CHAPTER
25

Unix Kernel Overflows

In this chapter, we explore kernel-level vulnerabilities and the development of robust, reliable exploits for Unix kernels. A few generic problems in various kernels, which could lead to exploitable conditions, will be identified, and we present several examples from known bugs. After familiarizing you with various types of kernel vulnerabilities, we advance the chapter by focusing on two exploits that were found in OpenBSD and Solaris operating systems during the initial research conducted for this chapter.

The vulnerabilities we discuss result in kernel-level access to OS resources in all versions of OpenBSD and Solaris. Kernel-level access has the rather serious consequence of easy privilege escalation, and consequently, the total compromise of any type of kernel-level security enforcements such as chroot, systrace, and any other commercial products that provide B1-trusted OS capabilities. We will also question OpenBSD's proactive security and its failure against kernel-level exploits. This will hopefully give you the motivation and spirit to target other supposedly secure-from-the-ground-up operating systems.

Kernel Vulnerability Types

Many functions and bad coding practices exist that can lead to exploitable conditions in kernel land. We will go over these weaknesses and provide examples from various kernels, giving hints about what to look for when conducting

audits. Dawson Engler's excellent paper and audit "Using Programmer-Written Compiler Extensions to Catch Security Holes" (`www.stanford.edu/~engler/sp-ieee-02.ps`) provides perfect examples of what to look for while hunting for kernel-land vulnerabilities.

Although many possible bad coding practices have been identified specific to kernel-level vulnerabilities, some potentially dangerous functions have still been missed even under rigorous code audits. OpenBSD's kernel stack overflow, presented in this chapter, falls into the category of a not-commonly-audited function. Kernel land contains potentially dangerous functions that can be the source of overflows, similar to the user-land APIs `strcpy` and `memcpy`.

These functions and various logic mistakes can be abstracted as follows:

- Signed integer problems
  - `buf[user_controlled_index]` vulnerabilities
  - `copyin/copyout` functions
- Integer overflows
  - `malloc/free` functions
  - `copyin/copyout` functions
  - integer arithmetic problems
- Buffer overflows (stack/heap)
  - `copyin` and several other similar functions
  - read/write from v-node to kernel buffer
- Format string overflows
  - `log`, `print` functions
- Design errors
  - `modload`, `ptrace`

Let's look at some publicly disclosed kernel-level vulnerabilities and work through various exploitation problems with real-life examples. Two OpenBSD kernel overflows (presented in *Phrack* 60, Article 0x6), one FreeBSD kernel information leak, and a Solaris design error are presented as case studies.

```
2.1 - OpenBSD select() kernel stack buffer overflow

sys_select(p, v, retval)
        register struct proc *p;
        void *v;
        register_t *retval;
{
        register struct sys_select_args /* {
                syscallarg(int) nd;
```

```
                syscallarg(fd_set *) in;
                syscallarg(fd_set *) ou;
                syscallarg(fd_set *) ex;
                syscallarg(struct timeval *) tv;
        } */ *uap = v;
        fd_set bits[6], *pibits[3], *pobits[3];
        struct timeval atv;
        int s, ncoll, error = 0, timo;
        u_int ni;

[1]     if (SCARG(uap, nd) > p->p_fd->fd_nfiles) {
                /* forgiving; slightly wrong */
                SCARG(uap, nd) = p->p_fd->fd_nfiles;
        }
[2]     ni = howmany(SCARG(uap, nd), NFDBITS) * sizeof(fd_mask);
[3]     if (SCARG(uap, nd) > FD_SETSIZE) {

        [deleted]

#define getbits(name, x)
[4]   if (SCARG(uap, name) && (error = copyin((caddr_t)SCARG(uap, name),
            (caddr_t)pibits[x], ni)))
                goto done;
[5]     getbits(in, 0);
        getbits(ou, 1);
        getbits(ex, 2);
#undef  getbits

     [deleted]
```

