OOPS!

An Introduction to Linux Kernel Debugging

Simon Trimmer <simon@urbanmyth.org>

Overview

- types of crashes
- oopsen
    - capturing
    - decoding
    - debugging...
- lockups
- bug reports
- further information

Not Covering...

Would have liked to have covered...

- OS principles
- kernel debuggers
- crash dump generators
- hardware faults

If there is the interest we could run another talk to cover them some other time.

Different types of crashes

- the common oops
- lockup types
    - temporary and permanent
- spontaneous reboot (not covered in detail)

Q? (what sort of audience is this?)

part 1a

oops basics

The Oops

An oops is triggered by some exception and is a dump of the CPU state and kernel stack at that instant.

oopsen can get sent to:

- the console
  - possibly a serial port
- the kernel ring buffer
  - klogd pulls it out and sends it to syslogd

Example

```
Unable to handle kernel NULL pointer dereference at virtual address  0000001
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0
c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000
00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038
00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>]
[<c0108f77>] [<c010002b>]

Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

Safety

You can’t trust an oopsed kernel

The running kernel component was killed along with any userspace process without releasing locks or cleaning up structures

Decoding oopsen

a closer look at the oops dump...

```
Unable to handle kernel NULL pointer dereference at virtual address  0000001
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0
c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000
00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038
00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>]
[<c0108f77>] [<c010002b>]


Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

Decoding oopsen: fault

```
Unable to handle kernel NULL pointer dereference at virtual address  0000001
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0
c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000
00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038
00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>]
[<c0108f77>] [<c010002b>]


Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

(low address implies accessing a structure member)

Decoding oopsen: oops counter

```
Unable to handle kernel NULL pointer dereference at virtual address  0000001
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0
c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000
00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038
00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>]
[<c0108f77>] [<c010002b>]


Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

(often one oops triggers more, only the first is reliable)

Decoding oopsen: EIP

```
Unable to handle kernel NULL pointer dereference at virtual address  0000001
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0
c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000
00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038
00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>]
[<c0108f77>] [<c010002b>]


Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

(EIP shows the code segment and instruction address)

Decoding oopsen: EFLAGS + Registers

```
Unable to handle kernel NULL pointer dereference at virtual address  0000001
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0
c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000
00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038
00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>]
[<c0108f77>] [<c010002b>]

Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

(program status and control register, the general purpose registers and more segment registers)

Decoding oopsen: stack

```
Unable to handle kernel NULL pointer dereference at virtual address  0000001
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0
c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000
00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038
00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>]
[<c0108f77>] [<c010002b>]


Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

(stack, full of stuff, half run ops, return addresses...)

Decoding oopsen: call trace

```
Unable to handle kernel NULL pointer dereference at virtual address  0000001
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0
c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000
00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038
00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>]
[<c0108f77>] [<c010002b>]

Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

(claim to be return addresses, useless on their own)

Resolving addresses

(from oops output)

```
EIP:    0010:[<c017d558>]
```

(from System.map)

```
....
c017cdf0 T reiserfs_dir_fsync
c017ce80 t reiserfs_readdir
c017d2f0 t create_virtual_node
c017d780 t check_left
c017d8d0 t check_right
....
```

EIP = function base address + instruction offset

Decoding with ksymoops

```
...
Process tar (pid: 2293, stackpage=c6155000)
...

>>EIP; c017d558 <create_virtual_node+298/490>   <=====
Trace; c017eb4f <ip_check_balance+34f/ae0>
Trace; c017fc44 <reiserfs_kfree+14/50>
Trace; c0180115 <fix_nodes+115/450>
Trace; c018a1c8 <reiserfs_insert_item+88/110>
Trace; c017bb3a <reiserfs_new_inode+3da/500>
Trace; c018738f <pathrelse+1f/30>
Trace; c0177a13 <reiserfs_lookup+73/d0>
Trace; d0871044 <END_OF_CODE+9a77/???
Trace; c0178274 <reiserfs_mkdir+d4/1d0>
Trace; c0142e36 <d_alloc+16/160>
Trace; c013c75f <vfs_mkdir+7f/b0>
Trace; c013c7f8 <sys_mkdir+68/b0>
Trace; c0108f77 <system_call+33/38>
Trace; c010002b <startup_32+2b/139>

Code;  c017d558 <create_virtual_node+298/490>
00000000 <_EIP>:
Code;  c017d558 <create_virtual_node+298/490>   <=====
   0:   8b 40 14                  mov    0x14(%eax),%eax   <=====
Code;  c017d55b <create_virtual_node+29b/490>
   3:   ff d0                     call   *%eax
```

