

# Forum:[Hack Forums](#)

Thread: [//TUT// Utilizing LD\\_PRELOAD to write ring3 Linux rootkits](#)

?YY? - April 14, 2015, 4:01 pm

I wrote and I have shared this on other forums, although this time, instead of reposting the guide, I'm going to improve it.

What is LD\_PRELOAD?

LD\_PRELOAD points to a shared object and loads it before any other shared objects, allowing you to write your own libc calls before anything else is loaded. LD\_PRELOAD is used in rootkits such as Azazel and Jynx2 - which will be left at the bottom of this thread. Typical usage of /etc/ld.so.preload is echoing a file path into the file.

Example: `echo /lib/somelib.so > /etc/ld.so.preload`

Although LD\_PRELOAD is a common vendor of attack for rootkits, it isn't always used maliciously.

Why LD\_PRELOAD?

LD\_PRELOAD loads a shared object before any other library, thus easy to implement. Though LD\_PRELOAD is an easy attack vendor, it does present some drawbacks; some LD\_PRELOAD kits are simple to remove, for example, Azazel can be removed simply by moving /etc/ld.so.preload to a different location. This occurs because Azazel isn't actually checking the rename() call for possible calls to any protected files, and it doesn't protect /etc/ld.so.preload, or infact it's shared object file, so that's one way a flaw can occur in an LD\_PRELOAD rootkit. I'll leave the github repositories for Azazel and Jynx2 at the bottom of this thread.

General background on rootkits

At the lowest level, we have ring0 rootkits, which have access to virtual kernel memory and all hardware drivers. ring0 rootkits are loaded in various ways, the most common way of loading ring0 rootkits are either:

a. to compile as a kernel object file and insert the object as a kernel module via insmod or modprobe

or

b. to insert the kernel object file directly into the /dev/kmem device. this is a better method as some Linux kernels don't enable the setting which allows the use of kernel modules. many mobile android rootkits use this method of delivery

Highest on the spectrum are ring3 rootkits. ring3 rootkits operate in the userspace of the machine. Since they do in fact use physical memory and not virtual kernel memory, they are much more noisy than a kernelspace (ring0) rootkit, although a rootkit is written in userspace (ring3), this does not mean it is automatically inferior to the attributes of a ring0 rootkit. If a userspace rootkit is written efficiently enough, it will prevail to be just as powerful as a kernelspace rootkit. We'll be writing a userspace rootkit in this tutorial.

How are these libraries installed?

This is the install script for panther - a userspace rootkit I wrote a long time ago - and procedurally

go through each step.

[img]http://a.pomf.se/zaafri.png[/img]

script opens

script identifies which uid is executing the script, if = 0 continue, else exit

dynamically compile with all warnings, library inclusion allowed, don't link the output and output the bin as panther.o

compile the dynamic bin as a shared library, local lib inclusion allowed, passing -soname to the linker, linking libvimcore.so, linking the dl lib, outputting as libvimcore.so, and target bin panther.o

skip 2 lines

installing the shared library under read/write/execute (owner) + read/execute (other users) (755) into the /lib/ directory

pointing LD\_PRELOAD to the newly installed library so that /lib/libvimcore.so loads before any other library

script notifies successful injection, removes compilation results and exits

This bash script is a replication of Azazel's Makefile.

Shut up already, how do I actually write my own libcall

First, you need to know what library you need to overwrite, in this case, let's make any file or directory starting with "\_\_" in its name unable to be removed by rm or rmdir. To do this, we need to find out what libcalls rm and rmdir reference, to do this, we'll use the tool "ltrace", similar to strace but it traces library calls and not system calls. Syntax of ltrace is simple, "ltrace /path/to/file optional arguments", and any calls are printed to the screen. Let's find out what libcalls rmdir and rm call.

rmdir

[img]http://a.pomf.se/kctorh.png[/img]

Alright, so this is a pretty simple screenshot. rmdir() is explicitly called as a libcall, after the getopt\_long() call. We already know what we need to overwrite. We can also see from this screenshot that rmdir() is of data type integer - as it returned a value of 0 - and it takes argument of char (char const \* to be exact). Let's look at the man 2 page for rmdir so that we can see rmdir() in more detail.

What we're interested in is SYNOPSIS, DESCRIPTION, and RETURN VALUE.

In SYNOPSIS, we can see the function prototype for the rmdir() call, which is  
Code int rmdir(const char \*pathname);

What we need to do is overwrite that call so that if the argument pathname does in fact contain our protected string, the return value of the call is malformed to return an error, so that the call does not continue and the pathname is not removed.