In order to make sense out of the selected syscall code, we need to extract the SCARG macro from the header files.

```
sys/systm.h:114
...
#if     BYTE_ORDER == BIG_ENDIAN
#define SCARG(p, k)     ((p)->k.be.datum)       /* get arg from args
pointer */
#elif   BYTE_ORDER == LITTLE_ENDIAN
#define SCARG(p, k)     ((p)->k.le.datum)       /* get arg from args
pointer */

sys/syscallarg.h: line 14

#define syscallarg(x)
        union {
                register_t pad;
                struct { x datum; } le;
                struct {
```

```
                        int8_t pad[ (sizeof (register_t) < sizeof (x))
                                ? 0
                                : sizeof (register_t) - sizeof (x)];
                        x datum;
                } be;
        }
```

SCARG() is a macro that retrieves the members of the struct sys_XXX_args structures (XXX representing the system call name), which are storage entities for system call related data. Access to the members of these structures is performed via SCARG() in order to preserve alignment along CPU register size boundaries, so that memory accesses will be faster and more efficient. The system call must declare incoming arguments as follows in order to make use of the SCARG() macro. The following declaration is for the incoming arguments structure of the select() system call:

```
sys/syscallarg.h: line 404

struct sys_select_args {
[6]     syscallarg(int) nd;
        syscallarg(fd_set *) in;
        syscallarg(fd_set *) ou;
        syscallarg(fd_set *) ex;
        syscallarg(struct timeval *) tv;
};
```

This specific vulnerability can be described as an insufficient check on the nd argument (you can find the exact line of code labeled [6], in the code example), which is used to calculate the length parameter for user-land-to-kernel-land copy operations.

Although there is a check [1] on the nd argument (nd represents the highest numbered descriptor plus one in any of the fd_sets), which is checked against the p->p_fd->fd_nfiles (the number of open descriptors that the process is holding). This check is inadequate. nd is declared as signed [6], so it can be supplied as negative; therefore, the greater-than check will be evaded [1]. Eventually nd is used by the howmany() macro [2] in order to calculate the length argument for the copyin operation ni.

```
#define howmany(x, y)   (((x)+((y)-1))/(y))

ni = ((nd + (NFDBITS-1)) / NFDBITS)  * sizeof(fd_mask);
ni = ((nd + (32 - 1)) / 32) * 4
```

Calculation of ni is followed by another check on the nd argument [3].

This check is also passed, because OpenBSD developers consistently forget about the signedness checks on the nd argument. Check [3] is done to determine

whether the space allocated on the stack is sufficient for the `copyin` operations following, and if not, then sufficient heap space will be allocated.

Given the inadequacy of the signed check, we'll pass check `[3]` and continue using stack space. Finally, the `getbits() [4, 5]` macro is defined and called to retrieve user-supplied `fd_sets` (`readfds`, `writefds`, `exceptfds`—these arrays contain the descriptors to be tested for *ready for reading*, *ready for writing*, or *have an exceptional condition pending*). Obviously if the `nd` argument is supplied as a negative integer, the `copyin` operation (within the `getbits`) will overwrite chunks of kernel memory, which could lead to code execution if certain kernel overflow tricks are used.

Eventually, with all pieces tied together, this vulnerability translates into the following pseudo code:

```
vuln_func(int user_number, char *user_buffer) {

char stack_buf[1024];

if( user_number > sizeof(stack_buf) )
       goto error;

copyin(stack_buf, user_buf, user_number);
/* copyin is somewhat the kernel land equivalent of memcpy */

}


2.2 - OpenBSD setitimer() kernel memory overwrite

sys_setitimer(p, v, retval)
        struct proc *p;
        register void *v;
        register_t *retval;
{
        register struct sys_setitimer_args /* {
[1]             syscallarg(u_int) which;
                syscallarg(struct itimerval *) itv;
                syscallarg(struct itimerval *) oitv;
        } */ *uap = v;
        struct itimerval aitv;
        register const struct itimerval *itvp;
        int s, error;
        int timo;

[2]     if (SCARG(uap, which) > ITIMER_PROF)
                return (EINVAL);
[deleted]

[3]             p->p_stats->p_timer[SCARG(uap, which)] = aitv;
        }
```

```
        splx(s);
        return (0);
}
```

This vulnerability can be categorized as a kernel-memory overwrite due to insufficient checks on a user-controlled index integer that references an entry in an array of kernel structure. The integer that represents the index was used to under-reference the structure, thus writing into arbitrary locations within the kernel memory. This was made possible due to a signedness vulnerability in validating the index against a fixed-sized integer (which represents the largest index number allowed).

