LINUX KERNEL

OOPSES AND DEBUGGING SYSTEM FAULTS

Important Notice : Courseware - Legal

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here.

Oops Messages

Most bugs show themselves in NULL pointer dereferences or by the use of other incorrect pointer values. The usual outcome of such bugs is an *oops* message.

Almost any address used by the processor is a virtual address and is mapped to physical addresses through a complex structure of page tables (the exceptions (to this rule) are physical addresses used within the memory management subsystem itself). When an invalid pointer is dereferenced, the paging mechanism fails to map the pointer to a physical address, and the processor signals a page fault to the operating system. If the address is not valid, the kernel is not able to “page in” the missing address; it (usually) generates an oops if this happens while the processor is in supervisor mode.

An oops displays the processor status at the time of the fault, including the contents of the CPU registers and other seemingly incomprehensible information. The message is generated by printk statements in the fault handler (*arch/*/kernel/traps.c*) and is dispatched as described earlier in the section “printk.”

Oops 1 :: Generating a (trivial) Oops dump

<< The below session run is on a VMware VMX VM running a Fedora 21 Guest on a Ubuntu 14.10 host system >>

```
/*
 * oops.c
 * Silly demo - make it Oops!
 * Kaiwan NB, kaiwanTECH
 */

#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>

MODULE_LICENSE("Dual BSD/GPL");
char *ptr = NULL; /* checkpatch says: ERROR: do not initialise globals to NULL */

static int __init oops2_init(void)
{
	pr_info("Hello, about to Oops!\n");
	*(ptr + 0x20) = 'a';
	return 0;
}

static void __exit oops2_exit(void)
{
	pr_info("Goodbye\n");
}
```

```
module_init(oops2_init);
module_exit(oops2_exit);

# make && insmod ./oops1.ko ; dmesg
[…]

[  444.059244] oops1_init:16 : #1Hello, about to Oops!
[  444.059262] BUG: unable to handle kernel NULL pointer dereference at           (null)
[  444.059264] IP: [<ffffffffa0005035>] oops1_init+0x35/0x1000 [oops1]
[  444.059267] PGD 14fb7067 PUD 3705c067 PMD 0
[  444.059269] Oops: 0002 [#1] SMP
[  444.059292] Modules linked in: oops1(OE+) bnep bluetooth fuse nf_conntrack_netbios_ns
nf_conntrack_broadcast ip6t_rpfilter ip6t_REJECT xt_conntrack cfg80211 rfkill ebtable_nat
ebtable_broute bridge stp llc ebtable_filter ebtables ip6table_nat nf_conntrack_ipv6 nf_defrag_ipv6
nf_nat_ipv6 ip6table_mangle ip6table_security ip6table_raw ip6table_filter ip6_tables iptable_nat
nf_conntrack_ipv4 nf_defrag_ipv4 nf_nat_ipv4 nf_nat nf_conntrack iptable_mangle iptable_security
iptable_raw snd_ens1371 gameport snd_rawmidi snd_ac97_codec ac97_bus snd_seq snd_seq_device snd_pcm
ppdev coretemp crct10dif_pclmul crc32_pclmul crc32c_intel ghash_clmulni_intel vmw_balloon serio_raw
snd_timer snd soundcore vmw_vmci i2c_piix4 shpchp parport_pc parport vmwgfx drm_kms_helper ttm drm
mptspi scsi_transport_spi mptscsih e1000
[  444.059312]  mptbase ata_generic pata_acpi
[  444.059315] CPU: 1 PID: 3057 Comm: insmod Tainted: G           OE  3.17.4-301.fc21.x86_64 #1
[  444.059316] Hardware name: VMware, Inc. VMware Virtual Platform/440BX Desktop Reference
Platform, BIOS 6.00 07/31/2013
[  444.059317] task: ffff880016b96bf0 ti: ffff880014ff0000 task.ti: ffff880014ff0000
[  444.059318] RIP: 0010:[<ffffffffa0005035>]  [<ffffffffa0005035>] oops1_init+0x35/0x1000 [oops1]
[  444.059320] RSP: 0018:ffff880014ff3d10  EFLAGS: 00010282
[  444.059320] RAX: 0000000000000027 RBX: ffffffff81c18040 RCX: 0000000000000027
[  444.059321] RDX: 0000000000000000 RSI: ffff88003f62e6f8 RDI: ffff88003f62e6f8
[  444.059322] RBP: ffff880014ff3d10 R08: 0000000000000092 R09: 000000000000062f
[  444.059322] R10: ffffffff81002138 R11: 000000000000062f R12: ffff88003737dfc0
[  444.059323] R13: 0000000000000000 R14: ffffffffa0005000 R15: 0000000000000001
[  444.059324] FS:  00007fe72e173700(0000) GS:ffff88003f620000(0000) knlGS:0000000000000000
[  444.059325] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  444.059326] CR2: 0000000000000000 CR3: 000000003b9f6000 CR4: 00000000000407e0
[  444.059369] Stack:
[  444.059370]  ffff880014ff3d88 ffffffff81002148 0000000000000001 ffffffffa040a090
[  444.059371]  0000000000000025 ffff88003b811180 0000000000000001 ffff880014ff3d70
[  444.059373]  ffffffff811d2442 ffff880014ff3ef0 000000000c684f4 ffff880014ff3ef0
[  444.059374] Call Trace:
[  444.059380]  [<ffffffff81002148>] do_one_initcall+0xd8/0x210
[  444.059383]  [<ffffffff811d2442>] ? __vunmap+0xa2/0x100
[  444.059388]  [<ffffffff81116fcf>] load_module+0x1ebf/0x2640
[  444.059389]  [<ffffffff811129e0>] ? store_uevent+0x70/0x70
[  444.059393]  [<ffffffff81212507>] ? kernel_read+0x57/0x90
[  444.059395]  [<ffffffff81117916>] SyS_finit_module+0xa6/0xe0 << finit_module syscall has loaded
the lkm >>
[  444.059399]  [<ffffffff81746ae9>] system_call_fastpath+0x16/0x1b
[  444.059400] Code: c7 20 a0 40 a0 48 89 e5 e8 f9 2d 38 e1 85 c0 74 1a ba 10 00 00 00 48 c7 c6 6b
90 40 a0 48 c7 c7 28 90 40 a0 31 c0 e8 26 91 73 e1 <c7> 04 25 00 00 00 00 7b 00 00 00 31 c0 5d c3
00 00 00 00 00 00
[  444.059413] RIP  [<ffffffffa0005035>] oops1_init+0x35/0x1000 [oops1]
```

```
[  444.059414]  RSP <ffff880014ff3d10>
[  444.059415] CR2: 0000000000000000
[  444.059416] ---[ end trace a4a3f32751d0c337 ]---
#
```

