Unix kernel Auditing

Ilja van Sprundel <ilja@suresec.org>

Who am I ?

Ilja van Sprundel:

- Employed by Suresec Ltd.
- Breaks stuff for fun and profit
- Working with unix for a few years
- Intrigued by operating system internals

suresec
SECURING THE SOURCE

Agenda

- What is unix
- Kernel vs userland
- Why kernels ?
- Bugs
  - Buffer overflows
  - Signedness problems
  - Integer overflows
  - Time of check time of use (race conditions)
  - Reference counter overflows
  - Information leaks
  - PANIC !
  - Userland interaction bugs
  - Dereferencing userdata directly
- Fuzzing the kernel:
  - What is fuzzing
  - Syscall argument fuzzing
  - More detailed argument fuzzing
  - Binary file fuzzing
  - Some comments on kernel fuzzing
- comments

What is unix

- An operating system based on a set of standards which define it's behavior
- Resources (memory, disk access, ...)
- Processes
- Threads
- Multi-user
- Mostly written in C, some in C++ (or parts of it), small parts written in assembler

Kernel vs user-space

- Privilege levels:
  - Most hardware supports different privilege levels
  - Level n can do less then n-1
  - Kernel usually runs at the lowest level (it needs it for various hardware reasons)
  - Userland applications usually run at the highest level.

Kernel vs user-space II

- The kernel provides services to a userland application:
  - Requesting memory
  - Reading a file
  - Opening a file
  - Making a network connection
  - Spawning a new program
  - Making a new process
  - Many many more ...
- Usually needed because hardware interaction is required or kernel data needs to be queried.

Kernel vs user-space III

- Communication between the kernel and the user-space applications:
  - System calls are used
  - Usually triggered by a software interrupt
  - Each system call has a number
  - This number is usually in a register or pushed on the stack
  - Referred to as a mode switch (changing from user mode to kernel mode)
  - List of system calls can usually be found in /usr/include/sys/syscall.h

Kernel vs user-space IV

- On x86 linux calling the exit() system call looks like:

```
movl $NR_EXIT, %eax
# move exit() nr
# in eax register
int $0x80
# software
# interrupt 0x80
```

```
void main (void) {
...
exit(0);
}
```

userspace

kernel

```
void exit (int n)
{
        ...
        exit_thread();
        ...
}
```

Kernel vs user-space V

- The stack:
  - A place to temporarily store data.
  - Each user process has a stack.
  - For each user process there is a kernel stack
  - A kernel stack is usually very limited in size (2 or 3 pages) and most data is stored on the heap somewhere.

Kernel vs user-space VI

- Copying data from and to user-space from and to kernel-space needs to be handled with special care:
  - Copyin(), copyout(), copy_from_user(), ...
  - Verify that the address used exists
  - Verify that it is indeed a userpage (and not a kernel page)
  - Verify that it's readable or writable
  - Some things are very unix specific
    - Some discard negative values, no copy will be done (Mac OS X, AIX)
    - Some will pad with 0bytes (linux)
    - Some will just stop copying when an unmapped page is hit (Most bsd's)
- Most unices have 150-500 system calls
- They have +1000 inputs and outputs

suresec SECURING THE SOURCE

Why kernels ?

- Fun to play with
- Hard to strip down a kernel unlike userland applications
- Huge programs, so extremely error prone
- More and more important as people are deploying all sorts of security solutions (mostly) designed to protect userland (grsec, PaX, execshield, argus, ssp, ...)

Bugs: Buffer overflow

- Known for a VERY long time
- Still an issue
- They also exist in the kernel.
- A deadly attack vector
- Allows execution of custom code inside the kernel (when exploited properly) !

Bugs: buffer overflow II

- Stack-based buffer overflow:
  - 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

Bugs: buffer overflow III

- Stack-based buffer overflow:
  - 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

Bugs: buffer overflow IV

- Stack-based buffer overflow:
  – 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 than some assembly code
  – In a lot of cases there are restricted characters (such as '\x00')
  – work around these restricted characters

```
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)
    }
```

Bugs: buffer overflow (example)

```
asmlinkage int solaris_sendmsg(int fd, struct sol_nmsghdr *user_msg, unsigned user_flags) {
    unsigned char ctl[sizeof(struct cmsghdr) + 20];
    unsigned char *ctl_buf = ctl;
    struct msghdr kern_msg;
    ...
    if(msghdr_from_user32_to_kern(&kern_msg, user_msg))
    ...
    if(kern_msg.msg_controllen) {
        struct sol_cmsghdr *ucmsg = (struct sol_cmsghdr *)kern_msg.msg_control;
        unsigned long *kcmsg;
        __kernel_size_t32 cmlen;

        if(kern_msg.msg_controllen > sizeof(ctl) &&
          kern_msg.msg_controllen <= 256) {
            err = -ENOBUFS;
            ctl_buf = kmalloc(kern_msg.msg_controllen, GFP_KERNEL);
            if(!ctl_buf)
                goto out_freeiov;
        }
        ...
        if(copy_from_user(kcmsg, &ucmsg->cmsg_level,kern_msg.msg_controllen - sizeof(__kernel_size_t32)))
```

