NOTE: ksymoops is useless on 2.6. Please use the Oops in its original format (from dmesg, etc). Ignore any references in this or other docs to "decoding the Oops" or "running it through ksymoops". If you post an Oops from 2.6 that has been run through ksymoops, people will just tell you to repost it.

Quick Summary

Find the Oops and send it to the maintainer of the kernel area that seems to be involved with the problem. Don't worry too much about getting the wrong person. If you are unsure send it to the person responsible for the code relevant to what you were doing. If it occurs repeatably try and describe how to recreate it. That's worth even more than the oops.

If you are totally stumped as to whom to send the report, send it to linux-kernel@vger.kernel.org. Thanks for your help in making Linux as stable as humanly possible.

Where is the Oops?

Normally the Oops text is read from the kernel buffers by klogd and handed to syslogd which writes it to a syslog file, typically /var/log/messages (depends on /etc/syslog.conf). Sometimes klogd dies, in which case you can run dmesg > file to read the data from the kernel buffers and save it. Or you can cat /proc/kmsg > file, however you have to break in to stop the transfer, kmsg is a "never ending file". If the machine has crashed so badly that you cannot enter commands or the disk is not available then you have three options :-

(1) Hand copy the text from the screen and type it in after the machine has restarted. Messy but it is the only option if you have not planned for a crash. Alternatively, you can take a picture of the screen with a digital camera - not nice, but better than nothing. If the messages scroll off the top of the console, you may find that booting with a higher resolution (eg, vga=791) will allow you to read more of the text. (Caveat: This needs vesafb, so won't help for 'early' oopses)

(2) Boot with a serial console (see Documentation/serial-console.txt), run a null modem to a second machine and capture the output there using your favourite communication program. Minicom works well.

(3) Use Kdump (see Documentation/kdump/kdump.txt), extract the kernel ring buffer from old memory with using dmesg gdbmacro in Documentation/kdump/gdbmacros.txt.

Full Information

NOTE: the message from Linus below applies to 2.4 kernel. I have preserved it for historical reasons, and because some of the information in it still applies. Especially, please ignore any references to ksymoops.

From: Linus Torvalds <torvalds@osdl.org>

How to track down an Oops.. [originally a mail to linux-kernel]

The main trick is having 5 years of experience with those pesky oops
messages ;-)

Actually, there are things you can do that make this easier. I have two
separate approaches:

```
gdb /usr/src/linux/vmlinux
gdb> disassemble <offending_function>
```

That's the easy way to find the problem, at least if the bug-report is
well made (like this one was - run through ksymoops to get the
information of which function and the offset in the function that it
happened in).

Oh, it helps if the report happens on a kernel that is compiled with the
same compiler and similar setups.

The other thing to do is disassemble the "Code:" part of the bug report:
ksymoops will do this too with the correct tools, but if you don't have
the tools you can just do a silly program:

```
char str[] = "\xXX\xXX\xXX...";
main(){}
```

and compile it with gcc -g and then do "disassemble str" (where the "XX"
stuff are the values reported by the Oops - you can just cut-and-paste
and do a replace of spaces to "\x" - that's what I do, as I'm too lazy
to write a program to automate this all).

Alternatively, you can use the shell script in scripts/decodecode.
Its usage is: decodecode < oops.txt

The hex bytes that follow "Code:" may (in some architectures) have a series
of bytes that precede the current instruction pointer as well as bytes at and
following the current instruction pointer. In some cases, one instruction
byte or word is surrounded by <> or (), as in "<86>" or "(f00d)". These
<> or () markings indicate the current instruction pointer. Example from
i386, split into multiple lines for readability:

Code: f9 0f 8d f9 00 00 00 8d 42 0c e8 dd 26 11 c7 a1 60 ea 2b f9 8b 50 08 a1
64 ea 2b f9 8d 34 82 8b 1e 85 db 74 6d 8b 15 60 ea 2b f9 <8b> 43 04 39 42 54
7e 04 40 89 42 54 8b 43 04 3b 05 00 f6 52 c0

Finally, if you want to see where the code comes from, you can do

```
cd /usr/src/linux
make fs/buffer.s        # or whatever file the bug happened in
```

and then you get a better idea of what happens than with the gdb
disassembly.

Now, the trick is just then to combine all the data you have: the C
sources (and general knowledge of what it _should_ do), the assembly

listing and the code disassembly (and additionally the register dump you also get from the "oops" message - that can be useful to see _what_ the corrupted pointers were, and when you have the assembler listing you can also match the other registers to whatever C expressions they were used for).

Essentially, you just look at what doesn't match (in this case it was the "Code" disassembly that didn't match with what the compiler generated). Then you need to find out _why_ they don't match. Often it's simple - you see that the code uses a NULL pointer and then you look at the code and wonder how the NULL pointer got there, and if it's a valid thing to do you just check against it..

Now, if somebody gets the idea that this is time-consuming and requires some small amount of concentration, you're right. Which is why I will mostly just ignore any panic reports that don't have the symbol table info etc looked up: it simply gets too hard to look it up (I have some programs to search for specific patterns in the kernel code segment, and sometimes I have been able to look up those kinds of panics too, but that really requires pretty good knowledge of the kernel just to be able to pick out the right sequences etc..)

Sometimes it happens that I just see the disassembled code sequence from the panic, and I know immediately where it's coming from. That's when I get worried that I've been doing this for too long ;-)

Linus

Notes on Oops tracing with klogd:

In order to help Linus and the other kernel developers there has been substantial support incorporated into klogd for processing protection faults. In order to have full support for address resolution at least version 1.3-pl3 of the sysklogd package should be used.