SIDEBAR | What do the '?' preceding the stack frames mean?

Short answer: the kernel figures that these are probably just "blips" left behind from old frames – just ignore them, the kernel is almost always right about this (being an incorrect stack frame).

Long(er) answer: see Kernel stacks on x86-64 bit *section "Printing backtraces on x86"*.

```
…
We always scan the full kernel stack for return addresses stored on
the kernel stack(s) [*], from stack top to stack bottom, and print out
anything that 'looks like' a kernel text address.

If it fits into the frame pointer chain, we print it without a question
mark, knowing that it's part of the real backtrace.

If the address does not fit into our expected frame pointer chain we
still print it, but we print a '?'. It can mean two things:

 - either the address is not part of the call chain: it's just stale
   values on the kernel stack, from earlier function calls. This is
   the common case.

 - or it is part of the call chain, but the frame pointer was not set
   up properly within the function, so we don't recognize it.

This way we will always print out the real call chain (plus a few more
entries), regardless of whether the frame pointer was set up correctly
or not - but in most cases we'll get the call chain right as well. The
entries printed are strictly in stack order, so you can deduce more
information from that as well.

The most important property of this method is that we _never_ lose
information: we always strive to print _all_ addresses on the stack(s)
that look like kernel text addresses, so if debug information is wrong,
we still print out the real call chain as well - just with more question
marks than ideal


…

<< This (above) is the 'GUESS_UNWINDER' >>
```

It's a well known fact that allowing the compiler to use frame pointers (CONFIG_FRAME_POINTERS), makes for accurate stack backtraces, and hence, easier debugging.

However, frame pointers are considered expensive – at least two assembly instructions required for every function; thus, they're ussually turned Off on production systems. But this leaves us with "if-fy" stack traces.

A feature on the horizon (with the 4.13 kernel) - *ORC Unwinder* !
There's a new menu item under *'Kernel Hacking': 'Choose kernel unwinder'*:

From ***arch/x86/Kconfig.debug***:

--snip--

```
choice
      prompt "Choose kernel unwinder"
      default FRAME_POINTER_UNWINDER
      ---help---
        This determines which method will be used for unwinding kernel stack
        traces for panics, oopses, bugs, warnings, perf, /proc/<pid>/stack,
        livepatch, lockdep, and more.

config FRAME_POINTER_UNWINDER
      bool "Frame pointer unwinder"
      select FRAME_POINTER
      ---help---
        This option enables the frame pointer unwinder for unwinding kernel
        stack traces.

        The unwinder itself is fast and it uses less RAM than the ORC
        unwinder, but the kernel text size will grow by ~3% and the kernel's
        overall performance will degrade by roughly 5-10%.

        This option is recommended if you want to use the livepatch
        consistency model, as this is currently the only way to get a
        reliable stack trace (CONFIG_HAVE_RELIABLE_STACKTRACE).

config ORC_UNWINDER
      bool "ORC unwinder"
      depends on X86_64
      select STACK_VALIDATION
      ---help---
        This option enables the ORC (Oops Rewind Capability) unwinder for
        unwinding kernel stack traces.  It uses a custom data format which
       is a simplified version of the DWARF Call Frame Information
       standard.

        This unwinder is more accurate across interrupt entry frames than
       the frame pointer unwinder.  It also enables a 5-10% performance
        improvement across the entire kernel compared to frame pointers.

        Enabling this option will increase the kernel's runtime memory usage
        by roughly 2-4MB, depending on your kernel config.
```

```
config GUESS_UNWINDER
      bool "Guess unwinder"
      depends on EXPERT
      ---help---
        This option enables the "guess" unwinder for unwinding kernel stack
        traces.  It scans the stack and reports every kernel text address it
        finds.  Some of the addresses it reports may be incorrect.

        While this option often produces false positives, it can still be
        useful in many cases.  Unlike the other unwinders, it has no runtime
        overhead.

endchoice

config FRAME_POINTER
      depends on !ORC_UNWINDER && !GUESS_UNWINDER
      bool
```

--*snip*--

Some details in the LWN article
The ORCs are coming, LWN, July 2017

The presence of debug symbols in the module does not depend much on the `EXTRA_CFLAGS / ccflags-y += -Og` in the module Makefile but rather on the kernel config:

if CONFIG_DEBUG_INFO=y
then
   it's a 'debug' kernel
   debug info is present...
else
   no debug info...
fi

(On ARM-64, 6.6 kernel, I found CONFIG_DEBUG_INFO_NONE=y implying no kernel debug info (Yocto build).)

