

Bug classes we have found in *BSD,
OS X and Solaris kernels

BSDCITIZEN



<http://www.bsdcitizen.org/>

Christer Öberg – christer [at] signedness.org
– christer [at] bsdcitizen.org

Neil Kettle – mu-b [at] bsdcitizen.org // digit-labs.org
– Digit Security Ltd
<http://www.digit-security.com/>

► But first, a prelude..

“Most security researchers [...] spend most of their time combing through source code looking for bugs. This is certainly useful, but it doesn't require very much skill, and I suspect that this is a task that will be taken over by computer programs before long, since most security flaws fall into the 'stupid mistake' category and are very easy to recognize if you look closely enough; I've been particularly impressed in this respect with results from software produced by Coverity.”

- Colin Percival,
FreeBSD Security Team

► Why do we bother? (aka motivation)

- Colin's words are a little too 'hilbertonian' for our liking,
 - despite 30 years of static analysis, trivial bugs still slip through (Rices) 'net'...
 - theoretical limitations are 'well' known (by 'well' we mean academia, otherwise unknown/ignored)
 - may soon get better! Fortify sponsored the 1st International Workshop on Bit-Precise Reasoning
 - but still... there is no 'silver'-bullet, nor could any 'silver'-bullet exist.
- Relying on static analysis alone, is therefore, somewhat foul-hardy

► Why do we bother? (aka motivation)

- To demonstrate that trivial bugs still exist!
 - the presence of trivial bugs in well-used Operating System kernels, despite the number of 'Cons that exist today, is frankly depressing!
 - in general, more 'Cons, more auditing
 - kernel bugs typically aren't worth many \$\$\$ ↯☺
- Oh no, not yet-another-exploit-howto,
 - not quite!, we wouldn't be doing this if it wasn't still easy!

▶ Why do we bother? (aka motivation)

- And finally, because you might find it interesting! (*)

* unless you happen to be *@gnucitizen.org

► What we will show you today

- Lots of new bugs
 - Covering many bug classes
 - signedness, race conditions, memory leaks, arbitrary memory corruption (kernel/user pointer mixing), NULL pointer dereferences, memory disclosures, DoS, and a remote overflow...
 - Covering many kernel interfaces...
 - system calls, file systems, network code etc...
 - Kernel exploits
 - can be made exceptionally reliable (with a few exceptions!)
 - are much easier than most people think

▶ Signedness Issues

- The Good News,
 - very common – found in every kernel!
 - the 'last' vestige of the trivial overflow (you would hope)
 - very simple to find – at least manually
 - as a consequence very satisfying to find! (for a sadistic mind)
- The Bad News,
 - many bugs are only Memory Disclosures
 - but we consider only code exec via a signedness bug as a 'signedness bug' herein
 - exploitation can be difficult in some scenarios (dependant on memory model)

▶▶ 'Current' Example

- Vulnerable: FreeBSD ≥ 7.0
- A signedness bug exists in the `ktimer_delete` syscall

```
{ AS(ktimer_delete_args),  
  (sy_call_t *)ktimer_delete, AUE_NULL,  
  NULL, 0, 0 }, /* 236 = ktimer_delete */
```

- Bug is over 3 years old now!

```
Sun Oct 23 04:22:55 2005 UTC (3 years, 4 months ago) by  
davidxu
```

- User-supplied integer used in heap allocated structure array lookup
- Allows for local kernel mode code execution...

▶▶▶ ktimer_delete

```
#define TIMER_MAX          32

sys/kern/kern_time.c:

static struct itimer *
itimer_find(struct proc *p, int timerid)
{
    ...

    if ((p->p_itimers == NULL) || (timerid >= TIMER_MAX)
        || (it = p->p_itimers->its_timers[timerid]) ==
            NULL) {
        return (NULL);
    }

    ...

    return (it);
}
```

- Reachable via a call to `ktimer_delete()` with user-supplied `timerid`.

