第一篇：定位 Oops 的具体代码行

(
来自 Linus Torvalds 的讨论:
[url]https://groups.google.com/group/linux.kernel/browse_thread/thread/b70bffe9015a8c41/ed9c0a0cfcd31111[/url]
又，[url]http://kerneltrap.org/Linux/Further_Oops_Insights[/url]
)

例如这样的一个 Oops：

Oops: 0000 [#1] PREEMPT SMP

Modules linked in: capidrv kernelcapi isdn slhc ipv6 loop dm_multipath snd_ens1371 gameport snd_rawmidi snd_ac97_codec ac97_bus snd_seq_dummy snd_seq_oss snd_seq_midi_event snd_seq snd_seq_device snd_pcm_oss snd_mixer_oss snd_pcm snd_timer snd parport_pc floppy parport pcnet32 soundcore mii pcspkr snd_page_alloc ac i2c_piix4 i2c_core button power_supply sr_mod sg cdrom ata_piix libata dm_snapshot dm_zero dm_mirror dm_mod BusLogic sd_mod scsi_mod ext3 jbd mbcache uhci_hcd ohci_hcd ehci_hcd

Pid: 1726, comm: kstopmachine Not tainted (2.6.24-rc3-module #2)

EIP: 0060:[<c04e53d6>] EFLAGS: 00010092 CPU: 0

EIP is at list_del+0xa/0x61

EAX: e0c3cc04 EBX: 00000020 ECX: 0000000e EDX: dec62000

ESI: df6e8f08 EDI: 000006bf EBP: dec62fb4 ESP: dec62fa4

DS: 007b ES: 007b FS: 00d8 GS: 0000 SS: 0068

Process kstopmachine (pid: 1726, ti=dec62000 task=df8d2d40 task.ti=dec62000)

Stack: 000006bf dec62fb4 c04276c7 00000020 dec62fbc c044ab4c dec62fd0 c045336c

df6e8f08 c04532b4 00000000 dec62fe0 c043deb0 c043de75 00000000 00000000

c0405cdf df6e8eb4 00000000 00000000 00000000 00000000 00000000

Call Trace:

[<c0406081>] show_trace_log_lvl+0x1a/0x2f
[<c0406131>] show_stack_log_lvl+0x9b/0xa3
[<c04061dc>] show_registers+0xa3/0x1df
[<c0406437>] die+0x11f/0x200
[<c0613cba>] do_page_fault+0x533/0x61a
[<c06123ea>] error_code+0x72/0x78
[<c044ab4c>] __unlink_module+0xb/0xf
[<c045336c>] do_stop+0xb8/0x108
[<c043deb0>] kthread+0x3b/0x63
[<c0405cdf>] kernel_thread_helper+0x7/0x10
=======================

Code: 6b c0 e8 2e 7e f6 ff e8 d1 16 f2 ff b8 01 00 00 00 e8 aa 1c f4 ff 89 d8 83 c4 10 5b 5d c3 90 90 90 55 89 e5 53 83 ec 0c 8b 48 04 <8b> 11 39 c2 74 18 89 54 24 08 89 44 24 04 c7 04 24 be 32 6b c0

EIP: [<c04e53d6>] list_del+0xa/0x61 SS:ESP 0068:dec62fa4

note: kstopmachine[1726] exited with preempt_count 1

1, 有自己编译的 vmlinux： 使用 gdb

编译时打开 **complie with debug info** 选项。

注意这行：

EIP is at list_del+0xa/0x61

这告诉我们，list_del 函数有 0x61 这么大，而 Oops 发生在 0xa 处。 那么我们先看一下 list_del 从哪里开始：

```
# grep list_del /boot/System.map-2.6.24-rc3-module
c10e5234 T plist_del
c10e53cc T list_del
c120feb6 T klist_del
c12d6d34 r __ksymtab_list_del
c12dadfc r __ksymtab_klist_del
c12e1abd r __kstrtab_list_del
c12e9d03 r __kstrtab_klist_del
```

于是我们知道，发生 Oops 时的 EIP 值是：

```
c10e53cc + 0xa  == c10e53d6
```

然后用 gdb 查看：

```
# gdb /home/arc/build/linux-2.6/vmlinux
(gdb) b *0xc10e53d6
Breakpoint 1 at 0xc10e53d6: file /usr/src/linux-2.6.24-rc3/lib/list_debug.c, line 64.
```