Decoding with klogd

it may also be saved in the system log, klogd also performs some symbol lookups

```
Jan 23 19:44:00 localhost kernel:  printing eip:
Jan 23 19:44:00 localhost kernel: c017d558
Jan 23 19:44:00 localhost kernel: *pde = 00000000
Jan 23 19:44:00 localhost kernel: Oops: 0000
Jan 23 19:44:00 localhost kernel: CPU:    0
Jan 23 19:44:00 localhost kernel: EIP:    0010:[create_virtual_node+664/1168]
Jan 23 19:44:00 localhost kernel: EFLAGS: 00210213
Jan 23 19:44:00 localhost kernel: eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
Jan 23 19:44:00 localhost kernel: esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
Jan 23 19:44:00 localhost kernel: ds: 0018   es: 0018   ss: 0018
Jan 23 19:44:00 localhost kernel: Process tar (pid: 2293, stackpage=c6155000)
Jan 23 19:44:00 localhost kernel: Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0 c6c79018
Jan 23 19:44:00 localhost kernel:        00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000 00000098
Jan 23 19:44:00 localhost kernel:        c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038 00000069
Jan 23 19:44:00 localhost kernel: Call Trace: [ip_check_balance+847/2784] [reiserfs_kfree+20/80] [fix_nodes+277/1104]
[reiserfs_insert_item+136/272] [reiserfs_new_inode+986/1280] [pathrelse+31/48] [reiserfs_lookup+115/208]
Jan 23 19:44:00 localhost kernel:        [<d0871044>] [reiserfs_mkdir+212/464] [d_alloc+22/352] [vfs_mkdir+127/176]
[sys_mkdir+104/176] [system_call+51/56] [stext+43/313]
Jan 23 19:44:00 localhost kernel:
Jan 23 19:44:00 localhost kernel: Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

Note: these offsets are in base10, not base16 like everywhere else!

part 1b

Basic Debugging

Disassembler basics

With the function name and offset we can work out where th
oops happened

- match the address with the assembler function
- match the function offset to the instruction
- match the assembler instructions to the C source

objdump

```
$ objdump -d fix_node.o
fix_node.o:     file format elf32-i386

Disassembly of section .text:

00000000 <create_virtual_node>:
       0:       55                      push   %ebp
       1:       57                      push   %edi
       2:       56                      push   %esi
       3:       53                      push   %ebx
       4:       83 ec 10                sub    $0x10,%esp
       7:       8b 44 24 24             mov    0x24(%esp,1),%eax
       b:       8b 5c 24 28             mov    0x28(%esp,1),%ebx
       f:       8b 50 10                mov    0x10(%eax),%edx
      12:       8b b0 90 01 00 00       mov    0x190(%eax),%esi
```

..... remember offset 298?

```
     25d:       b8 02 00 00 00          mov    $0x2,%eax
     262:       74 05                   je     269 <create_virtual_node+0x269>
     264:       b8 0f 00 00 00          mov    $0xf,%eax
     269:       89 c1                   mov    %eax,%ecx
     26b:       eb 10                   jmp    27d <create_virtual_node+0x27d>
     26d:       8d 76 00                lea    0x0(%esi),%esi
     270:       0f b6 4a 0f             movzbl 0xf(%edx),%ecx
     274:       c0 e9 04                shr    $0x4,%cl
     277:       0f b6 c1                movzbl %cl,%eax
     27a:       0f b7 c8                movzwl %ax,%ecx
     27d:       8b 5c 24 24             mov    0x24(%esp,1),%ebx
     281:       89 c8                   mov    %ecx,%eax
     283:       8b 04 85 00 00 00 00    mov    0x0(,%eax,4),%eax
     28a:       8b 8b 24 01 00 00       mov    0x124(%ebx),%ecx
     290:       51                      push   %ecx
     291:       8b 54 24 08             mov    0x8(%esp,1),%edx
     295:       52                      push   %edx
     296:       57                      push   %edi
     297:       56                      push   %esi
     298:       8b 40 14                mov    0x14(%eax),%eax
     29b:       ff d0                   call   *%eax
     29d:       89 c2                   mov    %eax,%edx
     29f:       8b 06                   mov    (%esi),%eax
     2a1:       83 c4 10                add    $0x10,%esp
     2a4:       01 c2                   add    %eax,%edx
```

Matching assembler to C

Things to look for:

- flow control... ifs, cases, whiles equate to tests, cmps and jumps
- returns equate to big jumps to near the end of the functions
- calling functions... pushing arguments onto the stack and making a call
- constants, ors, locks all stand out (asm/spinlock.h)
- use of pointers, arithmetic, memory manipulation

More than one way of doing things, compilers select different methods in different cases

It’s a bit of an art, you get used to it....

objdump (again)