▶▶ ktimer_delete

sys/kern/kern_time.c:

```
static int
kern_timer_delete(struct thread *td, int timerid)
{
    struct proc *p = td->td_proc;
    struct itimer *it;


    PROC_LOCK(p);
    it = itimer_find(p, timerid);
    if (it == NULL) {
        PROC_UNLOCK(p); return (EINVAL);
    }
    PROC_UNLOCK(p);

    ...

    it->it_flags &= ~ITF_WANTED;
    CLOCK_CALL(it->it_clockid, timer_delete, (it));

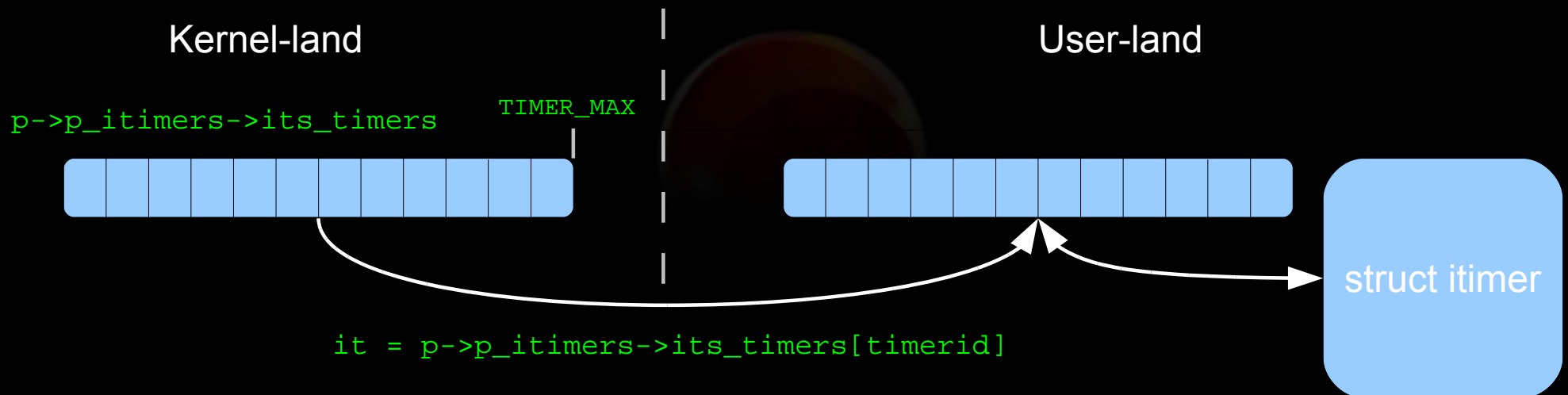
    ...

    return (0);
}
```



▶▶ ktimer_delete

- Since `*p_itimers` is allocated on the heap, `(it = p->p_itimers->its_timers[timerid])` resolves to a 'random' address
 - userland pointer spray required for a const `timerid` value with bounded `&its_itimers`
 - pointer resolves to userland address with a valid `itimer` structure



▶▶ ktimer_delete

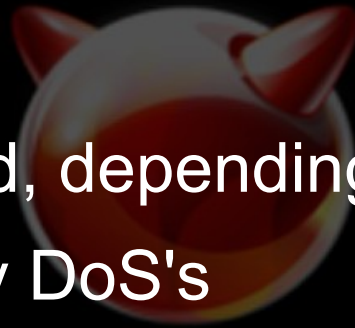
- Thanks to the brilliance of BSD and function pointers,

```
#define CLOCK_CALL(clock, call, arglist) \
    ((*posix_clocks[clock].call) arglist)
```

- compute a value of `it->it_clockid` such that `*posix_clocks[it->it_clockid]` resolves to a user-land address
- obtaining `&posix_clocks` is simple with `kdsym()`
- place the value for `it->it_clockid` into the user-land itimer structure with a suitable `*posix_clocks[it->it_clockid].timer_delete` pointer and our work is done 😊

▶ Race Conditions

- The Good News,
 - quite common – found in nearly every kernel!
- The Bad News,
 - not so simple to find, depending on complexity
 - many bugs are only DoS's
 - exploitation can be difficult in some scenarios



▶▶ 'Current' Example

- Vulnerable: Solaris ≥ 10 , FreeBSD ≥ 7.1
- Several copyin() race conditions exist in the DTRACE subsystem of the above kernels
- Bugs are really quite old now
- Allows all length/encapsulation checking to be easily by-passed, or local heap corruption
- At the moment, simply a DoS until someone proves that wrong...