看，gdb 直接就告诉你在哪个文件、哪一行了。

gdb 中还可以这样：

```
# gdb Sources/linux-2.6.24/vmlinux
(gdb) l *do_fork+0x1f
0xc102b7ac is in do_fork (kernel/fork.c:1385).
1380
1381    static int fork_traceflag(unsigned clone_flags)
1382    {
1383            if (clone_flags & CLONE_UNTRACED)
1384                    return 0;
1385            else if (clone_flags & CLONE_VFORK) {
1386                    if (current->ptrace & PT_TRACE_VFORK)
1387                            return PTRACE_EVENT_VFORK;
1388            } else if ((clone_flags & CSIGNAL) != SIGCHLD) {
1389                    if (current->ptrace & PT_TRACE_CLONE)
(gdb)
```

也可以直接知道 line number。

或者：

```
(gdb) l *(0xffffffff8023eaf0 + 0xff)  /* 出错函数的地址加上偏移 */
```

2, 没有自己编译的 vmlinux： TIPS

如果在 lkml 或 bugzilla 上看到一个 Oops，而自己不能重现，那就只能反汇编以"Code:"开始的行。 这样可以尝试定位到
源代码中。

注意，Oops 中的 Code:行，会把导致 Oops 的第一条指令，也就是 EIP 的值的第一个字节， 用尖括号<>括起来。 但是，有些
体系结构(例如常见的 x86)指令是不等长的(不一样的指令可能有不一样的长度)，所以要不断的尝试(trial-and-error)。

Linus 通常使用一个小程序，类似这样：

```
const char array[] = "\xnn\xnn\xnn...";
int main(int argc, char *argv[])
{
        printf("%p\n", array);
        *(int *)0 = 0;
}
```

e.g. /*{{{*/ /* 注意， array 一共有从 array[0]到 array[64]这 65 个元素， 其中出错的那个操作码<8b> == arry[43] */

```
#include <stdio.h>
#include <stdlib.h>


const char array[]
="\x6b\xc0\xe8\x2e\x7e\xf6\xff\xe8\xd1\x16\xf2\xff\xb8\x01\x00\x00\x00\xe8\xaa\
x1c\xf4\xff\x89\xd8\x83\xc4\x10\x5b\x5d\xc3\x90\x90\x90\x55\x89\xe5\x53\x83\x
ec\x0c\x8b\x48\x04\x8b\x11\x39\xc2\x74\x18\x89\x54\x24\x08\x89\x44\x24\x04\
xc7\x04\x24\xbe\x32\x6b\xc0";
int main(int argc, char *argv[])
{
        printf("%p\n", array);
        *(int *)0 = 0;
```

```
}
/*}}}*/
```

用 gcc -g 编译，在 gdb 里运行它：

```
[arc@dhcp-cbjs05-218-251 ~]$ gdb hello
GNU gdb Fedora (6.8-1.fc9)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<[url]http://gnu.org/licenses/gpl.html[/url]>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
(no debugging symbols found)
(gdb) r
Starting program: /home/arc/hello
0x80484e0

Program received signal SIGSEGV, Segmentation fault.
```

注意，这时候就可以反汇编 0x80484e0 这个地址：

```
(gdb) disassemble 0x80484e0
Dump of assembler code for function array:
0x080484e0 <array+0>:   imul   $0xfffffff8,%eax,%eax
0x080484e3 <array+3>:   jle,pn 0x80484dc <__dso_handle+20>
0x080484e6 <array+6>:   ljmp   *<internal disassembler error>
0x080484e8 <array+8>:   rcll   (%esi)
0x080484ea <array+10>:  repnz (bad)
0x080484ec <array+12>:  mov    $0x1,%eax
0x080484f1 <array+17>:  call   0x7f8a1a0
0x080484f6 <array+22>:  mov    %ebx,%eax
0x080484f8 <array+24>:  add    $0x10,%esp
0x080484fb <array+27>:  pop    %ebx
```

```
0x080484fc <array+28>:  pop    %ebp
0x080484fd <array+29>:  ret
0x080484fe <array+30>:  nop
0x080484ff <array+31>:  nop
0x08048500 <array+32>:  nop
0x08048501 <array+33>:  push   %ebp
0x08048502 <array+34>:  mov    %esp,%ebp
0x08048504 <array+36>:  push   %ebx
0x08048505 <array+37>:  sub    $0xc,%esp
0x08048508 <array+40>:  mov    0x4(%eax),%ecx
0x0804850b <array+43>:  mov    (%ecx),%edx
0x0804850d <array+45>:  cmp    %eax,%edx
0x0804850f <array+47>:  je     0x8048529
0x08048511 <array+49>:  mov    %edx,0x8(%esp)
0x08048515 <array+53>:  mov    %eax,0x4(%esp)
0x08048519 <array+57>:  movl   $0xc06b32be,(%esp)
0x08048520 <array+64>:  add    %ah,0xa70
End of assembler dump.
(gdb)
```