This index number is the `which [1]` argument to the system call; this is falsely claimed as an unsigned integer in the comment text block (hint, the `/* */`) `[1]`. The `which` argument is actually declared as a signed integer in the `sys/syscallargs.h` line 369 (checked in OpenBSD 3.1), thus making it possible for user-land applications to supply a negative value, which will lead to evading the validation checks done by `[2]`. Eventually, the kernel will copy a user-supplied structure into kernel memory using the `which` argument as the index into a buffer of structures `[3]`. At this stage, a carefully calculated negative `which` integer makes it possible to write into the credential structure of the process or the user, thus elevating privileges.

This vulnerability can be translated into the following pseudo code to illustrate a possible vulnerable pattern in various kernels:

```
vuln_func(int user_index, struct userdata *uptr) {

if( user_index > FIXED_LIMIT )
       goto error;

kbuf[user_index] = *uptr;

}


2.3 - FreeBSD accept() kernel memory infoleak

int
accept(td, uap)
        struct thread *td;
        struct accept_args *uap;
{

[1]     return (accept1(td, uap, 0));
}

static int
accept1(td, uap, compat)
        struct thread *td;
```

```
[2]     register struct accept_args /* {
                int     s;
                caddr_t name;
                int     *anamelen;
        } */ *uap;
        int compat;
{
        struct filedesc *fdp;
        struct file *nfp = NULL;
        struct sockaddr *sa;
[3]     int namelen, error, s;
        struct socket *head, *so;
        int fd;
        u_int fflag;

        mtx_lock(&Giant);
        fdp = td->td_proc->p_fd;
        if (uap->name) {
[4]             error = copyin(uap->anamelen, &namelen, sizeof (namelen));
                if(error)
                        goto done2;
        }
[deleted]
        error = soaccept(so, &sa);

[deleted]
        if (uap->name) {
                /* check sa_len before it is destroyed */
[5]             if (namelen > sa->sa_len)
                        namelen = sa->sa_len;
[deleted]

[6]             error = copyout(sa, uap->name, (u_int)namelen);

[deleted]

}
```

The fact that FreeBSD accepts system call vulnerability is a signedness issue that leads to a kernel memory information leakage condition. The `accept()` system call is directly dispatched to the `accept1()` function `[1]` with only an additional zero argument. The arguments from user land are packed into the `accept_args` structure `[2]` which contains:

- An integer that represents the socket
- A pointer to a `sockaddr` structure
- A pointer to signed integer that represents the size of the `sockaddr` structure

Initially [4], the accept1() function copies the value of the user-supplied size argument into a variable called namelen [3]. It is important to note that this is a signed integer and can represent negative values. Subsequently the accept1() function performs an extensive number of socket-related operations to set the proper state of the socket. This places the socket in a waiting-for-new-connections state. Finally the soaccept() function fills out a new sockaddr structure with the address of the connecting entity [5], which eventually will be copied out to user land.

The size of the new sockaddr structure is compared to the size of the user-supplied size argument [5], ensuring that there is enough space in the user-land buffer to hold the structure. Unfortunately, this check is evaded, and attackers can supply a negative value for the namelen integer and bypass this bigger-than comparison. This evasion on the size check leads to having a large chunk of kernel memory copied to the user-land buffer.

This vulnerability can be translated into the following pseudo code to illustrate a potential vulnerable pattern in various kernels:

```
struct userdata {
    int len;     /* signed! */
    char *data;
};

vuln_func(struct userdata *uptr) {

struct kerneldata *kptr;

internal_func(kptr); /* fill-in kptr */

if( uptr->len > kptr->len )
     uptr->len = kptr->len;

copyout(kptr, uptr->data, uptr->len);

}


Solaris priocntl() directory traversal

/*
 * The priocntl system call.
 */
long
priocntlsys(int pc_version, procset_t *psp, int cmd, caddr_t arg)
{
 [deleted]

        switch (cmd) {
```

```
[1]     case PC_GETCID:
...
[2]  if (copyin(arg, (caddr_t)&pcinfo, sizeof (pcinfo)))
...
                error =
[3]                     scheduler_load(pcinfo.pc_clname,
&sclass[pcinfo.pc_cid]);
 [deleted]
}

int
scheduler_load(char *clname, sclass_t *clp)
{
 [deleted]
[4]                     if (modload("sched", clname) == -1)
                                return (EINVAL);
                        rw_enter(clp->cl_lock, RW_READER);
 [deleted]
}
```