Linux sparc64 code

Copy user_msg struct to the kernel

Buffer-overflow

More buffer overflow stuff

- Shellcode
  - Usually not needed, just write it in c, compile exploit and jump to userland
    - Doesn't work on Mac OS X because there is a full address space split between userspace and kernelspace
  - Calling execve() doesn't work
  - What does work:
    - Find process structure
    - Overwrite uid/gid
    - Overwrite additional stuff (if needed)
  - Most of the time filter restrictions don't apply at all (for the shellcode)

No shellcode needed !

```
/* stolen from linux-2.4.29/include/asm-i386/current.h */
struct task_struct * get_current(void)
{
        struct task_struct *current;
        __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
        return current;
}

int kcode(void) {
        struct task_struct *p;
        p = get_current();
        p = p->p_pptr;
        p->uid = p->euid = p->fsuid = p->suid = 0;
        return -3;
}
```

Even more buffer overflow stuff

- Cleaning up after your shellcode is done
  - No need on linux, just let it oops (do make sure that you got rid of locks)
  - In most cases on most unices you can call something like schedule() in a loop (also get rid of locks)
  - Fix up the stack/heap/whatever you broke and jump back wherever you need to jump
- The last approach is for the not-so-lazy

Even more buffer overflow stuff

- Stack based buffer overflows are pretty much like those in userland.
- Heap based overflows
  - Overwrite memory management structures
    - Can be annoying, in most unices the heap meta-data and real data are separated
  - Carefully control what's on the heap
    - Use all sorts of info leaks (/proc/slabinfo, ...) to figure out what's where on the heap
    - Get something with a function pointer allocated right after the chunk you'll overflow
    - Can be fairly reliable (depends on the bug, system load, unix type, ...)

Bugs: Signedness problems

- Rather popular since a few years
- Usually important when comparing 2 signed values (can be both positive and negative)
- Potentially a lot of such bugs in unix kernels.
- illustration:

```
int somesyscall(void *data, int len) {
        char buf[128];
        if (len > 128)
                return(-ETOOLONG);
        if ( copyin(data, buf, len) ) {
                return(-EFAULT);
        }
        /* do something with the data
*/
        return(0);
}
```

Bugs: Signedness problems (example)

Linux bluetooth code

```
static int bluez_sock_create(struct socket *sock, int
proto)
{
        if (proto >= BLUEZ_MAX_PROTO)
                return -EINVAL;
        ...
        return bluez_proto[proto]->create(sock, proto);
}
```

Compare 2 signed valued (proto can be negative

Negative Indexing !

Demonstration

How this bug got fixed

- We mailed Marcel holtmann (maintainer of the linux bluetooth stack)
- He mailed us back about 20 minutes later with a fix
- A little while later it got committed to bitkeeper.
- It can't happen much faster then this !
- Kudos to Marcel !

Bugs: Integer overflow

- Related to signedness issues (they both misuse integers)
- An integer has a limited domain
- An unsigned 32 bit integer can represent numbers from 0 to 2**32-1
- When trying to put more into it (additions, multiplication, substraction (underflow)) the integer will wrap around and start from 0 again
- In kernel-space this can cause a lot of problems when calculating buffer space for dynamic memory

```
int main(void) {
        u_int32 a = ~0 + 1;
        printf("a: %u\n", a);
}
```

prints 0 instead of the expected 4294967296

Bugs: Integer overflow (example)

Mac OS X(FreeBSD, OpenBSD)

```
static int
vfs_hang_addrlist(struct mount *mp, struct netexport *nep,
struct export_args *argp) {
        register struct netcred *np;
        register int i;
        struct sockaddr *saddr;
        ...
        i = sizeof(struct netcred) + argp->ex_addrlen + argp->ex_masklen;
        MALLOC(np, struct netcred *, i, M_NETADDR, M_WAITOK);
        bzero((caddr_t)np, i);
        saddr = (struct sockaddr *)(np + 1);
        if (error = copyin(argp->ex_addr, (caddr_t)saddr, argp->ex_addrlen))
                goto out;
}
```

Integer overflow

Heap overflow

Not enough memory allocated due to integer overflow

Bugs: Time Of Check Time Of Use

- Bug occurs when some state is checked at time x – n
- And used at time n assuming the check done at time x - n is still valid
- When proper measures (locks, flags, reference, ...) are not in place this is a big problem in kernels.
- Sometimes very hard races to win (have to find a way to get the kernel to block or schedule so it switches to another process.)