OK, 现在你知道出错的那条指令是 array[43]，也就是 mov (%ecx),%edx，也就是说，(%ecx)指向了一个错误内存地址。

补充：

为了使汇编代码和 C 代码更好的对应起来， Linux 内核的 Kbuild 子系统提供了这样一个功能： 任何一个 C 文件都可以单独编译成汇编文件，例如：

make path/to/the/sourcefile.s

例如我想把 kernel/sched.c 编译成汇编，那么：

make kernel/sched.s V=1

或者：

make kernel/sched.lst V=1

编译出*.s 文件

有时侯需要对*.s 文件进行分析，以确定 BUG 所在的位置。 对任何一个内核 build 目录下的*.c 文件，都可以

直接编译出*.s 文件。

```
# make drivers/net/e100.s V=1
```

而对于自己写的 module，就需要在 Makefile 中有一个灵活的 target 写法：

```
# cat Makefile
obj-m := usb-skel.o
KDIR := /lib/modules/`uname -r`/build
traget := modules

default:
	make -C $(KDIR) M=$(shell pwd) $(target)
clean:
	rm -f *.o *.ko .*.cmd *.symvers *.mod.c
	rm -rf .tmp_versions


# make target=usb-skel.s V=1
```

这样，kbuild 系统才知道你要 make 的目标不是 modules，而是 usb-skel.s。

另外， 内核源代码目录的./scripts/decodecode 文件是用来解码 Oops 的：

```
./scripts/decodecode < Oops.txt
```

第二篇：定位可动态加载的内核模块的 OOPS 代码行

1. 从 vmlinux 获取具体的代码行

文 章中 albcamus 版主也提到了，需要有自己编译的 vmlinux，而且编译时打开 compile with debug info. 这个选项打开之后会使 vmlinux 文件比不加调试信息大一些。我这里代调试信息的是 49M。建议如果学习的时候，想使用 gdb 的方式获取出错代码行的 话，就加上这个编译条件。
然后就可以按照具体的方法去操作，可以定位到具体的 C 代码行。

2. 从自己编译的内核模块出错信息中获取代码行
以 ldd3 中提供的 misc-modules/faulty.c 为例。主要以 faulty_write 函数作分析。
（1）由于作者提供的函数代码就一样，过于简单，我这里简单加上一些代码（也就是判断和赋值），如下：

```
ssize_t faulty_write (struct file *filp, const char __user
*buf, size_t count,
                loff_t *pos)
{
        /* make a simple fault by dereferencing a NULL
pointer */
        if(count > 0x100)
                count = 0x100;
        *(int *)0 = 0;
        return count;
}
```