Panic-on-Oops is always set on production systems:

```
$ cat <kernel-src>/lib/Kconfig.debug
[...]
config PANIC_ON_OOPS
        bool "Panic on Oops"
        help
          Say Y here to enable the kernel to panic when it oopses. This
          has the same effect as setting oops=panic on the kernel command
```

```
          line.

          This feature is useful to ensure that the kernel does not do
          anything erroneous after an oops which could result in data
          corruption or other issues.

          Say N if unsure.
[...]
$
```

On a generic dekstop Linux

```
$ grep . /proc/sys/kernel/panic*
/proc/sys/kernel/panic:0
/proc/sys/kernel/panic_on_io_nmi:0
/proc/sys/kernel/panic_on_oops:0
/proc/sys/kernel/panic_on_rcu_stall:0
/proc/sys/kernel/panic_on_unrecovered_nmi:0
/proc/sys/kernel/panic_on_warn:0
/proc/sys/kernel/panic_print:0
$
```

SIDEBAR | What actually happens when derefercing a NULL pointer? - on Quora. Answer below by Robert Love.
I dislike answering these sorts of questions as there are so many different answers and the question is underspecified. As Tim Wilson noted, the correct answer is "that is undefined." Full stop. Nonetheless, I was asked to answer this, so I'll discuss *what actually happens* on a modern system such as Linux. I think the underlying mechanics are what you are trying to get at. First, some preliminaries: NULL is numerically zero. That is, NULL == 0 is always true. C and C++ allow, however, NULL 's internal representation to differ from zero as dictated by a specific implementation. It is up to the compiler then to translate zero to the proper internal representation and vice versa. For the rest of this answer, we'll assume NULL's internal representation and its numerical representation are both zero. That is true on nearly any system you will ever work on, including Linux/x86. So to your question: What actually happens when you dereference NULL? I'll repeat the previous caveat: Doing so is undefined, which means *literally anything can happen*. Undefined doesn't mean "crash" and it doesn't mean (as many people think) "implementation specific." It means anything—and maybe something different every time—can happen. Nonetheless, the behavior on Linux is rather consistent: *Segfault*. How that happens is both simple and elegant: 1. The page starting at virtual address 0x0 is mapped into every user-space process on Linux with no access permissions (by mapping it, the kernel ensures nothing else will ever be mapped there). 2. Linux compilers treat the internal representation of NULL as zero and happily let you dereference the pointer.

3. The page fault handler is invoked as the page at 0x0 is not resident.
4. The page fault handler notices your operation isn't allowed (as none are).
5. The page fault handler fails, refusing to fault in the page.
6. The kernel sends the process << 'current' actually >> a SIGSEGV , for which the default behavior is process termination plus core generation. In other words, you crash.

Thus, without any special compiler or kernel support except a page mapped at 0x0 , Linux provides the expected behavior for a NULL dereference.

Oops 2 :: Generating another (quite trivial) Oops (on x86_64)

```
/*
 * oops_simple.c
 * Slightly less Silly demo - make it Oops!
 * This time, by dereferencing a member of an invalid (null) structure pointer..
 *
 * Kaiwan NB, kaiwanTECH
 */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>

MODULE_LICENSE("Dual MIT/GPL");

struct faker {
  long longs[7];
  char dumb[7];
  long bad_cache_align;
};
struct faker *f1;

static int __init oops2_init(void)
{
#if 0 // uncomment this code to get rid of the silly bug   :-p
      f1 = kmalloc(sizeof(struct faker), GFP_KERNEL);
      if (!f1) {
        pr_warn("kmalloc f1 failed");
          return -ENOMEM;
        }
      pr_info("sizeof(long) = %d, sizeof(struct faker) = %lu, actual space alloced
= %lu\n",
             sizeof(long), sizeof(struct faker), ksize(f1));
#endif
      pr_info("Hello, about to Oops!\n");
      f1->bad_cache_align = 0xabcd;

      return 0;
}
```

```
static void __exit oops2_exit(void)
{
#if 0
	kfree(f1);
#endif
	pr_info("Goodbye\n");
}

module_init(oops_simple_init);
module_exit(oops_simple_exit);
```

```
# make && insmod ./oops_simple.ko ; dmesg
[…]
# insmod ./oops_simple.ko ; dmesg
Killed
[22578.606691] Hello, about to Oops!
[22578.606696] BUG: kernel NULL pointer dereference, address: 0000000000000040
<< Note the k va isn't 0 >> *
[22578.606698] #PF: supervisor write access in kernel mode
[22578.606699] #PF: error_code(0x0002) - not-present page
[22578.606700] PGD 0 P4D 0
[22578.606702] Oops: 0002 [#2] PREEMPT SMP PTI  << 0002 is a bitmask: see discussion below >>
[22578.606704] CPU: 2 PID: 20955 Comm: insmod Tainted: G      D    OE      6.1.25-lkp-kernel #7
[22578.606706] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[22578.606708] RIP: 0010:oops2_init+0x1c/0x1000 [oops_simple]
[22578.606712] Code: Unable to access opcode bytes at 0xffffffffc0625ff2.
[22578.606713] RSP: 0018:ffffb76044ffbbd8 EFLAGS: 00010246
--snip--
```