When a protection fault occurs the klogd daemon automatically translates important addresses in the kernel log messages to their symbolic equivalents. This translated kernel message is then forwarded through whatever reporting mechanism klogd is using. The protection fault message can be simply cut out of the message files and forwarded to the kernel developers.

Two types of address resolution are performed by klogd. The first is static translation and the second is dynamic translation. Static translation uses the System.map file in much the same manner that ksymoops does. In order to do static translation the klogd daemon must be able to find a system map file at daemon initialization time. See the klogd man page for information on how klogd searches for map files.

Dynamic address translation is important when kernel loadable modules are being used. Since memory for kernel modules is allocated from the kernel's dynamic memory pools there are no fixed locations for either the start of the module or for functions and symbols in the module.

The kernel supports system calls which allow a program to determine which modules are loaded and their location in memory. Using these system calls the klogd daemon builds a symbol table which can be used to debug a protection fault which occurs in a loadable kernel module.

At the very minimum klogd will provide the name of the module which generated the protection fault. There may be additional symbolic information available if the developer of the loadable module chose to export symbol information from the module.

Since the kernel module environment can be dynamic there must be a mechanism for notifying the klogd daemon when a change in module environment occurs. There are command line options available which allow klogd to signal the currently executing daemon that symbol information should be refreshed. See the klogd manual page for more information.

A patch is included with the sysklogd distribution which modifies the modules-2.0.0 package to automatically signal klogd whenever a module is loaded or unloaded. Applying this patch provides essentially seamless support for debugging protection faults which occur with kernel loadable modules.

The following is an example of a protection fault in a loadable module processed by klogd:

```
------------------------------------------------------------------------------
Aug 29 09:51:01 blizard kernel: Unable to handle kernel paging request at
virtual address f15e97cc
Aug 29 09:51:01 blizard kernel: current->tss.cr3 = 0062d000, %cr3 = 0062d000
Aug 29 09:51:01 blizard kernel: *pde = 00000000
Aug 29 09:51:01 blizard kernel: Oops: 0002
Aug 29 09:51:01 blizard kernel: CPU:    0
Aug 29 09:51:01 blizard kernel: EIP:    0010:[oops:_oops+16/3868]
Aug 29 09:51:01 blizard kernel: EFLAGS: 00010212
Aug 29 09:51:01 blizard kernel: eax: 315e97cc   ebx: 003a6f80   ecx: 001be77b
edx: 00237c0c
Aug 29 09:51:01 blizard kernel: esi: 00000000   edi: bffffdb3   ebp: 00589f90
esp: 00589f8c
Aug 29 09:51:01 blizard kernel: ds: 0018   es: 0018   fs: 002b   gs: 002b   ss:
0018
Aug 29 09:51:01 blizard kernel: Process oops_test (pid: 3374, process nr: 21,
stackpage=00589000)
Aug 29 09:51:01 blizard kernel: Stack: 315e97cc 00589f98 0100b0b4 bffffed4
0012e38e 00240c64 003a6f80 00000001
Aug 29 09:51:01 blizard kernel:        00000000 00237810 bfffff00 0010a7fa
00000003 00000001 00000000 bfffff00
Aug 29 09:51:01 blizard kernel:        bffffdb3 bffffed4 ffffffda 0000002b
0007002b 0000002b 0000002b 00000036
Aug 29 09:51:01 blizard kernel: Call Trace: [oops:_oops_ioctl+48/80]
[_sys_ioctl+254/272] [_system_call+82/128]
Aug 29 09:51:01 blizard kernel: Code: c7 00 05 00 00 00 eb 08 90 90 90 90 90 90
90 90 89 ec 5d c3
------------------------------------------------------------------------------
```

Roger Maris Cancer Center INTERNET: greg@wind.rmcc.com
820 4th St. N.
Fargo, ND 58122
Phone: 701-234-7556

Tainted kernels:

Some oops reports contain the string 'Tainted: ' after the program counter. This indicates that the kernel has been tainted by some mechanism. The string is followed by a series of position-sensitive characters, each representing a particular tainted value.

1: 'G' if all modules loaded have a GPL or compatible license, 'P' if any proprietary module has been loaded. Modules without a MODULE_LICENSE or with a MODULE_LICENSE that is not recognised by insmod as GPL compatible are assumed to be proprietary.

2: 'F' if any module was force loaded by "insmod -f", ' ' if all modules were loaded normally.

3: 'S' if the oops occurred on an SMP kernel running on hardware that hasn't been certified as safe to run multiprocessor. Currently this occurs only on various Athlons that are not SMP capable.

4: 'R' if a module was force unloaded by "rmmod -f", ' ' if all modules were unloaded normally.

5: 'M' if any processor has reported a Machine Check Exception, ' ' if no Machine Check Exceptions have occurred.

6: 'B' if a page-release function has found a bad page reference or some unexpected page flags.

7: 'U' if a user or user application specifically requested that the Tainted flag be set, ' ' otherwise.

8: 'D' if the kernel has died recently, i.e. there was an OOPS or BUG.

9: 'A' if the ACPI table has been overridden.

10: 'W' if a warning has previously been issued by the kernel. (Though some warnings may set more specific taint flags.)

11: 'C' if a staging driver has been loaded.

12: 'I' if the kernel is working around a severe bug in the platform firmware (BIOS or similar).

The primary reason for the 'Tainted: ' string is to tell kernel debuggers if this is a clean kernel or if anything unusual has occurred. Tainting is permanent: even if an offending module is unloaded, the tainted value remains to indicate that the kernel is not trustworthy.