Bugs: Time Of Check Time Of Use (example)

Linux 2.4.x ia64 and X86_64

```
asmlinkage long sys32_execve (char *filename,
        unsigned int argv, unsigned int envp, ...) {
        ...
        char **av, **ae;
        int na, ne, len;
        long r;
        na = nargs(argv, NULL);
        ne = nargs(envp, NULL);
        len = (na + ne + 2) * sizeof(*av);
        av = kmalloc(len, GFP_KERNEL);
        ae = av + na + 1;
        av[na] = NULL;
        ae[ne] = NULL;
        r = nargs(argv, av);
        if (r < 0)
                goto out;
        r = nargs(envp, ae);
        if (r < 0)
                goto out;
        ...
```

Count number of arguments

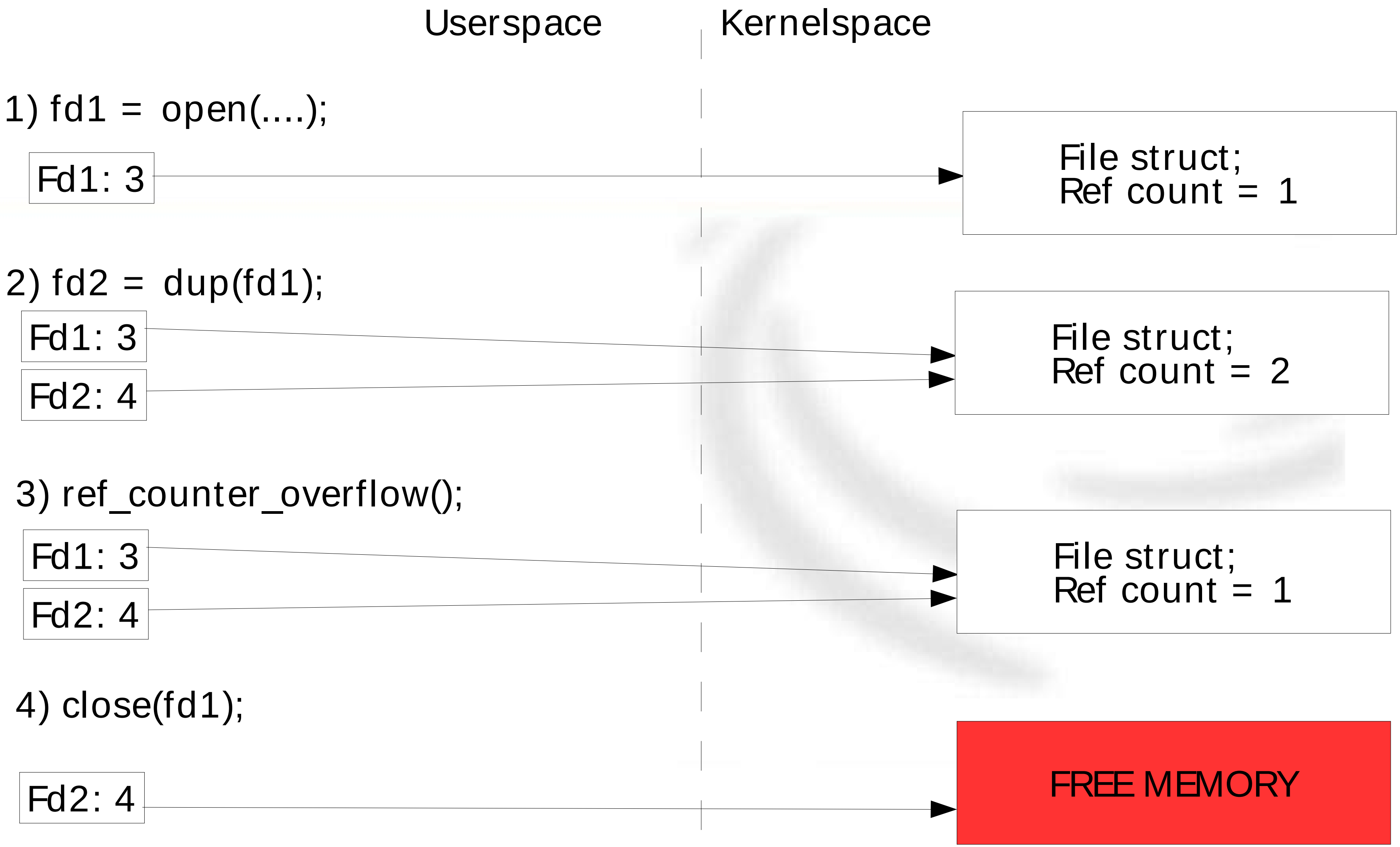
Allocate x bytes

Overflow allocated space

```
static int nargs(u32 src, char **dst){
        int cnt = 0;
        u32 val;
        do {
                int ret = get_user(val, (__u32 *)(u64)
src);
                if (dst) dst[cnt] = (char *)(u64)val;
                src += 4;
                ...
        } while(val);
        if (dst) dst[cnt-1] = 0;
        return cnt;
}
```