```
$ objdump -d fix_node.o
fix_node.o:     file format elf32-i386

Disassembly of section .text:

00000000 <create_virtual_node>:
       0:       55                      push   %ebp
       1:       57                      push   %edi
       2:       56                      push   %esi
       3:       53                      push   %ebx
       4:       83 ec 10                sub    $0x10,%esp
       7:       8b 44 24 24             mov    0x24(%esp,1),%eax
       b:       8b 5c 24 28             mov    0x28(%esp,1),%ebx
       f:       8b 50 10                mov    0x10(%eax),%edx
      12:       8b b0 90 01 00 00       mov    0x190(%eax),%esi
```

..... remember offset 298?

```
     25d:       b8 02 00 00 00          mov    $0x2,%eax
     262:       74 05                   je     269 <create_virtual_node+0x269>
     264:       b8 0f 00 00 00          mov    $0xf,%eax
     269:       89 c1                   mov    %eax,%ecx
     26b:       eb 10                   jmp    27d <create_virtual_node+0x27d>
     26d:       8d 76 00                lea    0x0(%esi),%esi
     270:       0f b6 4a 0f             movzbl 0xf(%edx),%ecx
     274:       c0 e9 04                shr    $0x4,%cl
     277:       0f b6 c1                movzbl %cl,%eax
     27a:       0f b7 c8                movzwl %ax,%ecx
     27d:       8b 5c 24 24             mov    0x24(%esp,1),%ebx
     281:       89 c8                   mov    %ecx,%eax
     283:       8b 04 85 00 00 00 00    mov    0x0(,%eax,4),%eax
     28a:       8b 8b 24 01 00 00       mov    0x124(%ebx),%ecx
     290:       51                      push   %ecx
     291:       8b 54 24 08             mov    0x8(%esp,1),%edx
     295:       52                      push   %edx
     296:       57                      push   %edi
     297:       56                      push   %esi
     298:       8b 40 14                mov    0x14(%eax),%eax
     29b:       ff d0                   call   *%eax
     29d:       89 c2                   mov    %eax,%edx
     29f:       8b 06                   mov    (%esi),%eax
     2a1:       83 c4 10                add    $0x10,%esp
     2a4:       01 c2                   add    %eax,%edx
```

fs/reiserfs/fix_node.c

create_virtual_node()

```
/* go through all items those remain in the virtual node (except for the new (inserted) one) */
for (new_num = 0; new_num < vn->vn_nr_item; new_num ++) {
    int j;
    struct virtual_item * vi = vn->vn_vi + new_num;
    int is_affected = ((new_num != vn->vn_affected_item_num) ? 0 : 1);


    if (is_affected && vn->vn_mode == M_INSERT)
        continue;

    /* get item number in source node */
    j = old_item_num (new_num, vn->vn_affected_item_num, vn->vn_mode);

    vi->vi_item_len += ih[j].ih_item_len + IH_SIZE;
    vi->vi_ih = ih + j;
    vi->vi_item = B_I_PITEM (Sh, ih + j);
    vi->vi_uarea = vn->vn_free_ptr;

    // FIXME: there is no check, that item operation did not
    // consume too much memory
    vn->vn_free_ptr += op_create_vi (vn, vi, is_affected, tb->insert_size [0]);
    if (tb->vn_buf + tb->vn_buf_size < vn->vn_free_ptr)
        reiserfs_panic (tb->tb_sb, "vs-8030: create_virtual_node: "
                        "virtual node space consumed");
```

objdump for cheats :)

(with -g compiled kernel objects)