▶▶ /dev/dtrace

```
static dof_hdr_t *
dtrace_dof_copyin(user_addr_t uarg, int *errp)
{
    dof_hdr_t hdr, *dof;

    if (copyin(uarg, &hdr, sizeof (hdr)) != 0) {
        dtrace_dof_error(NULL, "failed to copyin DOF header");
        *errp = EFAULT; return (NULL);
    }

    if (hdr.dofh_loadsz >= dtrace_dof_maxsize) {
        dtrace_dof_error(&hdr, "load size exceeds maximum");
        *errp = E2BIG; return (NULL);
    }

    if (hdr.dofh_loadsz < sizeof (hdr)) {
        dtrace_dof_error(&hdr, "invalid load size");
        *errp = EINVAL; return (NULL);
    }

    dof = dt_kmem_alloc_aligned(hdr.dofh_loadsz, 8, KM_SLEEP);

    if (copyin(uarg, dof, hdr.dofh_loadsz) != 0)
        ...
}
```

▶▶ Another 'Current' Example

- Another copyin race condition exists in the /dev/fasttrap IOCTL handler
- Further race conditions exist in the OS X HFS_SET_PKG_EXTENSIONS sysctl interface for HFS filesystems
 - permits either a memory leak, or un-exploitable double free()



▶ Arbitrary Memory Corruption

- The Good News,
 - very uncommon – a `vmsplice()` doesn't come along too often!
 - unless it's a third-party Windows driver!
 - very simple to find – but very few in existence
 - very simple to exploit
- The Bad News,
 - very few in existence
 - nothing much else...

▶▶ 'Current' Example

- Vulnerable: Mac OS X \geq 10.4.0 (xnu \geq 792.0)
- An arbitrary memory read/write bug exists in the HFS IOCTL handler

```
int hfs_vnop_ioctl(struct vnop_ioctl_args *ap)
```

- Bug is nearly 4 years old now!
- A 'Logic' flaw whereby a piece of kernel-mode **only** code is reachable via user-land
 - code is obviously kernel-mode **only** since even non-malicious use is guaranteed to cause a panic()
- Allows for local kernel mode code execution...

▶▶ HFS IOCTL

```
#define FCNTL_FS_SPECIFIC_BASE 0x00010000
#define HFS_GET_BOOT_INFO      (FCNTL_FS_SPECIFIC_BASE + 0x00004)
#define HFS_SET_BOOT_INFO      (FCNTL_FS_SPECIFIC_BASE + 0x00005)

int
hfs_vnop_ioctl(struct vnop_ioctl_args ... *ap)
{
    ...
    case HFS_SET_BOOT_INFO:
        ...
        HFS_MOUNT_LOCK(hfsmp, TRUE);
        bcopy(ap->a_data, &hfsmp->vcbFndrInfo, sizeof(hfsmp->vcbFndrInfo));
        HFS_MOUNT_UNLOCK(hfsmp, TRUE);
        (void) hfs_flushvolumeheader(hfsmp, MNT_WAIT, 0);
        break;

    case HFS_GET_BOOT_INFO:
        ...
        HFS_MOUNT_LOCK(hfsmp, TRUE);
        bcopy(&hfsmp->vcbFndrInfo, ap->a_data, sizeof(hfsmp->vcbFndrInfo));
        HFS_MOUNT_UNLOCK(hfsmp, TRUE);
        break;
    ...
}
```

- It is possible to control `ap->a_data` via `fcntl()`

▶▶ HFS IOCTL

- From the `fcntl()` syscall we have,

```
int
fcntl_nocancel(proc_t p, struct fcntl_nocancel_args *uap, ...)
{
    argp = uap->arg;
    ...
    switch (uap->cmd) {
    ...
    default:
        if (uap->cmd < FCNTL_FS_SPECIFIC_BASE) {
            error = EINVAL; goto out;
        }

        // if it's a fs-specific fcntl() then just pass it through
        ...
        error = VNOP_IOCTL(vp, uap->cmd, CAST_DOWN(caddr_t, argp), 0,
            &context);
        ...
    }
}
```

- Since `HFS_GET_BOOT_INFO > FCNTL_FS_SPECIFIC_BASE`,
`uap->arg` is left untouched!