Reference counter overflows

- Special case of integer overflow
- Certain structures contain reference counters
- To prevent releasing something when it's still in use
- When a datastructure of this kind is being using the reference counter gets increased, when it's no longer being used the reference counter is decreased.
- Sometimes (mostly in very unlikely error conditions) reference counters don't get decreased.
- In such cases its possible to overflow the counter.
- Have 2 references to some datastruct, ref counter overflow, free 1 of the references, the other one will now point to a freed piece of kernel memory.


Userspace
Kernelspace
1) fd1 = open(....);
Fd1: 3
File struct;
Ref count = 1
2) fd2 = dup(fd1);
Fd1: 3
Fd2: 4
File struct;
Ref count = 2
3) ref_counter_overflow();
Fd1: 3
Fd2: 4
File struct;
Ref count = 1
4) close(fd1);
Fd2: 4
FREE MEMORY

Reference counter overflows

Linux 2.4.x
ppc

```
do_mmap2(unsigned long addr, size_t len,
        unsigned long prot, unsigned long flags,
        unsigned long fd, unsigned long pgoff)
{
        struct file * file = NULL;
        int ret = -EBADF;
        flags &= ~(MAP_EXECUTABLE | MAP_DENYWRITE);
        if (!(flags & MAP_ANONYMOUS)) {
                if (!(file = fget(fd)))
                        goto out;
        }
        ret = -EINVAL;
        if ((! allow_mmap_address(addr)) && (flags & MAP_FIXED))
                goto out;
        down_write(&current->mm->mmap_sem);
        ret = do_mmap_pgoff(file, addr, len, prot, flags, pgoff);
        up_write(&current->mm->mmap_sem);
        if (file)
                fput(file);
out:
        return ret;
}
```

Increase reference counter

Reference counter never gets decreased

Bugs: Information leaks

- Leaking kernel memory to the user
- Could potentially contain useful information (for an attacker)
- Such as tty buffer, memory from sshd, parts of /etc/shadow, bits and pieces of the buffer cache, ...
- Information leaks are usually easy to trigger.
- illustration:

```
#define HOSTNAMELEN 256
char hostname[HOSTNAMELEN];

long gethostname(char *name, int len) {
        if (len > HOSTNAMELEN) {
                len = HOSTNAMELEN;
        }
        copy_to_user(name, hostname, len);
}
```

Bugs: Information leaks (example)

Mac OS X
FreeBSD 4.x

```
static int
ifconf(cmd, data)
        u_long cmd;
        caddr_t data;
{
        struct ifreq ifr, *ifrp;
        ...
        ifrp = ifc->ifc_req;
        for (; space > sizeof (ifr) && ifp; ifp = ifp->if_link.tqe_next)
{
                char workbuf[64];
                int ifnlen, addrs;
                        ...
                        strcpy(ifr.ifr_name, workbuf);
                        ...
                addrs = 0;
                ifa = ifp->if_addrhead.tqh_first;

                for ( ; space > sizeof (ifr) && ifa;
                    ifa = ifa->ifa_link.tqe_next) {
                        if (sa->sa_len <= sizeof(*sa)) {
                                ifr.ifr_addr = *sa;
                                error = copyout((caddr_t)&ifr, (caddr_t)ifrp,
                                                sizeof (ifr));
```

Uninitialized memory

Copy till 0byte, leave the rest uninitialized

Copy everything to the user

PANIC !

- Calling panic() inside most kernels will halt the system
- Usually used when the kernel is in an unrecoverable inconsistent state
- It shouldn't be triggerable from userland (maybe in debug kernel's ?)
- Only results in a denial of service, but a pretty effective one. (no cpu hog, massive stream of packets, ...)

```
int fpathconf(p, uap, retval)
        struct proc *p;
        register struct fpathconf_args *uap;
        register_t *retval;
{
        int fd = uap->fd;
        struct fileproc *fp;
        struct vnode *vp;
        struct vfs_context context;
        int error = 0;
        short type;
        caddr_t data;
         ...
        if ( (error = fp_lookup(p, fd, &fp, 0)) )
                return(error);
        type = fp->f_type;
        data = fp->f_data;

        switch (type) {
        ...
        default:
                panic("fpathconf (unrecognized - %d)", type);
        }
        /*NOTREACHED*/
        ...
}
```

Mac OS X
Old FreeBSD code

An unkown filetype will cause a panic

Bugs: Userland interaction bugs

- Sometimes there are bugs which allow an attacker to modify some resources of a process.
- Resources: screwed up rlimits, closing fd 0,1,2, ptrace bugs, ...
- Requires an suid binary in most cases (or possibly a kernel thread)