（2）编译该模块，并且 mknod /dev/faulty
（3）向该模块写入数据：echo 1 > /dev/faulty, 内核 OOPS，信息如下：

```
<1>BUG: unable to handle kernel NULL pointer dereference at
virtual address 00000000
printing eip:
f8e8000e
*pde = 00000000
```

```
Oops: 0002 [#3]
SMP
Modules linked in: faulty autofs4 hidp rfcomm l2cap ...... //
此处省略若干字符

CPU: 1
EIP: 0060:[<f8e8000e>] Not tainted VLI
EFLAGS: 00010283 (2.6.18.3 #2)
EIP is at faulty_write+0xe/0x19 [faulty]
eax: 00000001 ebx: f4f6ca40 ecx: 00000001 edx: b7c2d000
esi: f8e80000 edi: b7c2d000 ebp: 00000001 esp: f4dc5f84
ds: 007b es: 007b ss: 0068
Process bash (pid: 6084, ti=f4dc5000 task=f7c8d4d0
task.ti=f4dc5000)
Stack: c1065914 f4dc5fa4 f4f6ca40 fffffff7 b7c2d000 f4dc5000
c1065f06 f4dc5fa4
       00000000 00000000 00000000 00000001 00000001 c1003d0b
00000001 b7c2d000
       00000001 00000001 b7c2d000 bfd40aa8 ffffffda 0000007b
c100007b 00000004
Call Trace:
[<c1065914>] vfs_write+0xa1/0x143
[<c1065f06>] sys_write+0x3c/0x63
[<c1003d0b>] syscall_call+0x7/0xb
Code: Bad EIP value.
EIP: [<f8e8000e>] faulty_write+0xe/0x19 [faulty] SS:ESP
0068:f4dc5f84
```

其中，我们应该关注的信息是第一行红色标出部分：告诉我们操作了 NULL 指针。其次，就是第二行红色部分：EIP is at faulty_write+0xe/0x19。这个出错信息告诉我们 EIP 指针出现问题的地方时 faulty_write 函数，而且指出了是 faulty 模块。
同时，faulty_write+0xe/0x19 的后半部分 0xe/0x19，说明该函数的大小时 0x019，出错位置是在 0x0e。这两个值应该值得都是汇编代码的值。

（4）将 faulty 模块反汇编出汇编代码：

```
objdump -d faulty.ko > faulty.s
```

或

```
objdump -d faulty.o > faulty.s
```

然后，我们打开 faulty.s 文件。由于我们需要关注的部分正好在文件的前面，因此我这里只贴出文件的前面一部分内容：

```
faulty.o:  file format elf32-i386


Disassembly of section .text:


00000000 <faulty_write>:
   0:  81 f9 00 01 00 00  cmp $0x100,%ecx
   6:  b8 00 01 00 00  mov $0x100,%eax
   b:  0f 46 c1  cmovbe %ecx,%eax
   e:  c7 05 00 00 00 00 00  movl $0x0,0x0
  15:  00 00 00
  18:  c3  ret


00000019 <cleanup_module>:
  19:  a1 00 00 00 00  mov 0x0,%eax
  1e:  ba 00 00 00 00  mov $0x0,%edx
  23:  e9 fc ff ff ff  jmp 24 <cleanup_module+0xb>


00000028 <faulty_init>:
  28:  a1 00 00 00 00  mov 0x0,%eax
  2d:  b9 00 00 00 00  mov $0x0,%ecx
  32:  ba 00 00 00 00  mov $0x0,%edx
  37:  e8 fc ff ff ff  call 38 <faulty_init+0x10>
  3c:  85 c0  test %eax,%eax
  3e:  78 13  js 53 <faulty_init+0x2b>
```

```
  40: 83 3d 00 00 00 00 00 cmpl $0x0,0x0
  47: 74 03 je 4c <faulty_init+0x24>
  49: 31 c0 xor %eax,%eax
  4b: c3 ret
  4c: a3 00 00 00 00 mov %eax,0x0
  51: 31 c0 xor %eax,%eax
  53: c3 ret
```

由以上汇编代码可以看出，faulty_write 函数的大小确实是 0x18 -0x00 +1 = 0x19. 那么 EIP 指针出问题的地方是 0x0e 处，代码为：

```
e: c7 05 00 00 00 00 00 movl $0x0,0x0
```

这行汇编代码就是将 0 值保存到 0 地址的位置。那么很显然是非法的。这一行对应的 C 代码应该就是：

```
*(int *)0 = 0;
```

（5）以上是对模块出错信息的分析。不过也有一定的局限。

首先就是 EIP 出错的位置正好在本模块内部，这样可以在本模块定位问题；

其次，要求一定的汇编基础，特别是当一个函数的代码比较多时，对应的汇编代码也比较大，如何准确定位到 C 代码行需要一定的经验和时间。

实际运用中，可以将内核代码行的定位和可动态加载的内核模块代码行的定位结合起来使用，应该可以较快的定位问题。