```
$ objdump --source -d fix_node.o
fix_node.o:     file format elf32-i386

Disassembly of section .text:

00000000 <create_virtual_node>:
  return new_num + 1;
}

static void create_virtual_node (struct tree_balance * tb, int h)
{
       0:       55                      push   %ebp
       1:       57                      push   %edi
       2:       56                      push   %esi
       3:       53                      push   %ebx
       4:       83 ec 10                sub    $0x10,%esp
    struct item_head * ih;
    struct virtual_node * vn = tb->tb_vn;
       7:       8b 44 24 24             mov    0x24(%esp,1),%eax
       b:       8b 5c 24 28             mov    0x28(%esp,1),%ebx

...

        // FIXME: there is no check, that item operation did not
        // consume too much memory
        vn->vn_free_ptr += op_create_vi (vn, vi, is_affected, tb->insert_size [0]);
     27d:       8b 5c 24 24             mov    0x24(%esp,1),%ebx
     281:       89 c8                   mov    %ecx,%eax
     283:       8b 04 85 00 00 00 00    mov    0x0(,%eax,4),%eax
     28a:       8b 8b 24 01 00 00       mov    0x124(%ebx),%ecx
     290:       51                      push   %ecx
     291:       8b 54 24 08             mov    0x8(%esp,1),%edx
     295:       52                      push   %edx
     296:       57                      push   %edi
     297:       56                      push   %esi
     298:       8b 40 14                mov    0x14(%eax),%eax
     29b:       ff d0                   call   *%eax
     29d:       89 c2                   mov    %eax,%edx
     29f:       8b 06                   mov    (%esi),%eax
```

GDB

what to expect...

```
(gdb) gdb vmlinux
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
(gdb) disassemble create_virtual_node
Dump of assembler code for function create_virtual_node:
0xc017d2f0 <create_virtual_node>:       push   %ebp
0xc017d2f1 <create_virtual_node+1>:     push   %edi
0xc017d2f2 <create_virtual_node+2>:     push   %esi
0xc017d2f3 <create_virtual_node+3>:     push   %ebx

......and so on

(gdb) info scope create_virtual_node
Scope for create_virtual_node:
Symbol tb is an argument at stack/frame offset 36, length 4.
Symbol h is an argument at stack/frame offset 40, length 4.
Symbol h is a local variable in register $ebx, length 4.
Symbol ih is a local variable at frame offset 12, length 4.
Symbol vn is a local variable in register $esi, length 4.
Symbol new_num is a local variable in register $ebp, length 4.
Symbol Sh is a local variable at frame offset 8, length 4.
(gdb)
```

Can run gdb on a running kernel
(include debugging symbols, that’s -g in CFLAGS)

```
> gdb -g vmlinux /proc/kcore
```

Hang on,

So, what went wrong?

it faulted on the mov (298)

```
     290:        51                        push   %ecx
     291:        8b 54 24 08               mov    0x8(%esp,1),%edx
     295:        52                        push   %edx
     296:        57                        push   %edi
     297:        56                        push   %esi
     298:        8b 40 14                  mov    0x14(%eax),%eax
     29b:        ff d0                     call   *%eax
```

but...the call was:

```
op_create_vi(vn, vi, is_affected, tb->insert_size [0]);
```

why is it loading EAX rather than calling the address directly?

The answer is: it’s another macro...

So what went wrong? Revisited

it faulted on the mov (298)

```
     290:       51                      push   %ecx
     291:       8b 54 24 08             mov    0x8(%esp,1),%edx
     295:       52                      push   %edx
     296:       57                      push   %edi
     297:       56                      push   %esi
     298:       8b 40 14                mov    0x14(%eax),%eax
     29b:       ff d0                   call   *%eax
```