The Solaris `priocntl()` vulnerability is a perfect example of the design error vulnerability genre. Without going into unnecessary detail, let's examine how this vulnerability is possible. `priocntl` is a system call that gives users control over the scheduling of light-weight processes (LWPs), which can mean either a single LWP of a process or the process itself. There are several supported scheduling classes available in a typical Solaris installation:

- The real-time class
- The time-sharing class
- The fair-share class
- The fixed-priority class

All these scheduling classes are implemented as dynamically loadable kernel modules. They are loaded by the `prionctl` system call based on user-land requests. This system call typically takes two arguments from user-land `cmd` and a pointer to a structure `arg`. The vulnerability resides in the `PC_GETCID` `cmd` type that is handled by the case statement `[1]`. The displacement of the `cmd` argument is followed by copying the user-supplied `arg` pointer into the relevant scheduling class-related structure `[2]`. The newly copied structure contains all information regarding the scheduling class, as we can see from this code fragment:

```
typedef struct pcinfo {
        id_t    pc_cid;                 /* class id */
        char    pc_clname[PC_CLNMSZ];   /* class name */
        int     pc_clinfo[PC_CLINFOSZ]; /* class information */
} pcinfo_t;
```

The interesting piece of this particular structure is the `pc_clname` argument. This is the scheduling class's name as well as its relative pathname. If we want to use the scheduling class name `myclass`, the `prionctl` system call will search the `/kernel/sched/` and `/usr/kernel/sched/` directories for the `mycall` kernel module. If it finds it, the module will be loaded. All these steps are orchestrated by the `[3] scheduler_load` and `[4] modload` functions. As previously discussed, the scheduler class name is a relative pathname; it is appended to the predefined pathname in which all kernel modules reside. When this appending behavior exists without a check for directory traversal conditions, it is possible to supply a class name with `../` in its name. Now, we can take advantage of this vulnerability and load arbitrary kernel modules from various locations on the filesystem. For example, a `pc_clname` argument such as `../../tmp/mymod` will be translated into `/kernel/sched/../../tmp/mymod`, thus allowing a malicious kernel module to be loaded into memory.

Although several other interesting design errors were identified in various kernels (`ptrace`, `vfork`, and so on), we believe that this particular flaw is an excellent example of a kernel vulnerability. At time of writing, this vulnerability could be located and exploited in a similar fashion in all current versions of the Solaris operating system. The `priocntl` bug is an important discovery. It leads us to look into the `modload` interface, which allows us to discover additional exploitable kernel-level weaknesses. We recommend that you look into previously found kernel vulnerabilities and try to translate them into pseudo code, or some sort of bug primitive, which will eventually help you identify and exploit your own `0day`.

0day Kernel Vulnerabilities

We will now present a few of the new kernel-level vulnerabilities in major operating systems that existed at the time this book was written. These vulnerabilities represent a few new techniques for discovering and exploiting vulnerabilities that have never before been published.

OpenBSD exec_ibcs2_coff_prep_zmagic() Stack Overflow

Let's begin by looking at the interface that has escaped so many auditing eyes:

```
int
vn_rdwr(rw, vp, base, len, offset, segflg, ioflg, cred, aresid, p)
[1]     enum uio_rw rw;
[2]     struct vnode *vp;
[3]     caddr_t base;
[4]     int len;
```

```
        off_t offset;
        enum uio_seg segflg;
        int ioflg;
        struct ucred *cred;
        size_t *aresid;
        struct proc *p;
{
...
```

The `vn_rdwr()` function reads and writes data to or from an object represented by a v-node. A v-node represents access to an object within a virtual filesystem. It is created or used to reference a file by pathname.