▶ NULL Pointer Dereferences

- The Good News,
 - pretty common – found them in lots of kernels
 - often caused by pointer being NULL initialized
 - as opposed to assigned from failed calls etc...
 - Great news, function pointers everywhere!
 - NULL can be mapped with `mmap()`
 - controlling a kernel function pointer...
 - what could possibly go wrong... ☺

▶ NULL Pointer Dereferences

- The Bad News,
 - pretty much none...
 - constitutes a DoS if NULL page unusable (OS X)
 - unless the offset is user-definable!
 - unfortunately NULL page is usable on linux, but they are still seen as 'stability' issues

▶▶ Recent Example

- FreeBSD-SA-08:13.protosw
 - function pointers NULL initialized for certain sockets
 - NetGraph and Bluetooth sockets
 - function pointer called by shutdown() on socket
 - good example how trivial kernel bugs can be
 - mmap() NULL address and copy shellcode
 - create socket
 - call shutdown() on socket
 - our exploit is available at www.bsdcitizen.org

▶▶ More NULL Pointer Fun

- malloc() can fail...
 - FreeBSD malloc Fault Injection
 - kernel option MALLOC_MAKE_FAILURES
 - causes every N'th call to malloc() with M_NOWAIT to fail
- How to force failures?
 - resource starvation...
 - using other bugs
 - or just consuming a lot them

▶▶ 'Current' Example

- Vulnerable: FreeBSD ≥ 7.0
- A resource starvation bug exists in the kenv syscall
- From kenv() (FreeBSD 7.0 and up)

```
if (uap->len > 0 && uap->value != NULL)
    buffer = malloc (uap->len, M_TEMP, M_WAITOK|M_ZERO);
```

- Allocating ~2gig in kernel?
 - just not going to happen, one-liner DoS
 - `kenv (KENV_DUMP, "uh", "oh", 0x7fffffff);`
 - a moderate request could potentially
 - cause an M_WAITOK malloc() to fail may lead to other exploitable NULL pointer bugs

▶▶ kenv

```
[christer@ ~]$ cat kenv.c
#include <kenv.h>
main()
{
    kenv(KENV_DUMP, "uh", "oh", 0x7fffffff);
}

[christer@ ~]$ cc kenv.c -o kenv
[christer@ ~]$ ./kenv
panic: kmem_malloc(-2147483648): kmem_map too small: 5701632 total allocated
cpuid = 0
Uptime: 13m20s
Physical memory: 499 MB
Dumping 36 MB: 21 5
Dump complete
Automatic reboot in 15 seconds - press a key on the console to abort
█
```

▶▶ Memory Disclosures

- The Good News,
 - pretty common – found them in lots of kernels
 - many different types,
 - length argument
 - completely user controlled (yes, they do exist!!!)
 - length miscalculated due to signedness bugs, etc
 - source address can sometimes be,
 - influenced (reading at an arbitrary offset into a file for example)
 - completely user controlled (I'm not joking!)
- The Bad News,
 - don't permit local code execution

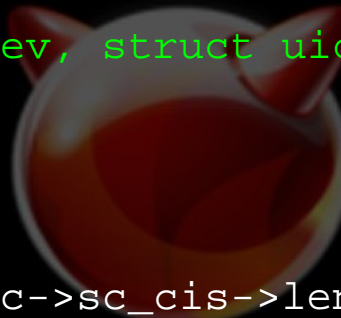
▶▶ 'Current' Example

- Vulnerable: FreeBSD ≥ 7.0
- A memory disclosure bug exists in the cardbus read handler

```
int
cardbus_read(struct cdev *dev, struct uio *uio,
             int ioflag)
{
    ...

    if (sc->sc_cis == NULL ||
        uio->uio_offset > sc->sc_cis->len)
        return (0);

    return (uiomove(sc->sc_cis->buffer + uio->uio_offset,
                   MIN(uio->uio_resid,
                       sc->sc_cis->len - uio->uio_offset), uio));
}
```



- `uio->uio_offset > sc->sc_cis->len` comparison is signed
- Many, many more examples of this still in BSD

▶▶ cardbus handler

- Raw dump of kmem to FS == Easy
 - but lame, better to target something useful
 - PRNG pools, Network / console buffers
- Getting your addresses right
 - target addresses – `kldsymb()`
 - `main_console` for console buffers for example
 - base address
 - start with large negative offset
 - scan until we find something we recognise at known address

▶▶ Another 'Current' Example

- Vulnerable: FreeBSD 8.0, NetBSD 5.0
- An arbitrary memory disclosure bug exists in the linux `sys_set_robust_list` handler
- Allows arbitrary kernel memory to be read via a user-definable pointer