Bugs: Userland interaction bugs (example)

Mac OS X

```
int dosetrlimit(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 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;
...
}
}
```

Pass all checks if the rlimit is negative

```
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 */
};
```

Rlimits are signed

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

Getdtablesize can return a neg. value

Bugs: Userland interaction bugs (example)

- Some explanation
  - Rlimits are inherited thru execve()
  - An attacker can set the RLIMIT_NOFILE (maximum open file) to a negative value
  - Almost everywhere in the kernel that value is cast to unsigned (ensuring normal behavior)
  - Getdtablesize() returns that rlimit or the system maximum whatever is smallest
  - A lot of suid binaries used getdtablesize() in a for loop to close file descriptors right before they spawn off a userdefined process (and ofcourse after a privdrop).

More userland interaction stuff

- /proc/pid/mem
- Procfs is a virtual file system
- Used on many unices
- Makes the address space of a process readable and writable for other process thru the use of a virtual file
- Lots of bugs in different implementations
- See http://ilja.netric.org/files/kernelhacking/procpidmem.pdf for more info

More userland interaction stuff II

- /proc/pid/memis called /proc/pid/as on solaris (as == address space)
- Imagine the following code being suid:

```
....
/* open a file without superuser privs */
setreuid(geteuid(), getuid());
fd = open("/proc/mypid/as", O_RDWR);
if (fd < 0) exit(0);
setreuid(getuid(), geteuid());
lseek(fd, whereeveryouwant, SEEK_SET);
write(fd, whateveryouwant, somesize);
...
```

- On solaris open() doesn't fail !!!

SECURING THE SOURCE

Dereferencing user data directly

- When copying data from or to userland some verification needs to be done
- Usually done by functions like copyin()/copyout()
- Sometimes programmers forget to use these functions and dereference pointers given from userspace directly

Dereferencing user data directly (example)

Linux 2.4.x sparc

```
asmlinkage int
sys_ipc (uint call, int first, int second, int third, void *ptr, long fifth)
{
        int version, err;

        version = call >> 16; /* hack for backward compatibility */
        call &= 0xffff;

        if (call <= SHMCTL)
                switch (call) {
                case SHMAT:
                        switch (version) {
                        case 0: default: ....
                        case 1: /* iBCS2 emulator entry point */
                          err = sys_shmat (first, (char *) ptr, second, (ulong *) third);
                          goto out;
                        }
                ...
```

User specified address

```
asmlinkage long
sys_shmat (int shmid, char *shmaddr, int shmflg, ulong *raddr){
        ...
        *raddr = (unsigned long) user_addr;
        ...
}
```