You may be thinking, why delve into this filesystem code when looking for kernel vulnerabilities? First, this vulnerability requires that we read from a file and store it in a kernel stack buffer. It makes the slight mistake of trusting the user-supplied size argument. This vulnerability has not been identified in any of the systematic audits conducted on the OpenBSD operating system, probably because the auditors were not aware of any possible problems with the `vn_rdwr()` interface. We urge you to look into the kernel API and try to identify what could be the next big class of kernel vulnerabilities, instead of repeatedly looking for the familiar copyin/malloc problems.

`vn_rdwr()` has four significant arguments that we need to know about; the others we can safely ignore. The first is `rw enum`. The `rw` arguments represent operation mode. It will read from, or conversely, write to, a virtual node (v-node). Next is the `vp` pointer. It will point to the v-node of the file to read or write. The third argument to be aware of is the base pointer, which is a pointer to the kernel storage (stack, heap, and so on). Finally we have the `len` integer, or the size of the kernel storage pointed to by the `base` argument.

The `rw` argument `UIO_READ` means that `vn_rdwr` is used to read `len` bytes of a file and store it into the kernel storage `base`. `UIO_WRITE` writes `len` bytes to a file from the kernel buffer `base`. As the operation implies, `UIO_READ` can be a convenient source for overflows, because it is similar to a `copyin()` operation. `UIO_WRITE`, on the other hand, may lead to an information leak that is similar to various `copyout()` problems. As always, after identifying a potential problem and a possible new class of kernel-level security bug, you should use Cscope (a source code browser) on the entire kernel source tree. Alternatively, if the source code is not available, you could begin conducting binary audits with IDA Pro.

After briefly shifting through the `vn_rdwr` function in OpenBSD kernel, we found a humorous kernel bug, which existed in all versions of OpenBSD at the time this book was written. The only possible workaround is to custom compile a kernel, leaving out certain compatibility options. In the field, most people leave the compatibility options enabled, even though they compile custom

kernels. We should also remind you that `compat` options exist in the secure default installation.

The Vulnerability

The vulnerability exists in the `exec_ibcs2_coff_prep_zmagic()` function. Naturally, in order to understand the vulnerability you should first become familiar with the code:

```
/*
 * exec_ibcs2_coff_prep_zmagic(): Prepare a COFF ZMAGIC binary's exec package
 *
 * First, set the various offsets/lengths in the exec package.
 *
 * Then, mark the text image busy (so it can be demand paged) or error
 * out if this is not possible.  Finally, set up vmcmds for the
 * text, data, bss, and stack segments.
 */

int
exec_ibcs2_coff_prep_zmagic(p, epp, fp, ap)
        struct proc *p;
        struct exec_package *epp;
        struct coff_filehdr *fp;
        struct coff_aouthdr *ap;
{
        int error;
        u_long offset;
        long dsize, baddr, bsize;
[1]     struct coff_scnhdr sh;

        /* set up command for text segment */
[2a]    error = coff_find_section(p, epp->ep_vp, fp, &sh,
COFF_STYP_TEXT);

     [deleted]

        NEW_VMCMD(&epp->ep_vmcmds, vmcmd_map_readvn, epp->ep_tsize,
                  epp->ep_taddr, epp->ep_vp, offset,
                  VM_PROT_READ|VM_PROT_EXECUTE);

        /* set up command for data segment */
[2b]    error = coff_find_section(p, epp->ep_vp, fp, &sh,
COFF_STYP_DATA);

     [deleted]

        NEW_VMCMD(&epp->ep_vmcmds, vmcmd_map_readvn,
                  dsize, epp->ep_daddr, epp->ep_vp, offset,
                  VM_PROT_READ|VM_PROT_WRITE|VM_PROT_EXECUTE);
```

```
        /* set up command for bss segment */
     [deleted]

        /* load any shared libraries */
[2c]    error = coff_find_section(p, epp->ep_vp, fp, &sh, COFF_STYP_SHLIB);
        if (!error) {
                size_t resid;
                struct coff_slhdr *slhdr;
[3]             char buf[128], *bufp;   /* FIXME */
[4]             int len = sh.s_size, path_index, entry_len;

                /* DPRINTF(("COFF shlib size %d offset %d\n",
                         sh.s_size, sh.s_scnptr)); */

[5]             error = vn_rdwr(UIO_READ, epp->ep_vp, (caddr_t) buf,
                                len, sh.s_scnptr,
                                UIO_SYSSPACE, IO_NODELOCKED, p->p_ucred,
                                &resid, p);
```