Alright, that's the rmdir function understood, let's move on to rm itself.

rm

We need to do the same thing we done the last time for `rmdir()`, we need to `ltrace` the binary and remove a temporary file.

[img]http://a.pomf.se/mrsteg.png[/img]

Alright, granted, `rm`'s `ltrace` output is more complex in comparison to `rmdir`, but it is always easy to spot what you're looking for. Look on the right of the output and have a look for calls that have a return value of 0. A return value of 0 indicates a successful operation. Find return values of 0 and look back to the left and see what call is returning that value.

The call we should be particularly interested in is `unlinkat()`, the purpose of this call is literally in the name. Again, we can see from the `ltrace` output what kind of data type `unlinkat()` is, as it returns a value of 0 for success. And again, let's use `man` to go into deeper detail.

We can see the function prototype again, and it's arguments, which is:

```
Code int unlinkat(int dirfd, const char *pathname, int flags);
```

Again, just like `rmdir()`, we need to check if the second argument - `pathname` - contains our protected string, and if it does, return a malformed value so that the call does not continue. If `pathname` does NOT contain our protected string, it will call the regular `lib` and allow the call to continue.

Getting into the good stuff

To overwrite a `libcall`, you need to create C source file, define `_GNU_SOURCE` so that we gain access to non-standard GNU/Linux functions, include any includes mentioned in the `man` of your target `libcall(s)` and copy the prototype(s) on the `man` page too. It's simple, here's an example.

[img]http://a.pomf.se/ozscyg.png[/img]

You could say the `libcall` has been overwritten now, but if you compile and load this shared library, none of your `rm` or `rmdir` shell binaries will work because the calls are now empty, that's not what we want. In order to call the old, original, `libcall`, we need to use the dynamic linking interface, `dl`, this library will be linked in compilation in the installation script. Specifically, we'll use the `dlsym()` function from the `dl` library. Read more about `dl` here: <http://linux.die.net/man/3/dlsym>  
Using `dlsym()` you can call the original library even after you overwrite it.

An example as follows.

[img]http://a.pomf.se/swftyh.png[/img]

Using `typeof()` automatically grabs the data type of target call and then creates it using that said data type. When you're returning the original call, you always need to return the original arguments too, unless of course you're returning the call if your protected string is found. Example:

```
Code typeof(somecall) *old_somecall = dlsym(RTLD_NEXT, "somecall");
if(strstr(someargument1, "some file"))
{
    return old_somecall(someargument0, "some fake file", someargument2);
}
```

```
return old_somecall(someargument0, someargument1, someargument2);
```

Now if you were to compile this and load the shared object using the LD\_PRELOAD environment variable, the old libcall would be overwritten and instead of doing nothing, it will return the old syscall - if you want to make sure that the new syscall is indeed being called, you can use printf to display a significant message when a file is removed.

Example:

```
Code typedef(somecall) *old_somecall = dlsym(RTLD_NEXT, "somecall");  
printf("somecall() was hooked and old_somecall was created\n");  
return old_somecall(arg0, arg1, arg2, arg3);
```

Okay, so you've overwritten a call and you're returning the old call into the new one, but this isn't very beneficial as to what we want, which is to remove the ability to remove files that start with "\_\_". To do this, we'll use strncmp(), which compares str1 to str2, checking if str1 starts with the contents of str2, which, if they do, returns 0 - true - and if they don't, returns 1 - false. You could also use strstr(), which compares str1 to str2, checking if str1 contains the general contents of str2, again, this function is boolean type return data. If using strncmp(), do not use strcmp(), not unless you actually want to have your new calls riddled with stack-based buffer overflow vulns.

[img]http://a.pomf.se/wjwhrf.png[/img]

Now if you compile and load this, all users, root and non-root, will be unable to remove any files starting with "\_\_".

Thanks for reading and I hope you liked reading.

If there are any misunderstandings or things you would like to ask about, please post here or private message me. I'm also selling a private ring0 Linux rootkit, private message me if you would like to enquire.

Azazel: <https://github.com/chokepoint/azazel>

Jynx2: <https://github.com/chokepoint/Jynx2>

**lolx45** - October 31, 2015, 10:08 pm

EXTREMELY useful tutorial! Why doesn't this have any replies?!

**SIM-1866423** - October 31, 2015, 11:31 pm

Finally some good tutorials in this section .. good job mate !