<<

Interpreting the LSB 0, 1 and 2 bit – error code – upon page fault:

!!! NOTE !!!   Bitmask interpretation is very arch (cpu) - specific

arch/x86/mm/fault.c

```
/*
 * Page fault error code bits:
 *
 *   bit 0 ==    0: no page found   1: protection fault
 *   bit 1 ==    0: read access     1: write access
 *   bit 2 ==    0: kernel-mode access  1: user-mode access
 *   bit 3 ==               1: use of reserved bit detected
 *   bit 4 ==               1: fault was an instruction fetch
 */
```

Interpret the LSB 3 bits

```
   Bit 2    |  Bit 1 |    Bit 0
0  K-mode    |  read  |  no page found
1  U-mode    |  write |  protection fault
```

Oops bitmask value		*Interpretation of LSB 3 bits Bit 2, Bit 1, Bit 0*
Decimal	*Binary*	
0000	000	Kernel-mode, read, no page found
0001	001	Kernel-mode, read, protection fault
0002	010	Kernel-mode, write, no page found
0003	011	Kernel-mode, write, protection fault
0004	100	User-mode, read, no page found
0005	101	User-mode, read, protection fault
0006	110	User-mode, write, no page found
0007	111	User-mode, write, protection fault

Thus 0002 = **0010** binary => Kernel-mode, write, no page found !
[#1] => # of times this Oops occurred.

(See below for intepretation on ARM-32)
>>

```
...
[22578.606702] Oops: 0002 [#2] PREEMPT SMP PTI
[22578.606704] CPU: 2 PID: 20955 Comm: insmod Tainted: G      D    OE      6.1.25-lkp-kernel #7
[22578.606706] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[22578.606708] RIP: 0010:oops2_init+0x1c/0x1000 [oops_simple]
[22578.606712] Code: Unable to access opcode bytes at 0xffffffffc0625ff2.
[22578.606713] RSP: 0018:ffffb76044ffbbd8 EFLAGS: 00010246
[22578.606715] RAX: 0000000000000000 RBX: 0000000000000000 RCX: 0000000000000000
[22578.606716] RDX: 0000000000000000 RSI: 0000000000000000 RDI: 0000000000000000
[22578.606717] RBP: ffffb76044ffbbd8 R08: 0000000000000000 R09: 0000000000000000
[22578.606718] R10: 0000000000000000 R11: 0000000000000000 R12: ffffffffc0626000
[22578.606719] R13: ffff9f2721883410 R14: ffffb76044ffbdd0 R15: ffffffffc08a8040
[22578.606721] FS:  00007ff406aa8c40(0000) GS:ffff9f277fc80000(0000) knlGS:0000000000000000
[22578.606722] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[22578.606723] CR2: ffffffffc0625ff2 CR3: 0000000016b0e005 CR4: 00000000000706e0
[22578.606727] Call Trace:
[22578.606728]  <TASK>
[22578.606730]  do_one_initcall+0x49/0x230
[22578.606734]  ? kmalloc_trace+0x2a/0xb0
[22578.606737]  do_init_module+0x52/0x210
[22578.606740]  load_module+0x1fa6/0x2460
```

```
[22578.606743]  __do_sys_finit_module+0xcc/0x150
[22578.606745]  ? __do_sys_finit_module+0xcc/0x150
[22578.606748]  __x64_sys_finit_module+0x18/0x30
[22578.606749]  do_syscall_64+0x5c/0x90
[22578.606752]  ? exit_to_user_mode_prepare+0x34/0x190
[22578.606754]  ? syscall_exit_to_user_mode+0x2a/0x50
[22578.606756]  ? do_syscall_64+0x69/0x90
[22578.606758]  ? irqentry_exit+0x43/0x50
[22578.606760]  ? exc_page_fault+0x92/0x1b0
[22578.606762]  entry_SYSCALL_64_after_hwframe+0x63/0xcd
[22578.606764] RIP: 0033:0x7ff40631ea3d
[22578.606765] Code: 5b 41 5c c3 66 0f 1f 84 00 00 00 00 00 f3 0f 1e fa 48 89 f8 48 89 f7 48 89 d6
48 89 ca 4d 89 c2 4d 89 c8 4c 8b 4c 24 08 0f 05 <48> 3d 01 f0 ff ff 73 01 c3 48 8b 0d c3 a3 0f 00
f7 d8 64 89 01 48
[22578.606767] RSP: 002b:00007ffe53e48be8 EFLAGS: 00000246 ORIG_RAX: 0000000000000139
[22578.606769] RAX: ffffffffffffffda RBX: 000055da671f27f0 RCX: 00007ff40631ea3d
[22578.606770] RDX: 0000000000000000 RSI: 000055da65770cd2 RDI: 0000000000000003
[22578.606771] RBP: 0000000000000000 R08: 0000000000000000 R09: 0000000000000000
[22578.606772] R10: 0000000000000003 R11: 0000000000000246 R12: 000055da65770cd2
[22578.606773] R13: 000055da671f2780 R14: 000055da6576f888 R15: 000055da671f2900
[22578.606775]  </TASK>
[22578.606776] Modules linked in: oops_simple(OE+) oops(OE+) hello(O) vboxvideo(OE) vboxsf(OE)
intel_rapl_msr binfmt_misc vmwgfx snd_intel8x0 snd_ac97_codec ac97_bus intel_rapl_common snd_pcm
crct10dif_pclmul crc32_pclmul ghash_clmulni_intel aesni_intel crypto_simd cryptd snd_seq
drm_kms_helper rapl snd_timer snd_seq_device syscopyarea sysfillrect input_leds sysimgblt
fb_sys_fops joydev snd drm_ttm_helper ttm serio_raw vboxguest(OE) soundcore mac_hid sch_fq_codel
min_sysinfo(OE) drm msr parport_pc ppdev lp parport ramoops pstore_blk reed_solomon pstore_zone
efi_pstore ip_tables x_tables autofs4 hid_generic usbhid hid psmouse ahci e1000 libahci pata_acpi
i2c_piix4 [last unloaded: hello(O)]
[22578.606804] CR2: 0000000000000040
[22578.606805] ---[ end trace 0000000000000000 ]---
#
```