The `exec_ibcs2_coff_prep_zmagic()` function is responsible for creating an execution environment for the `COFF ZMAGIC`-type binaries. It is called by the `exec_ibcs2_coff_makecmds()` function, which checks whether a given file is a `COFF`-formatted executable. It also checks for the magic number. This magic number will be further used to identify the specific handler responsible for setting up the virtual memory layout for the process. In `ZMAGIC`-type binaries, this handler will be the `exec_ibcs2_coff_prep_zmagic()` function. We should remind you that the entry point to reach these functions is the `execve` system call, which supports and emulates many executable types such as `ELF`, `COFF`, and other native executables from various Unix-based operating systems. The `exec_ibcs2_coff_prep_zmagic()` function can be reached and executed by crafting a `COFF` (type `ZMAGIC`) executable. In the following sections, we will create this type of executable, embedding our overflow vector into a malicious binary. We are getting ahead of ourselves here, however; first let's talk about the vulnerability.

The code path to the vulnerability is as follows:

```
user mode:

0x32a54 <execve>:       mov    $0x3b,%eax
0x32a59 <execve+5>:     int    $0x80

     |
     |
     V

kernel mode:
```

```
[ ISR and initial syscall handler skipped]

int
sys_execve(p, v, retval)
        register struct proc *p;
        void *v;
        register_t *retval;
{
     [deleted]
     if ((error = check_exec(p, &pack)) != 0) {
                goto freehdr;
        }
     [deleted]
}
```

Let's talk about the important structures in this code snippet. The `execsw` array stores multiple `execsw` structures that represent various executable types. The `check_exec()` function iterates through this array and calls the functions that are responsible for identifying the certain executable formats. The `es_check` is the function pointer that is filled with the address of the executable format verifier in every executable format handler.

```
struct execsw {
        u_int   es_hdrsz;               /* size of header for this format */
        exec_makecmds_fcn es_check;     /* function to check exec format */
};

...

struct execsw execsw[] = {
[deleted]
#ifdef _KERN_DO_ELF
        { sizeof(Elf32_Ehdr), exec_elf32_makecmds, },   /* elf binaries */
#endif
[deleted]
#ifdef COMPAT_IBCS2
        { COFF_HDR_SIZE, exec_ibcs2_coff_makecmds, },   /* coff binaries */
[deleted]


check_exec(p, epp)
        struct proc *p;
        struct exec_package *epp;
{
     [deleted]
          newerror = (*execsw[i].es_check)(p, epp);
```

Again, it is important that you follow what this code does. The COFF binary type will be identified by the COMPAT_IBCS2 element of execsw structure and that function (es_check=exec_ibcs2_coff_makecmds) will gradually dispatch ZMAGIC-type binaries to the exec_ibcs2_coff_prep_zmagic() function.

```
}

     |
     |
     V

int
exec_ibcs2_coff_makecmds(p, epp)
        struct proc *p;
        struct exec_package *epp;
{
     [deleted]
     if (COFF_BADMAG(fp))
                return ENOEXEC;
```

This macro checks whether the binary format is COFF, and if so, execution continues.

```
     [deleted]
        switch (ap->a_magic) {
     [deleted]
        case COFF_ZMAGIC:
                error = exec_ibcs2_coff_prep_zmagic(p, epp, fp, ap);
                break;
     [deleted]
}

     |
     |
     V

int
exec_ibcs2_coff_prep_zmagic(p, epp, fp, ap)
        struct proc *p;
        struct exec_package *epp;
        struct coff_filehdr *fp;
        struct coff_aouthdr *ap;
```

Let's walk through the function so that we understand what will eventually lead us to a stack-based buffer overflow. In [1], we see that coff_scnhdr defines the information regarding a section for a COFF binary (called the *section header*), and this structure is filled in by the coff_find_section() function [2a, 2b 2c] based on the queried section type. ZMAGIC COFF binaries are parsed

for COFF_STYP_TEXT (.text), COFF_STYP_DATA (.data), and COFF_STYP_SHLIB (shared library) section headers, respectively. During the execution flow, coff_find_section() is called several times. The coff_scnhdr structure is filled with the section header from the binary, and the section data is mapped into the process's virtual address space by the NEW_VMCMD macro.