```
 #define op_create_vi(vn,vi,is_affected,insert_size)
      item_ops[le_ih_k_type ((vi)->vi_ih)]->create_vi
(vn,vi,is_affected,insert_size)
```

It’s getting the address of the handler function from an array called item_ops but it’s basing the offset into that array on a pointer which could have failed without checking it.

Enough, please!

...

I’m afraid that I’ve seen too many people fix bugs by looking at debugger output, and that almost inevitably leads to fixing the symptoms rather than th underlying problems.

...

"Use the Source, Luke, use the Source. Be one with the code.". Think of Luke Skywalker discarding the automatic firing system when closing on the deathstar, and firing the proton torpedo (or whatever) manually. _Then_ do you have the right mindset for fixing kernel bugs.

...

Linus Torvalds

part 2

lockups

(all down hill from here!)

Lockups

Lockups are when the system just stops, no messages and doesn’t respond

Kinds of lockups:

- hardware lock ups
- lockups with interrupts enabled
- lockups without interrupts enabled

Hardware lockup

Tricky little area,

- Andrea Arcangeli’s print-EIP patch (part of IKD)
- Hardware monitoring tool (expensive!)

Lockups with IRQs enabled

Spinning in a loop? waiting on a lock?

Toggle keyboard lights, like caps-lock

The kernel is half alive,

On console, try the sysinfo keys

- shift-scroll lock   show memory
- control-scroll lock   show process state
- rightalt-scroll lock   show registers

Magic sysrq key

- usually ALT-PRINTSCREEN on x86
- BREAK on serial console
- often disabled on distributions
- echo "1" > /proc/sys/kernel/sysrq

It can:

sync data to disks	s
remount partitions read only	u
reboot the machine	b
power the machine off	o
turns off keyboard raw mode and puts it in XLATE	r
dump the cpu registers and flags	p
list tasks and some useful information about them	t
kills of a process on the current console	k
send SIGTERM to everything except init	e
send SIGKILL to everything except init	i
send SIGKILL to everything including init	l
change the console loglevel	0-9

Using the magic sysrq key

- use SYSRQ-P to dump cpu information a few times

```
SysRq: Show Regs

EIP: 0010:[<c011251f>] CPU: 0 EFLAGS: 00200296
EAX: 00005305 EBX: 00000000 ECX: 00000000 EDX: c147bfa0
ESI: 00000002 EDI: 00005305 EBP: 00000001 DS: 0018 ES: 0018
CR0: 8005003b CR2: 4001a000 CR3: 0cc66000 CR4: 00000690
Call Trace: [<c0112604>] [<c0112eb7>] [<c011389e>] [<c0113600>]
[<c010757b>]
```

- look to see how much the EIP changes
  - if it’s in a loop you can spot the small changes in the address
  - resolve the EIP like in an oops to find out where
  - deadlocks, lock ordering inversion

Lockups with IRQs disabled

- The NMI Watchdog
  - NMI = non-maskable interrupt
  - delivered whatever (usually...)
  - can detect when a cpu is locked up
  - NMI watchdog ticks every few seconds
  - when the system locks up it will automatically generate an oops
- Print EIP patch might help
- X Windows

Making bug reports

See REPORTING-BUGS file in kernel souce for a suggested bug-report format

Information required

- summarise the bug into a line or so
- use the short summary as the subject of the email
- have a more terse description of the bug
- decoded oops
- kernel version
- kernel configuration
- if you can describe how to reproduce the bug
- any other relevant information

Email the report to the maintainer of that system

- MAINTAINERS file
- if you don’t know who to send it to, sent it to the Linux Kernel Mailing list

Summary

- different lockups and oopsen
- collecting information
- decoding the raw data into something useful
- what to do with useful information

being able to apply this makes you useful!

If you want to go forward and learn more...

- understand the kernel
- learn C and Intel assembler
- experiment, play around, break things :)
- be patient, it takes time

Questions?

Further reading, references & links...

These slides will be published on:

- http://www.urbanmyth.org/linux

Intel CPU Documentation

- http://developer.intel.com/design/Pentium4/manuals/

The Linux Kernel Mailing list http://www.tux.org/lkml

Andrea Arcangeli’s IKD and Talk on Kernel Debugging

- ftp://ftp.kernel.org/pub/linux/people/andrea/ikd
- ftp://ftp.suse.com/pub/people/andrea/talks/

Kernel Debuggers

- KDB http://oss.sgi.com/projects/kdb
- KGDB http://kgdb.sourceforge.net