**** The faulting kernel virtual address is non-zero, and a small quantity (0x40 = 64). This hints at it as likely being an offset to a structure (or other) pointer, where the base pointer is null!***

Also notice:

```
arch/x86/mm/fault.c
static void
show_fault_oops(struct pt_regs *regs, unsigned long error_code, << the OOPS dumping code
>>
        unsigned long address) << 'address' is the faulting address –
                                 this address when accessed caused a page fault to occur! >>
{
...
 printk(KERN_ALERT "BUG: unable to handle kernel ");
    if (address < PAGE_SIZE)
        printk(KERN_CONT "NULL pointer dereference");
    else
        printk(KERN_CONT "paging request");

    printk(KERN_CONT " at %p\n", (void *) address);
```

```
    printk(KERN_ALERT "IP:");
    printk_address(regs->ip);
...
}
```

Approach 1 : Use objdump when a kernel module Oops'es:

This approach does assume that kernel debug information is builtin to the module; IOW, CONFIG_DEBUG_INFO=y (among other kernel debug configs)!

(*F.e. with our oops_simple buggy LKM, in another run)*

```
sudo insmod ./oops_simple.ko && lsmod|grep oops_simple
------------------------------
<...>/lkm: line 18: 27111 Killed                  sudo insmod ./oops_simple.ko
 ^--[FAILED]
------------------------------
dmesg
------------------------------
[118231.996485] oops_simple_init:45 : Hello, about to Oops!
[118231.998261] BUG: kernel NULL pointer dereference, address: 0000000000000040
[118232.000127] #PF: supervisor write access in kernel mode
[118232.002867] #PF: error_code(0x0002) - not-present page
[118232.004096] PGD 0 P4D 0
[118232.004635] Oops: 0002 [#1] SMP PTI
[118232.005482] CPU: 1 PID: 27112 Comm: insmod Tainted: P           OE     5.4.0-
llkd01 #2
[118232.007097] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox
12/01/2006
[118232.008482] RIP: 0010:oops2_init+0x3f/0x1000 [oops_simple]
[118232.009282] Code: 89 e5 e8 e4 b5 21 f4 85 c0 74 18 ba 2d 00 00 00 48 c7 c6 88
90 59 c0 48 c7 c7 28 90 59 c0 e8 3b 6a 96 f3 48 8b 05 c1 d3 ff ff <48> c7 40 40 cd
ab 00 00 31 c0 5d c3 00 00 00 00 00 00 00 00 00 00
[118232.012292] RSP: 0018:ffffbe7d41d63c60 EFLAGS: 00010286
[...]
```

Use objdump like this:

```
$ grep "oops_simple" /proc/modules
oops_simple 24576 1 - Loading 0x0000000000000000 (OE+)      << security, no info-
leak >>
$ sudo grep "oops_simple" /proc/modules
oops_simple 24576 1 - Loading 0xffffffffc0598000 (OE+)
$
$ objdump -dS --adjust-vma=0xffffffffc0598000 ./oops_simple.ko    << accurate
disassembly/source intermix; specifying the VMA has objdump display the kernel
virtual addresses too ! >>
./oops_simple.ko:     file format elf64-x86-64
```

```
Disassembly of section .init.text:

ffffffffc0598000 <init_module>:
  long bad_cache_align; // use 63+8=71 bytes, thus spilling over the cacheline!
Very naughty.
};
struct faker *f1;

static int __init oops2_init(void)
{
ffffffffc0598000: e8 00 00 00 00          callq  ffffffffc0598005 <init_module+0x5>
ffffffffc0598005: 55                      push   %rbp
          return -ENOMEM;
        }
[...]
```

Also, the key line here is:

```
RIP: 0010:oops2_init+0x3f/0x1000 [oops_simple]
```