Now, the header regarding the .text segment's section is read into sh (coff_scnhdr) [2a]. Various checks and calculations are performed, followed by the NEW_VMCMD macro to actually map the section into memory. Precise steps have been taken for the .data segment [2b], which will create another memory region. The third step reads in the section header [2c], representing all the linked shared libraries, and then maps them into the executable's address space one at a time. After the section header representing .shlib is read in [2c], the section's data is read in from the executable's v-node. Next, vn_rdwr() is called with the size gathered from section header [4] into a static stack buffer that is only 128 bytes [4]. This can result in typical buffer overflow. What is really happening here is that data is read into a static stack buffer based on user-supplied size and from user-supplied data.

Because we can construct a fake COFF binary with all the necessary section headers and most importantly, a.shlib section header, we can overflow this buffer. We need a size field greater than 128 bytes, which will lead us to smash the OpenBSD's stack and gain complete ring 0 (kernel mode) code execution of any user-supplied payload. Remember that we said there was some humor attached to this vulnerability? The humor behind this vulnerability is hidden at [3], where the local kernel storage char buf[128] is declared:

```
/* FIXME */
```

Not quite a cocktail party joke, but funny nevertheless. We hope OpenBSD developers finally do what they meant to do a long time ago.

Now that you have a solid understanding of the vulnerability, we will move on to a vulnerability in a closed source operating system. We will also demonstrate a few generic kernel exploitation techniques and shellcode.

Solaris vfs_getvfssw() Loadable Kernel Module Traversal Vulnerability

Once again, let's look directly at the vulnerable code and make sense of what it does before delving into the details of the vulnerability:

```
/*
 * Find a vfssw entry given a file system type name.
 * Try to autoload the filesystem if it's not found.
 * If it's installed, return the vfssw locked to prevent unloading.
 */
struct vfssw *
```

```
vfs_getvfssw(char *type)
{
        struct vfssw *vswp;
        char    *modname;
        int rval;

        RLOCK_VFSSW();
        if ((vswp = vfs_getvfsswbyname(type)) == NULL) {
                RUNLOCK_VFSSW();
                WLOCK_VFSSW();
                if ((vswp = vfs_getvfsswbyname(type)) == NULL) {
[1]                     if ((vswp = allocate_vfssw(type)) == NULL) {
                                WUNLOCK_VFSSW();
                                return (NULL);
                        }
                }
                WUNLOCK_VFSSW();
                RLOCK_VFSSW();
        }

[2]     modname = vfs_to_modname(type);

        /*
         * Try to load the filesystem.  Before calling modload(), we drop
         * our lock on the VFS switch table, and pick it up after the
         * module is loaded.  However, there is a potential race:  the
         * module could be unloaded after the call to modload() completes
         * but before we pick up the lock and drive on.  Therefore,
         * we keep reloading the module until we've loaded the module
         * _and_ we have the lock on the VFS switch table.
         */
        while (!VFS_INSTALLED(vswp)) {
                RUNLOCK_VFSSW();
                if (rootdir != NULL)
[3]                     rval = modload("fs", modname);

     [deleted]
}
```

The Solaris operating system has most of its kernel-related functionality implemented as kernel modules that are loaded on demand. Other than the core kernel functionality, most of the kernel services are implemented as dynamic kernel modules, including various filesystem types. When the kernel receives a service request for a filesystem that has not been previously loaded into kernel space, the kernel searches for a possible dynamic kernel module for that filesystem. It loads the module from one of the previously mentioned module directories, thus gaining the capability to serve the request. This particular vulnerability, much like the `priocntl` vulnerability, involves tricking

the operating system into loading a user-supplied kernel module (in this particular case, a module representing a filesystem), thus gaining full kernel execution rights.

