed of the tool to employ it in the development and support stages of future projects.

The host hardening report generator tool has been developed to work with only one running container at a time, expanding the scope of the tool could lead to further optimization. By requesting reports based on container id there would not be the need of stopping the other running containers, even further the tool could be modified to analyze every container running in the host and produce a single report containing the gathered information.
The tool works by checking every rule every time it is run but for specific situations, it could be of use excluding certain tests that are bound to fail due to implementation requirements, which could show that the system is in a vulnerable state. So making it possible for a user to exclude the testing of some rules could increase usability and efficiency. Moreover, Docker's new versions and updates should be monitored in case a functional mechanism of the software is changed or new security functionalities are implemented and if they are in line with the general host hardening rules, including them in the tool can help expand its security coverage.
The report generated by the tool is a simple text file and should be used for informative purposes to support the development of a Docker environment. Since anyone with access to the host can access the reports and modify their contents, in case of an internal attack these files could be manipulated to show that certain security properties were configured on a container at a certain point in time when they were not. To improve the reliability of the reports a mechanism for guaranteeing the integrity of these files should be put in place. This could be implemented in several ways such as the generation of a message digest to be stored offline or in secure storage to be retrievable when required.

The practical part of the thesis is focused mostly on Docker. In the future, it might be useful to define a set of specific host hardening rules for Xen Project Hypervisor and develop the relative report tool to assess its status. Moreover, since the process defined in this thesis for the definition of host hardening rules is general, it could be applied to any virtualization paradigm that could be developed in the future. This process could potentially help secure any environment employing virtualization features from cloud services to embedded systems.

Bibliography

[1] Karen Scarfone Murugiah Souppaya John Morello. «Application Container Security Guide». In: (Sept. 2017).

[2] Ramaswamy Chandramouli. «Security Recommendations for Server-based Hypervisor Platforms». In: (June 2018).

[3] Jakub Szefer Diego Perez-Botero and Ruby B. Lee. «Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers». In: (May 2013).

[4] Paul Cullum. «A Survey Of The Host Hypervisor Security Issues Presented In Public IAAS Environments And Their Solutions». In: (Dec. 2020).

[5] Mukesh Gidwani Nancy Arya and Shailendra Kumar Gupta. «Hypervisor Security - A Major Concern». In: (Aug. 2013).

[6] Tahir Alyas et al. «Container Performance and Vulnerability Management for Container Security Using Docker Engine». In: (Aug. 2022).

[7] Thanh Bui. «Analysis of Docker Security». In: (Jan. 2015).

[8] Imtiaz Ahmad Sari Sultan and Tassos Dimitriou. «Container Security: Issues, Challenges, and the Road Ahead». In: (May 2019).

[9] Vartul Goyal. *Docker Security Implementation.* May 2022. URL: `https://www.linkedin.com/pulse/docker-security-implementation-vartul-goyal?trk=articles_directory`.

[10] Susanta Nanda Tzi-cker Chiueh. «A Survey on Virtualization Technologies». In: (2005).

[11] Datadog. *OOTB Rules.* 2022. URL: `https://docs.datadoghq.com/security_platform/default_rules/`.

[12] Tenable. *Audits.* 2022. URL: `https://www.tenable.com/audits`.

[13] Docker. *Documentation.* 2022. URL: `https://docs.docker.com/`.

[14] Michael Kerrisk. *The Linux Programming Interface.* 2010.

[15] Charlotte wright. *All You Need to Know About Hypervisors.* 2019. URL: `https://blog.resellerclub.com/what-is-a-hypervisor-and-how-does-it-work/`.

[16] Nirajan Koirala. «Docker Performance Evaluation». In: (Sept. 2022).

[17] OWASP. *Docker Security Cheat Sheet.* URL: `https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html#rule-11-lint-the-dockerfile-at-build-time`.

[18] The Linux Foundation. *Xen Project Hypervisor Wiki.* URL: `https://wiki.xenproject.org/wiki/Main_Page`.

[19] Pavan Sutha Indukuri. «Performance comparison of Linux containers (LXC) and OpenVZ during live migration». MA thesis. Karlskrona: Blekinge Institute of Technology, 2016.

[20] Ltd. Huawei Technologies Co. «Cloud Computing Technology». In: (Oct. 2022).