▶▶ Another 'Current' Example

```
linux_sys_set_robust_list(  
    struct lwp *l,  
    const struct linux_sys_set_robust_list_args *uap,  
    register_t *retval)  
{  
    struct proc *p = l->l_proc;  
    struct linux_emuldata *led = p->p_emuldata;  
  
    led->robust_futexes = SCARG(uap, head);  
    *retval = 0;  
  
    return 0;  
}  
  
linux_sys_get_robust_list(  
    struct lwp *l,  
    const struct linux_sys_get_robust_list_args *uap,  
    register_t *retval)  
{  
    ...  
    led = p->p_emuldata;  
    head = led->robust_futexes;  
    ...  
    return copyout(head, SCARG(uap, head),
```

► Overflows

- Not really that different from userland
 - but careful, you only get one chance!
 - stack overflows are quite rare
 - heap overflows
 - Argp (Patroklos Argyroudis) has done lots of work on FreeBSD UMA overflows...
 - MBUF overflows
 - lots of NetBSD bugs, Alfredo Ortega's IPv6 bug, etc



▶▶ Quick Mbuf Intro

- Mbuf are small units of memory
- Mbuf got a couple of headers
 - pointers for linked lists etc
 - standard unlink heap overflow stuff...
 - Mbuf can also have external storage / free
 - in which case a function pointer in mbuf will be called
 - this is the best way to exploit them
 - several arguments to the called function are controlled
 - this turned out to be a life saver for the next exploit!

▶▶ m_ext_free()

```
m_ext_free(struct mbuf *m)
{
    ...

    else if (m->m_ext.ext_free) {
        (*m->m_ext.ext_free)(m, m->m_ext.ext_buf,
            m->m_ext.ext_size,
            m->m_ext.ext_arg);
    }
    ...
}
```

▶▶▶ 'Current' Example

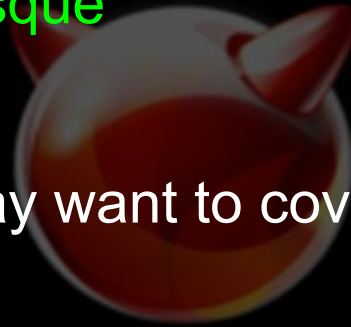
- Vulnerable: NetBSD (up to 4.0 out of box)
- Several remote bugs exist in the OSI, TP networking stack
 - two remote bugs, DoS and code execution
 - and countless locals...
- NetBSD up to 4.0 is vulnerable out of the box
 - XEN kernel and other configs still enable it by default
 - the DoS does not even require the protocols to be used, as the attack can be encapsulated in IP
 - for remote code execution a “listener” is required...

Sequential Ipids actually useful for a change!!! Who would have thought!!

▶▶ WARNING WARNING WARNING...

- Some comment from the code...
 - “Decomposing the tpdu is some of the most **laughable code**.”
 - “such **abominations** as the macros WHILE_OPTIONS“
 - “this seems a bit **grotesque**”

If you are sensitive you may want to cover your eyes for the next couple of slides.....



▶▶ WTF???!!!!

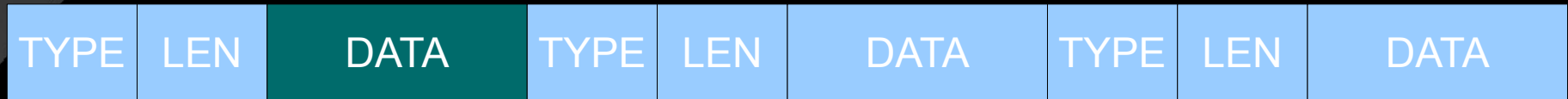
```
#define WHILE_OPTIONS(P, hdr, format)\
{ char *P = tpdu_info[(hdr)->tpdu_type][(format)] + (char *)hdr;\
  char *PLIM = 1 + hdr->tpdu_li + (char *)hdr;\
  for (;;) P += 2 + ((struct tp_vbp *)P)->tpv_len) {\
    CHECK((P > PLIM), E_TP_LENGTH_INVALID, ts_inv_length,\
          respond, P - (char *)hdr);\
    if (P == PLIM) break;\
  }\
#define END_WHILE_OPTIONS(P) } }
```

▶▶ Trying To Make Sense Of It...