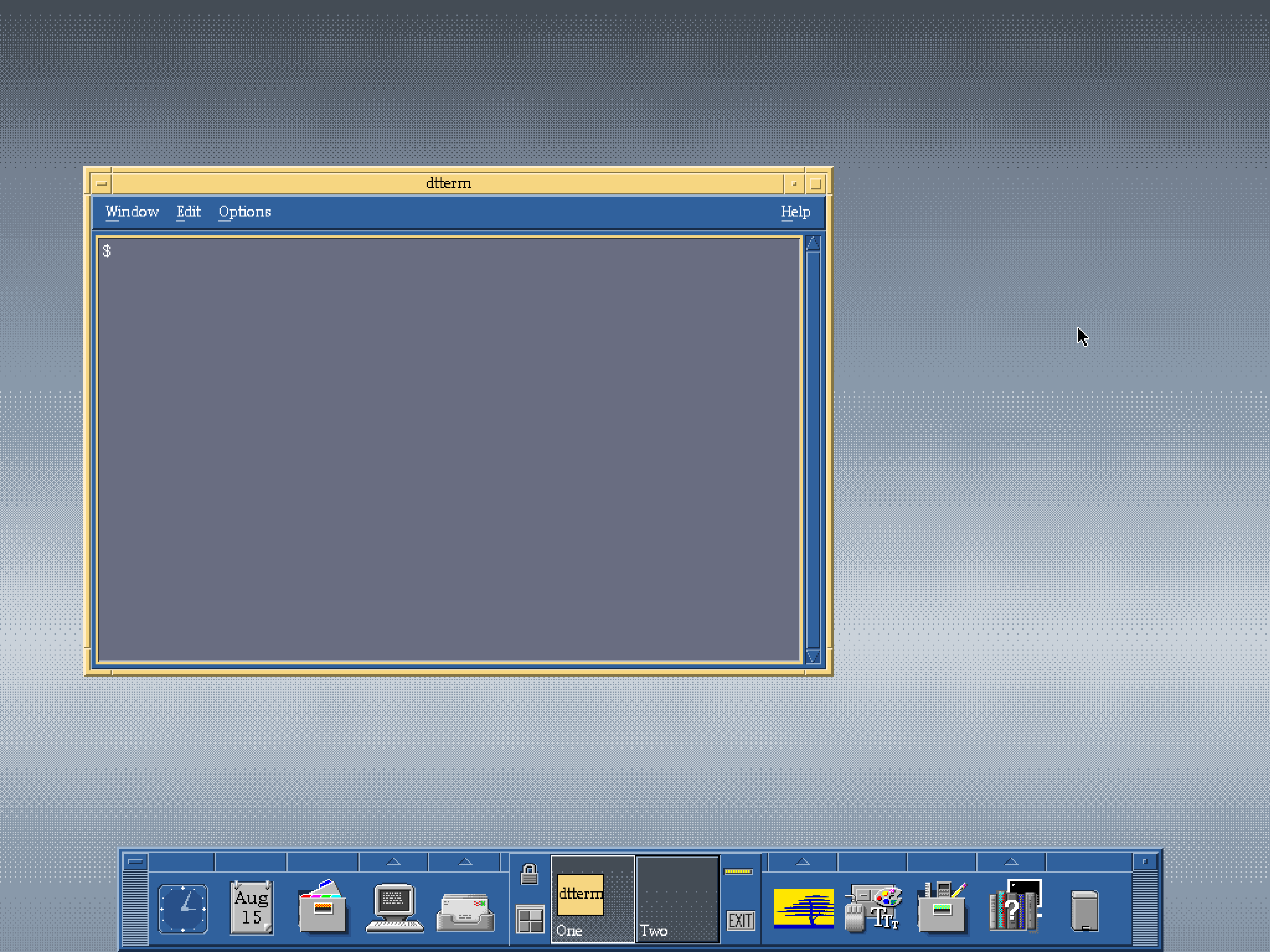
Fuzzing the kernel

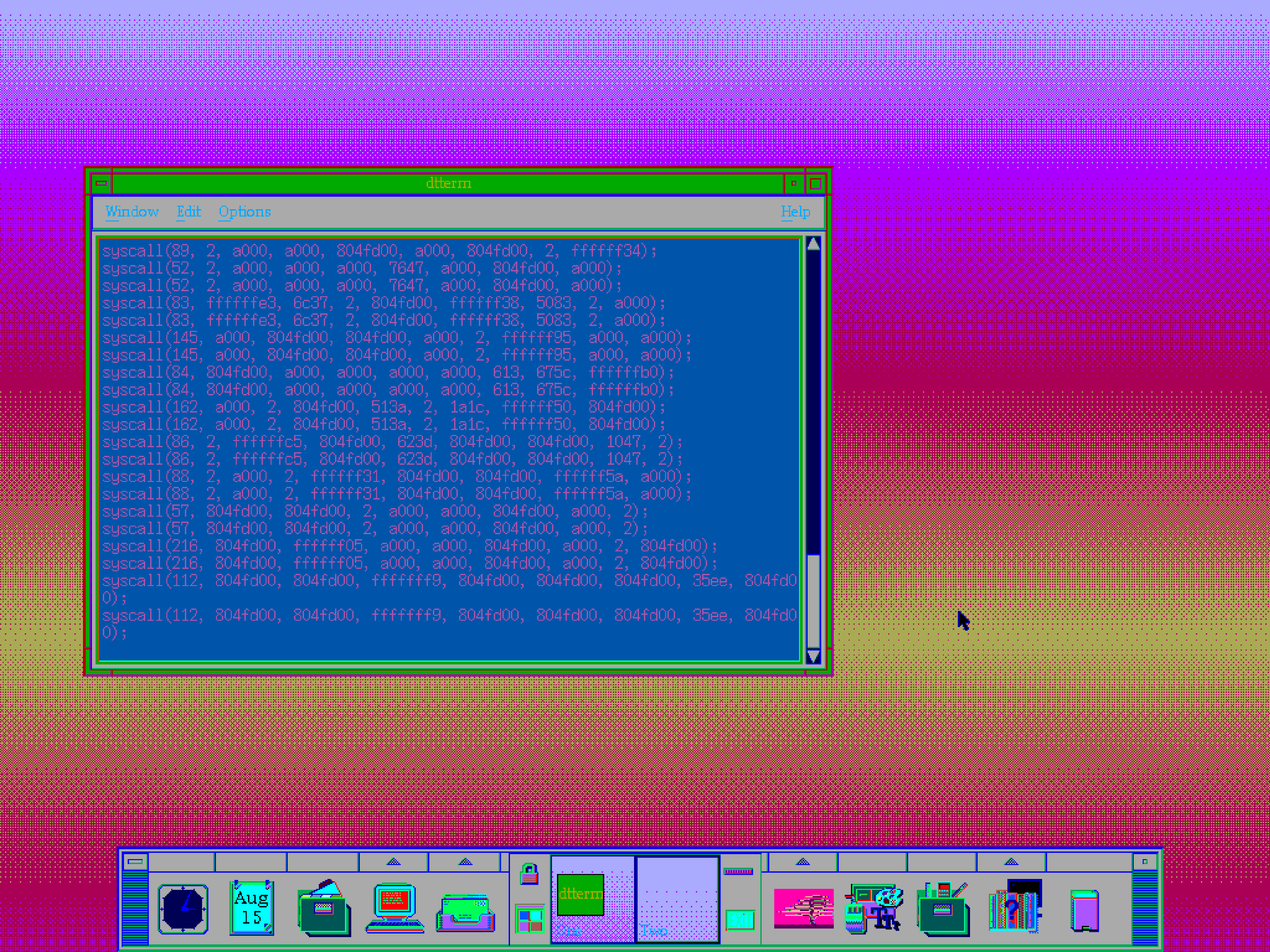
What is fuzzing

- Using semi-valid data: good enough to pass initial checks, bad enough so things might go wrong
- Can be used in a lot of things
- We'll only discuss fuzzing related to the xnu kernel
- What can you fuzz:
  - Syscall arguments
  - Binary files the kernel has to process

Syscall argument fuzzing

- Generate a random syscall number
  - Mac OS X also has some negative syscall nr's !
- All syscalls have at most 8 arguments (special case: 1 mach syscall has 9 arguments)
- Generate 8 “random” arguments
- “random”:
  - Some random number
  - A valid userland address
    - Get some (random) data on it
  - Address of an unmapped page
  - Some kernelspace address
  - Small negative nr
  - ...


dtterm
Window
Edit
Options
Help
$
Aug
15
dtterm
One
Two
EXIT


dtterm
Window
Edit
Options
Help
syscall(89, 2, a000, a000, 804fd00, a000, 804fd00, 2, ffffff34);
syscall(52, 2, a000, a000, a000, 7647, a000, 804fd00, a000);
syscall(52, 2, a000, a000, a000, 7647, a000, 804fd00, a000);
syscall(83, ffffffe3, 6c37, 2, 804fd00, ffffff38, 5083, 2, a000);
syscall(83, ffffffe3, 6c37, 2, 804fd00, ffffff38, 5083, 2, a000);
syscall(145, a000, 804fd00, 804fd00, a000, 2, ffffff95, a000, a000);
syscall(145, a000, 804fd00, 804fd00, a000, 2, ffffff95, a000, a000);
syscall(84, 804fd00, a000, a000, a000, a000, 613, 675c, ffffffb0);