So, we have to find the line of code @ `oops2_init+0x3f`. Lets use the above *objdump* output carefully:

```
...
static int __init oops2_init(void)
{
ffffffffc0598000:    e8 00 00 00 00       callq  ffffffffc0598005 <init_module+0x5>
   << this kva, ffffffffc0598000, is the start addr of oops2_init() >>
ffffffffc0598005:    55                   push   %rbp
         return -ENOMEM;
...

Now
ffffffffc0598000 + 0x3f = ffffffffc059803f
```

Here it is – closest match:

```
ffffffffc0598033:    e8 00 00 00 00       callq  ffffffffc0598038 <init_module+0x38>
       f1->bad_cache_align = 0xabcd;       << so this is the line that caused the Oops! >>
ffffffffc0598038:    48 8b 05 00 00 00 00 mov    0x0(%rip),%rax    # ffffffffc059803f
<init_module+0x3f>
ffffffffc059803f:    48 c7 40 40 cd ab 00 movq   $0xabcd,0x40(%rax)
ffffffffc0598046:    00
...
```

Approach 2 : A simpler way with objdump

Lets say this the Oops output:

```
...
```

```
 RIP: 0010:oops_init+0x1c/0x1000 [oops_simple]
...


$ objdump -dS ./oops_simple.ko
...
...
<< Look for the offset 0x1c into the releavnt function >>

static int __init oops2_init(void)
{
   0: e8 00 00 00 00          call   5 <init_module+0x5>
   5: 55                      push   %rbp                   this is the offset
      if (!f1)
            return -ENOMEM;
      pr_info("sizeof(long) = %ld, sizeof(struct faker) = %lu, actual space alloced = 
%lu\n",
            sizeof(long), sizeof(struct faker), ksize(f1));
#else
      pr_info("Hello, about to Oops!\n");
   6: 48 c7 c7 00 00 00 00    mov    $0x0,%rdi
{
   d: 48 89 e5                mov    %rsp,%rbp
      pr_info("Hello, about to Oops!\n");
  10: e8 00 00 00 00          call   15 <init_module+0x15> This is the nearest line of C code corr
#endif                                                       to this offset. Perfect.
      f1->bad_cache_align = 0xabcd;
  15: 48 8b 05 00 00 00 00    mov    0x0(%rip),%rax       # 1c <init_module+0x1c>
  1c: 48 c7 40 40 cd ab 00    movq   $0xabcd,0x40(%rax)
  23: 00
      return 0;
}
```

(Well, actually, it's the code following the C code shown – the assembly – that, when it runs, causes the bug and thus the Oops to occur…)

Approach 3:
Much simpler with addrline!

`addr2line [-e filename|--exe=filename] [...] [addr addr ...]`

```
$ addr2line -e ./oops_simple.ko 0x1c
/home/osboxes/linux-6.1.38/./arch/x86/include/asm/current.h:15
```

Whooops! addr2line didn't work here!

<< It does work at times… :-)

```
$ addr2line -e ./oops_simple.ko 0x3f
<...>/oops_simple.c:46
$
```

Line #46 is:

```
f1->bad_cache_align = 0xabcd;
```

Perfect!

>>

So, if it doesn't work?

Approach 4:
Use GDB !

```
(gdb) list *<func>+<offset>
```

```
oops_simple $ gdb -q ./oops_simple.ko
Reading symbols from ./oops_simple.ko...
(gdb) list *oops2_init+0x1c
0xe8 is in oops2_init (/home/osboxes/kaiwanTECH/L5_debug_trg/kernel_debug/k_oo
ps_warn_panic/oops_simple/oops_simple.c:32).
27              pr_info("sizeof(long) = %ld, sizeof(struct faker) = %lu, actua
l space alloced = %lu\n",
28                  sizeof(long), sizeof(struct faker), ksize(f1));
29      #else
30              pr_info("Hello, about to Oops!\n");
31      #endif
32              f1->bad_cache_align = 0xabcd;
33              return 0;
34      }
35
36      static void __exit oops2_exit(void)
(gdb)
```

Approach 5:
Now, with faddr2line!

<recent-kernel-src-tree>/scripts/faddr2line

```
$ ~/6.1.8/scripts/faddr2line
usage: faddr2line [--list] <object file> <func+offset> <func+offset>...
$
$ ~/6.1.8/scripts/faddr2line ./oops_simple.ko oops2_init+0x1c
oops2_init+0x1c/0x2e:
oops2_init at /home/osboxes/kaiwanTECH/L5_debug_trg/kernel_debug/k_oops_warn_panic/oops
_simple/oops_simple.c:32
$
```

PRO TIPS

Use faddr2line in place of addr2line when KASLR's enabled (usually is by default).

From my Linux Kernel Debugging book:
...