- Entering the loop
 - P points to beginning of options
 - PLIM points to end of options
- FOR loop parses options until
 - $P == PLIM$
 - $P > PLIM$ (error, invalid packet length)
 - each iteration adds length + 2 to P
- Options are dealt with in-between the two macros

▶▶ Maybe Visualization Helps

- `for (; P += 2 + ((struct tp_vbp *)P)->tpv_len)`



- Got a slight problem here...
 - signedness issue. We can have negative length!
 - `tpv_len == 0xfe`
 - will not increment / move P along options pointer
 - and loop will never break

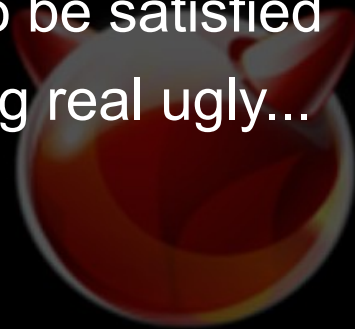
But simple DoS is not nearly cool enough...

▶▶ What About The Actual Options?

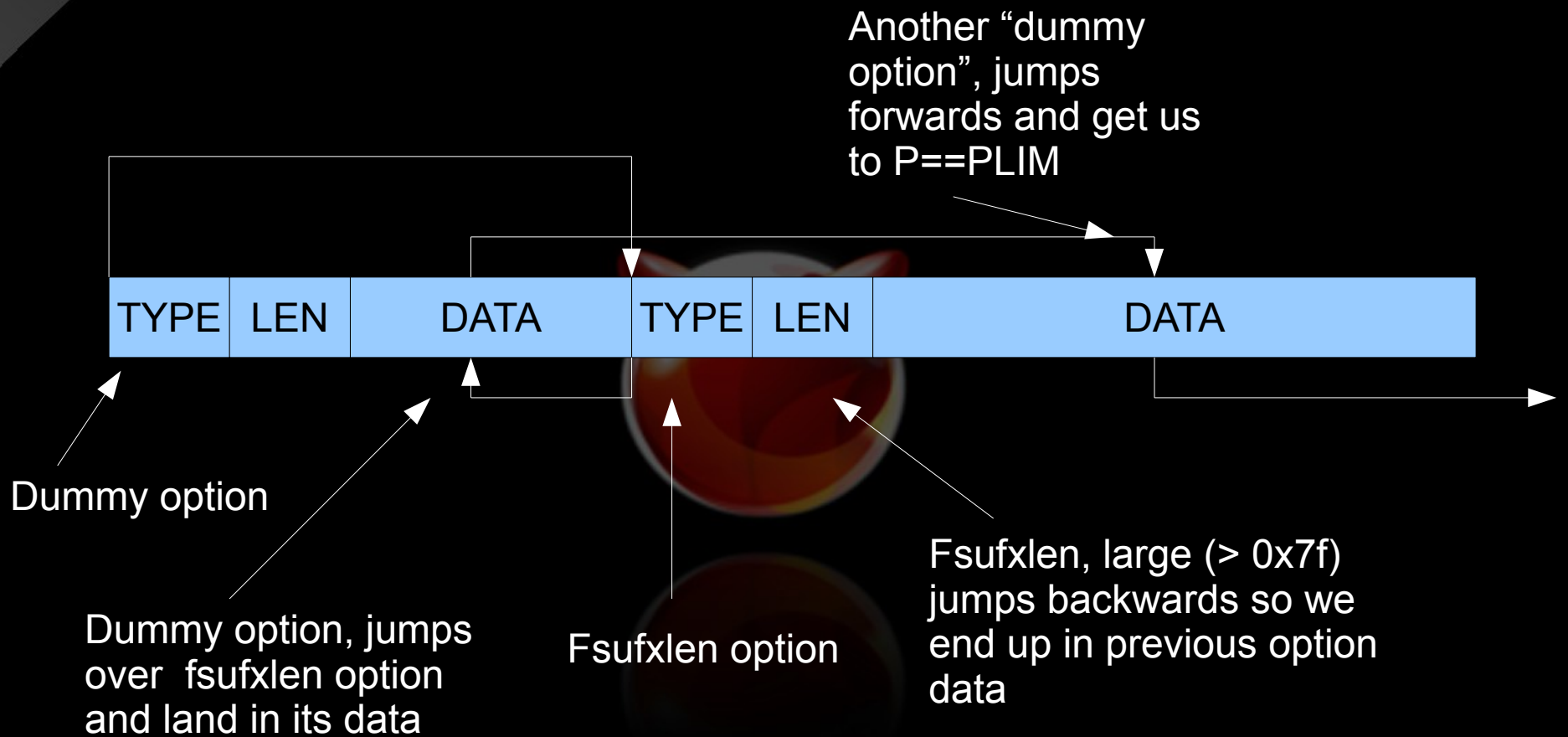
```
case TPP_calling_sufx:
    /*
     * could use vb_getval, but we want to save the loc &
     * len for later use
     */
    fsufxloc = (void *) & vbptr(P)->tpv_val;
    fsufxlen = vbptr(P)->tpv_len;
    ...
END_WHILE_OPTIONS
    ...
    bcopy(fsufxloc, tpcb->tp_fsuffix, fsufxlen);
    ...
```