syscall(84, 804fd00, a000, a000, a000, a000, 613, 675c, ffffffb0);
syscall(162, a000, 2, 804fd00, 513a, 2, 1a1c, ffffff50, 804fd00);
syscall(162, a000, 2, 804fd00, 513a, 2, 1a1c, ffffff50, 804fd00);
syscall(86, 2, ffffffc5, 804fd00, 623d, 804fd00, 804fd00, 1047, 2);
syscall(86, 2, ffffffc5, 804fd00, 623d, 804fd00, 804fd00, 1047, 2);
syscall(88, 2, a000, 2, ffffff31, 804fd00, 804fd00, ffffff5a, a000);
syscall(88, 2, a000, 2, ffffff31, 804fd00, 804fd00, ffffff5a, a000);
syscall(57, 804fd00, 804fd00, 2, a000, a000, 804fd00, a000, 2);
syscall(57, 804fd00, 804fd00, 2, a000, a000, 804fd00, a000, 2);
syscall(216, 804fd00, ffffff05, a000, a000, 804fd00, a000, 2, 804fd00);
syscall(216, 804fd00, ffffff05, a000, a000, 804fd00, a000, 2, 804fd00);
syscall(112, 804fd00, 804fd00, fffffff9, 804fd00, 804fd00, 804fd00, 35ee, 804fd00);
syscall(112, 804fd00, 804fd00, fffffff9, 804fd00, 804fd00, 804fd00, 35ee, 804fd00);
Aug
15
dtterm
One
Two

More detailed argument fuzzing

- The previous method is trivial and not detailed at all, but can be implemented in a matter of minutes
- You can do more detailed syscall fuzzing
- Examples:
  - socket() fuzzing
    - Once socket is made so al sort of socket operations on it:
      - Setsockopt
      - Getsockopt
      - Bind
      - ...
  - Check out Peter Holm's Stresstest suit for the FreeBSD kernel! (there is a Mac OS X port by Christian Klein)

More detailed argument fuzzing (example)