So, let's appropriately invoke the `faddr2line` script:

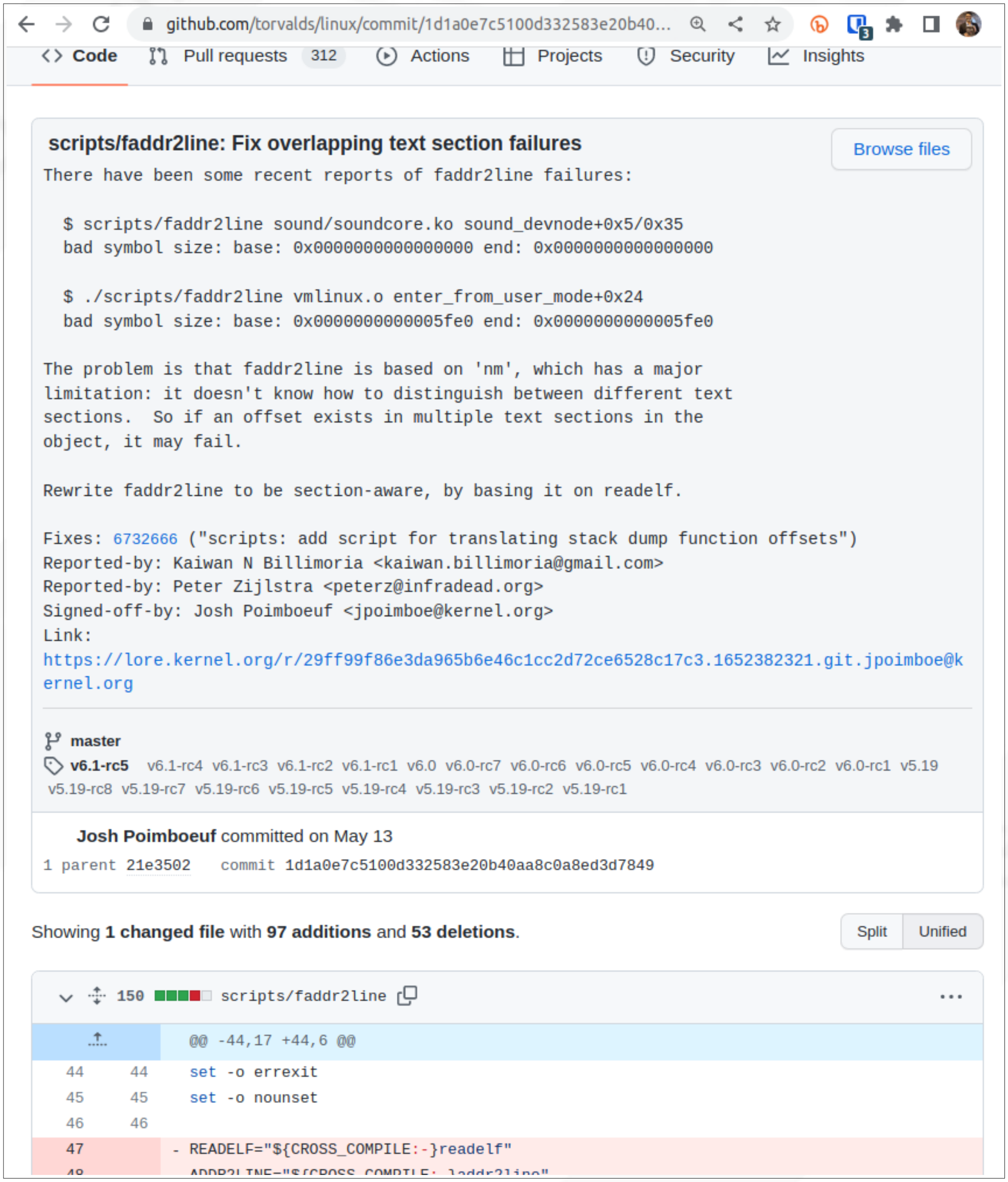
```
$ ~/lkd_kernels/productionk/linux-5.10.60/scripts/faddr2line ./
oops_tryv2.ko do_the_work+0x124
bad symbol size: base: 0x0000000000000000 end:
0x0000000000000000
```

Hey, that's really not what we expected!

> **Tip – Patch the faddr2line Script or Use a Newer Fixed Version**
> Upon encountering this issue with `faddr2line`, I reported it to the maintainer, Josh Poimboeuf (link: `https://lkml.org/lkml/2022/1/16/305`). By May 2022, Josh had fixed it (the underlying issue was that the `nm` utility wasn't good enough; he switched to using `readelf` – you'll find the details in the patch: `https://lore.kernel.org/lkml/29ff99f86e3da965b6e46c1cc2d72ce6528c17c3.1652382321.git.jpoimboe@kernel.org/`). So, until this fix hits the upcoming mainline kernel (it will, and I am hoping it happens soon – as of this writing, the process is just getting started), you'll have to manually apply this patch to the existing `scripts/faddr2line` script. (The fixed faddr2line should make it into the 5.19 kernel.)

Original email thread
…
Patch for 5.10

The **actual fix** (commit ID 1d1a0e7c5100d332583e20b40aa8c0a8ed3d7849 dt 13 May 2022 on):
https://github.com/torvalds/linux/commit/1d1a0e7c5100d332583e20b40aa8c0a8ed3d7849



"…

Once the patch (mentioned just above) is applied, or, you have a fixed version of the faddr2line script from a later kernel source tree (this should definitely be the case soon enough), let's retry:

```
$ <...>/scripts/faddr2line ./oops_tryv2.ko do_the_work+0x124/0x15e
do_the_work at <...>/Linux-Kernel-Debugging/ch7/oops_tryv2/
oops_tryv2.c:62
```

Ah, that's perfect! Line 62 (oopsie->data = 'x';) is indeed the buggy one.”