▶▶ That Looks Interesting

- But there are a few problems
 - we'll need more than 0x7f to hit interesting things
 - signedness issue complicates things
 - $P \Rightarrow PLIM$ needs to be satisfied
 - sigh... This is turning real ugly...



►► Confusing Enough Yet?



▶▶ What Do We Have Now?

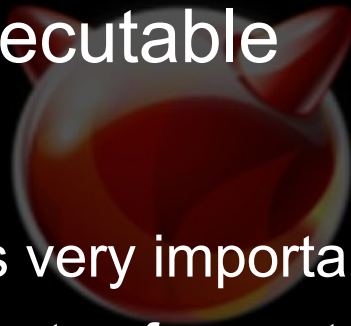
- Arbitrary MFREE()
 - as we are overwriting an mbuf pointer
 - can't point it to userland though...
 - need to spray the remote system with packets containing “fake” mbufs
 - payload can also be stored in spray packets
 - just point function pointer to payload and that is it?
 - I tried... but it is not quite that easy...

▶▶ AMD64...

- Trying to execute an int3 in mbuf
 - stopped the kernel
 - but for the wrong reason...
 - turns out on AMD64 mbufs are non-exec!\$%!
 - so another problem to solve...
 - defeating non-exec page remotely... can't be that hard can it?

▶▶ Getting Code Execution

- Thought about “ret-into-text”
 - calling standard not really helping...
- Found that .bss is executable
 - “bootinfo”
 - doesn't look like it is very important
 - and it does have plenty of room to store code



▶▶ Getting Code Execution 2

- First argument function called is mbuf pointer
 - memcpy(mbuf,bootinfo,size)
 - arguments wrong way around...
 - bcopy(mbuf,bootinfo,size) – would work...
 - but bcopy() is just a macro around memcpy
 - kcopy() - like bcopy but with exception handling!
 - and it is a proper function
- Using the m_ext_free() technique
 - I can now copy code to executable location...
 - but also “used up” my arbitrary code execution...

▶▶ Getting Code Execution 3

- Another linked “fake mbuf” in spray packets
 - gives me another arbitrary function pointer
 - this means I can execute the executable copy of my code
 - but also leaves very little room in spray packets...
 - will have to stage it and spray another set of packets...
 - 1st Stage locates 2nd stage, copies it and executes it

▶▶ 2nd Stage Shellcode

- I'm lazy / not as cool as cmn / Karl Janmar (*)
 - instead of hooking network input functions
 - I “cheat” and infect a userland process
 - nice side effect is killing syslogd (we don't need that :))
 - shellcode scans “allproc” list looking for “syslog”
 - copies out a userland shellcode (copyout_proc())
 - executes it with process_set_pc()

* Karl's 802.11 bug / exploit is IMHO the most under appreciated exploit ever... So if you haven't seen it you should check it out :)

► Final Notes...

- If you're a Solaris kernel developer don't,
 - assume `round(a,b) == a` for all values of `a`, `b`
 - particularly important when computing lengths
 - leave 'test' code such as,

```
int _sd_test_init(void *args)
{
    ...
    if (copyin(uap->addr, devarray[uap->ar], uap->len))
    {
        return (EFAULT);
    }
    ...
}
```

lying around, reachable if root executes 'ncall -e'

► Final Notes...

- If you're an OS X kernel developer don't,
 - fix the Appletalk networking stack!
 - if it actually worked properly we would be demonstrating two remote kernel bugs and not just one!\$%!
 - fix anything else too! There are many other 'bugs' that are, as yet (*), unreachable due to developer bugs!\$%!

* despite me trying to get them fixed.

► Conclusions

- Source code auditing is **still** required,
 - static analysis will never take over, thankfully
- Kernel bugs are still in abundance!
 - even in commonly used kernels
 - PWN2OWN will likely never cover kernel bugs,
 - there aren't enough Macs to give away



DEMOS

(if the OSX bug crashes there will be no conclusions)