- Linux bluetooth driver NULL pointer dereference:

```
Unable to handle kernel NULL pointer dereference at virtual address 00000018
 printing eip:
dcc3342b
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<dcc3342b>]    Tainted: P
EFLAGS: 00010246
eax: 00000000   ebx: d4b4e030   ecx: 00000003   edx: c038ad98
esi: d587bf04   edi: c038ad98   ebp: bffff788   esp: d587bedc
ds: 0018   es: 0018   ss: 0018
Process sfuzz (pid: 27651, stackpage=d587b000)
Stack: d4b77b50 d587bf04 00000001 c02654c5 d4b77b50 d587bf04 d587befc 00000001
       00000004 c01c705c db84001f 0000000a 00000001 db84f000 00000246 00000286
       00000000 40017059 c01c8e59 db84f988 00000000 00000000 d587a000 00000000
Call Trace:    [<c02654c5>] [<c01c705c>] [<c01c8e59>] [<c01c4975>] [<c0265dc6>]
  [<c0136ae7>] [<c0108a93>]

Code: 66 8b 40 18 66 89 46 02 ff 05 98 ad 38 c0 8b 83 a8 00 00 00

katrien%
```

suresec
SECURING THE SOURCE

Binary file fuzzing

- Unintelligent file fuzzing
  - Take a valid file
  - Randomly modify some bytes
  - VERY EASY
  - SHOCKING RESULTS
- Intelligent fuzzing
  - Can take a while to make something decent
  - Need to know specifics of the kind of file parsing that you're going to fuzz
  - Might be hard for closed source kernels
- What can you fuzz with it:
  - Mach-o-runtime
  - .dmg image file loading
- You should check out Michael Sutton and Adam Greene's slides from blackhat.

JFS breaks withing seconds

```
VFS: Busy inodes after unmount. Self-destruct in 5 seconds.  Have a nice day...
VFS: Busy inodes after unmount. Self-destruct in 5 seconds.  Have a nice day...
ERROR: (device loop(7,7)): XT_GETPAGE: xtree page corrupt
ERROR: (device loop(7,7)): XT_GETPAGE: xtree page corrupt
ERROR: (device loop(7,7)): XT_GETPAGE: xtree page corrupt
BUG at jfs_xtree.c:751 assert(!BT_STACK_FULL(btstack))
kernel BUG at jfs_xtree.c:751!
invalid operand: 0000
CPU:    0
EIP:    0010:[<dcbe3c01>]    Not tainted
EFLAGS: 00010286
eax: 0000003a   ebx: 00000002   ecx: d3cac000   edx: d661d2e0
esi: d3f82280   edi: 00eb0003   ebp: 00000000   esp: d3cadbc4
ds: 0018   es: 0018   ss: 0018
Process mount (pid: 6098, stackpage=d3cad000)
Stack: dcc008be dcc008b2 000002ef dcc0089a 0003dc18 d3cadc85 d445bb30 c02c40d8
       d46a6918 00000000 d46a6860 00000000 00000000 00000001 00000002 d3f82280
       d3f8224c 00000000 00000000 ffffffff d3f82200 00000000 00000000 00000000

Call Trace:    [<dcc008be>] [<dcc008b2>] [<dcc0089a>] [<c02c40d8>] [<dcbe30ac>]
  [<c0153440>] [<dcbe0170>] [<c0138d37>] [<c01395bf>] [<c012ece8>] [<dcbe033f>]
  [<dcbe0080>] [<c012ad11>] [<dcbf8397>] [<dcbe0330>] [<dcbecafc>] [<dcbe8ff7>]
  [<dcbe2791>] [<dcbdf77a>] [<dcc00ff0>] [<c013bc9b>] [<dcc00ff0>] [<dcc00ff0>]
  [<dcc00ff0>] [<c014c66b>] [<dcc00ff0>] [<c013bf91>] [<dcc00ff0>] [<c014d5cc>]
  [<c014d883>] [<c014d6fb>] [<c014dc8b>] [<c0108a93>]

Code: 0f 0b ef 02 b2 08 c0 dc 83 c4 10 8b 5c 24 70 8b 13 e9 6e ff
katrien%
```

Some comments on kernel fuzzing

- Finding the actual problem once you trigger a crash can be hard
- You need to get memory dumps of the panic'd kernel
- Kernel debugging is useful: Mac OS X has default gdb stubs for remote debugging.
- Fuzzing race conditions is possible, but it's hard
- Figuring out where a race happens and under what conditions is pure hell!

Some comments

- This is not a complete list, a lot more things can (and will) go wrong
- This is meant to give an idea that:
  - os designers also make coding mistakes
  - That you don't have to be a c guru to find bugs in kernels
  - All unix kernels have critical security bugs
- Go out and break a kernel :)
- Better yet, go fix one

sureSEC
SECURING THE SOURCE

Q&A