Hiding with a Linux Rootkit

Exploit Development rootkit, malware, reverseengineering

jordan9001 1 Dec '17

I have always wanted to dive a bit deeper into how to linux kernel internals work. To do this I figured a good starting point would be to create a small proof of concept rootkit! You can find the code for the rootkit on github https://github.com/jordan9001/superhide 375.

This is a very simple rootkit. All it does is hide files that begin with a certain prefix from being seen. You can still access those files/folders if you know where they are, but 'ls -a' or a file manager won't see them. It also hides itself from being listed with lsmod or by viewing /proc/modules.
##Looking at LS
So, my first task was to see how ls finds it's files. Whenever a program works with networking, the file system, or other system specific activities, it is going to have to go through the kernel. That means it will use a system call. You can see a table of linux 64 bit system calls here 42.

In order to find what system calls ls was using, I used a tool called strace 5. Strace will list system calls made by a program. When you run `strace ls` there will be a lot of noise having to do with the program linking, but if we scroll a bit down, we see the following lines.

```
openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
getdents(3, /* 11 entries */, 32768)     = 344
getdents(3, /* 0 entries */, 32768)      = 0
close(3)                                 = 0
```

Sweet. Looks like the system call we need to mess with is getdents 35. Ls is probably calling the libc function, readdir, but it all boils down to getdents eventually.

We will get back to getdents in a second. First let’s talk about how we are going to get into the kernel.
##Loadable Kernel Modules
On linux, when we want to run code in the kernel we can make a Loadable Kernel Module (LKM). I won’t spend too much explaining how they work, because others have 27 done so better than I can here 15. The idea is that we can load our own module into the kernel. Some useful kernel symbols will be exported by the kernel for us to be able to use, or we can grab them with the `kallsyms_lookup_name` function. Once in the kernel, we are going to want to intercept the getdents call, and make changes to the values it returns. We are also going to want to hide ourselves a bit so it is not readily apparent that our module is in the system.

So how do we hook getdents. How does the linux kernel respond to a system call?
##System Call Table
In the kernel, there is a table called the system call table. The system call number (rax’s value at the time of syscall) is the offset into that table for that handler. If we were on Windows, this table would be untouchable due to PatchGuard 21. On linux, for now, we can get away with it. *(It is noisy of us to mess with this table, and if this was more than a silly proof of concept, we would want to put our hook somewhere else.)*

The system call table is at the symbol `sys_call_table`, which in not a symbol that modern linux kernels will export for you to use. *(There really is not a legitimate reason for you to mess with it.)* So we have 4 options to find it:

1. Bruteforce search for it. This worked better on 32 bit, but is still doable if you are smart about it.
2. Find somewhere it is used. There are a few functions that use the ```sys_call_table``` symbol, and if we parse through them, we can find their reference.
3. Find it somewhere else. It is actually easy to find if you are not in the kernel.
4. Don't find it. This is the best *(but most involved)* option. Instead of hooking at the table, we could put our hook in at the handler.

For this simple module, we will go with option #3. #4 would be fun, and I may write about doing that in a separate post. As for #3, we only have to read and parse the /boot/System.map-$(uname -r) 11 file. We can do this when we insert ourselves into the kernel, and we will be sure we have the correct address. My code does this in my build_and_install.sh 21 file. I generate a header file that will be compiled with the LKM.

```
smap="/boot/System.map-$(uname -r)"
echo -e "#pragma once" &gt; ./sysgen.h
echo -e "#include <linux/fs.h>" >> ./sysgen.h
symbline=$(cat $smap | grep '\Wsys_call_table$')
set $symbline
echo -e "void** sys_call_table = (void**)0x$1;" >> ./sysgen.h
```

##The Hook
The system call table is read only. Fortunately, when we are in the kernel that doesn't mean much. In the kernel the CR0 register 16 is a control register that modifies how the processor operates. Bit 16 is the Write Protect flag, and if this flag is 0, then the CPU will let the kernel write to read-only pages. Linux provides us with two helper functions for messing with the CR0 register, write_cr0 and read_cr0.

In my code I turn off write protection by `write_cr0(read_cr0() & (~WRITE_PROTECT_FLAG));` and turn it back on with `write_cr0(read_cr0() | WRITE_PROTECT_FLAG);` where `#define WRITE_PROTECT_FLAG (1<<16)`.

I save the current entry for the getdents handler, so I can put it back when I remove my module. Then I write my new handler to the table, with `sys_call_table[GETDENTS_SYSCALL_NUM] = sys_getdents_new;`.

The sys_getdents_new function simply runs the original handler, and then inspects its results, removing any entries that start with our special prefix or have to do with our module.
##Hiding from /proc/modules
On linux the /proc/ file system serves as an interface between userspace and the kernel. They are not traditional files, but rather opening, writing, or reading from then invokes a handler in the kernel that will gather the required information. By hooking the read handler for the /proc/modules file, we can filter out any lines referring to our module. In order to do this, first we need to know where the /proc/modules file is registered in the kernel.

After a bit of sleuthing *(I searched the linux source on github for 'proc_create("modules")')* we find the following lines 23:

```
static int __init proc_modules_init(void)
{
    proc_create("modules", 0, NULL, &proc_modules_operations);
    return 0;
}
```

Sweet! So proc_modules_operations is a struct that contains a the handler functions. It isn't an exported symbol, but we are using the System.map file already, so we might as well just use it again.

Once it is imported, adding our hook is as simple as `proc_modules_operations->read = proc_modules_read_new;`. In `proc_modules_read_new` we run the original read, then take the result and filter out any lines that tell on us.
##Conclusion
And then we have made our first linux rootkit! From here we could improve by hooking outside of the system call table, and by hiding out module from the kernel as well.

I hope I explained everything well. I had a lot of fun making this, and if anyone has any questions or finds a bug in my code, feel free to ask! I would also love to hear suggestions for other cool rootkit tricks! I am new to the community, so feel free to give me pointers on my post.

18

created	last reply	6	6.1k	4	18	12
Dec '17	Jan '18	replies	views	users	likes	links

3

Sirius First 100 Dec '17

This was an incredible write-up, any plans for related material in the future?

2 Replies

oaktree Programmer and Part-Time Savage Dec '17

@Sirius how did you read that so fast!?!?!

Sirius First 100 Dec '17

I was already familiar with the basic concepts of the article so I didn't have to reread sections as often as I normally would have.

jordan9001 Sirius Dec '17

I hope so! I want to write about hooking other places besides the system call table, but I need to find/make a small instruction-length decoder for x86_64 in c first. If you know of any, let me know!

Sirius First 100 Dec '17

Unfortunately not off the top of my head, most of my knowledge is fairly basic. Which is why I'm looking forward to future publications from your account! 😃

2 MONTHS LATER

CLOSED JAN 21, '18

This topic was automatically closed after 30 days. New replies are no longer allowed.

Reply

Suggested Topics

Topic	Replies	Activity
Windows 7 after the Supportend **Exploit Development** windows	13	4d
Stack based buffer overflows… still relevant? **Beginner Guides**	7	Feb 22
HackTheBox Write-Up - Access **Hackthebox Writeups**	18	May 28
[Pwnable/Code Auditing] NoREpls - Part 7 **Challenges** reverseengineering, windows, exploitation, codeauditing	2	Jan 3
Conference Meetups **Social** networking	19	8h

Want to read more? Browse other topics in Exploit Developm… or view latest topics.