The Solaris kernel keeps track of loaded filesystems with the Solaris filesystem `switch` table. Essentially, this table is an array of `vfssw_t` structures.

```
typedef struct vfssw {
        char            *vsw_name;      /* type name string */
        int             (*vsw_init)(struct vfssw *, int);
                                        /* init routine */
        struct vfsops   *vsw_vfsops;    /* filesystem operations vector
*/
        int             vsw_flag;       /* flags */
} vfssw_t;
```

The `vfs_getvfssw()` function traverses the `vfssw[]` array searching for a matching entry based on the `vsw_name` (which is the `type` char string passed to the function). If no matching entry is found, `vfs_getvfssw()` function first allocates a new entry point in the `vfssw[]` array `[1]` and then calls a translation function `[2]`, which basically does nothing more than parse the `type` argument for certain strings. This behavior is of no real interest when exploiting the vulnerability. Finally, it autoloads the filesystem by calling the infamous `modload` function `[3]`.

During our kernel audit, we found that two of the Solaris system calls use the `vfs_getvfssw()` function with a user-land supplied `type`. It will be translated into the module name to be loaded from either the `/kernel/fs/` directory or the `/usr/kernel/fs/` directory. Once again, `modload` interface can be attacked with simple directory traversal tricks that will give us kernel execution. `mount` and `sysfs` system calls have been identified and successfully exploited during our audits. (Exploitation of this vulnerability is presented in Chapter 26.) Let's now look at the two possible code paths that lead to the `vfs_getvfssw()` with user-controlled input.

The sysfs() System Call

The `sysfs()` syscall is one example of a code path to `vfs_getvfssw()` that will allow user-controlled input.

```
int
sysfs(int opcode, long a1, long a2)
{
        int error;

        switch (opcode) {
        case GETFSIND:
                error = sysfsind((char *)a1);
```

```
      [deleted]

      |
      |
      V

static int
sysfsind(char *fsname)
{
         /*
          * Translate fs identifier to an index into the vfssw structure.
          */
         struct vfssw *vswp;
         char fsbuf[FSTYPSZ];
         int retval;
         size_t len = 0;

         retval = copyinstr(fsname, fsbuf, FSTYPSZ, &len);

      [deleted]

         /*
          * Search the vfssw table for the fs identifier
          * and return the index.
          */
         if ((vswp = vfs_getvfssw(fsbuf)) != NULL) {

      [deleted]
```

The mount() System Call

The `mount()` syscall is another example of a code path to `vfs_getvfssw()` that will allow user-controlled input.

```
int
mount(char *spec, char *dir, int flags,
         char *fstype, char *dataptr, int datalen)
{

      [deleted]

         ua.spec = spec;
         ua.dir = dir;
         ua.flags = flags;
         ua.fstype = fstype;
         ua.dataptr = dataptr;
         ua.datalen = datalen;
```

```
      [deleted]

         error = domount(NULL, &ua, vp, CRED(), &vfsp);
      [deleted]

      |
      |
      V

int
domount(char *fsname, struct mounta *uap, vnode_t *vp, struct cred
*credp,
        struct vfs **vfspp)
{
      [deleted]

                           error = copyinstr(uap->fstype, name,
                                     FSTYPSZ, &n);
      [deleted]

                  if ((vswp = vfs_getvfssw(name)) == NULL) {
                          vn_vfsunlock(vp);
      [deleted]
}
```

We must admit that we did not check all possible kernel interfaces that use the `vfs_getvfssw()` function, but most likely this is all there is. You are encouraged to look into `modload()`-related problems, which might reveal still more exploitable interfaces.

Conclusion

In this chapter, we introduced methods with which to discover new vulnerabilities in two operating systems, OpenBSD and Solaris. Understanding kernel vulnerabilities is difficult; therefore, we have saved the actual exploitation for the next chapter. Move on only when you have a complete understanding of the concepts and vulnerabilities described in this chapter.

We hope you develop a feeling for certain types of kernel-level vulnerabilities by reading this chapter. We will now leave the exploit construction of the Solaris and OpenBSD vulnerabilities to Chapter 26. Turn the page for some more serious fun!