<<
A driver bug when testing on the BeagleBone Black (TI AM335x):
here, the kernel’s an older one: 4.19.94-ti-r74; hence, faddr2line from the kernel headers won’t work…
So, simply scp in a later (>=5.19) kernel tree’s faddr2line script; it works!

```
...
bbb $ <...>/faddr2line ../drv_rwmem/devmem_rw.ko rwmem_ioctl+0x4d8/0x648
rwmem_ioctl+0x4d8/0x648:
__raw_writel at
/usr/src/linux-headers-4.19.94-ti-r74/./arch/arm/include/asm/io.h:100
(inlined by) iowrite32 at /usr/src/linux-headers-4.19.94-ti-r74/./include/asm-
generic/io.h:745
(inlined by) rwmem_ioctl at /home/debian/.../devmem_rw.c:228
bbb $
```
>>

TIPS:
- You can always disable KASLR by passing the parameter `nokaslr` to the kernel at boot.

- On an AArch64:

```
$ /usr/src/kernel/scripts/faddr2line ./oops_simple.ko oops2_init+0x30/0xff8
skipping oops2_init address at 0x38 due to size mismatch (0xff8 != 0x40)
no match for oops2_init+0x30/0xff8
kenix-raspberrypi4-64-dca632f7ca5a oops_simple $
```

Oops, it failed…
Try just giving the offset..

```
$ /usr/src/kernel/scripts/faddr2line ./oops_simple.ko oops2_init+0x30
oops2_init+0x30/0x40:
oops2_init at /home/.../k_oops_warn_panic/oops_simple/oops_simple.c:32
```

`Perfect.`

objdump, GDB and faddr2line often succeed straight away; however, they might (likely) require the presence of symbols in the binary…

From my LKD book:

Taking advantage of kernel scripts to help debug kernel issues

The modern Linux kernel has many helper scripts, helping you to debug kernel bugs. Here's a quick table summarizing them. A bit of a more detailed take on how to practically use them follows:

Script	Purpose
`scripts/checkstack.pl`	Estimates the stack size used by functions within the kernel (or module), in descending order by size.
`scripts/decode_stacktrace.sh`	A script that tries to convert all kernel (virtual) addresses passed to it (usually by redirecting standard input from the `dmesg` output or piped to it), to source filenames with line numbers.
`scripts/decodecode`	A script that attempts to add useful information by parsing the `Code: <...machine code bytes...>` line in a typical Oops report; figures out and specifies the instruction where the fault actually occurred and shows it as `<-- trapping instruction` to the right of that line.
`scripts/faddr2line`	The same as `addr2line` but appropriate for use on systems using KASLR (for security) and for interpreting stack dumps from kernel modules. (Note – use the most recent version, as earlier versions were flawed; the upcoming *Exploiting the faddr2line script on KASLR systems* section has the details.)
`tools/debugging/kernel-chktaint`	Interprets the kernel tainted flags. Can pass the tainted bitmask as a parameter. If not passed, it looks up the current system taint state and prints its report.
`scripts/get_maintainer.pl`	With the `-f file|directory` parameter, this script identifies and prints details on the maintainer(s), the mailing list, and so on. It's useful to quickly find who maintains a given piece of code in the kernel!

Table 7.3 – A summary table of several useful kernel helper scripts

The list isn't exhaustive but is plenty to work with. Let's get going!

Interpreting the Kernel Code at Oops Time

Use the *scripts/decodecode* shell script.

Eg. running our simple oops_simple.ko LKM on an ARM-32:

```
ARM # insmod oops_simple.ko
oops_simple: loading out-of-tree module taints kernel.
oops_simple_init:45 : Hello, about to Oops!
Unhandled fault: page domain fault (0x81b) at 0x00000024
pgd = 9ee8c000
[00000024] *pgd=7ee3d831, *pte=00000000, *ppte=00000000
Internal error: : 81b [#1] SMP ARM
Modules linked in: oops_simple(O+)
CPU: 0 PID: 743 Comm: insmod Tainted: G           O    4.9.1 #4
Hardware name: ARM-Versatile Express
task: 9f53b980 task.stack: 9ee7a000
PC is at oops2_init+0x50/0x5c [oops_simple]
LR is at console_unlock+0x5b0/0x624
pc : [<7f002050>]    lr : [<8016ef0c>]    psr: 60040013
sp : 9ee7bd90  ip : 9ee7bc80  fp : 9ee7bd9c
r10: 80a03008  r9 : 9ee43ca4  r8 : 00000001
r7 : ffffe000  r6 : 7f002000  r5 : 00000000  r4 : 80a03008
r3 : 00000000  r2 : 0000abcd  r1 : 60040013  r0 : 00000000
Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment none
Control: 10c5387d  Table: 7ee8c059  DAC: 00000051
Process insmod (pid: 743, stack limit = 0x9ee7a210)
Stack: (0x9ee7bd90 to 0x9ee7c000)
bd80:                                     9ee7be1c 9ee7bda0 80101d34 7f00200c
bda0: 9ee43ca4 80a03008 9ee7bdcc 9ee7bdb8 806c1364 806c09e0 00000001 024000c0
bdc0: 9ee7bde4 9ee7bdd0 806c13c8 806c1354 9f401f00 024000c0 9ee7be1c 9ee7bde8
bde0: 8024e46c 806c1388 00000001 00000017 9ee43d80 00040901 7f000140 00000001
be00: 7f000140 9ee43d80 00000001 9ee43ca4 9ee7be44 9ee7be20 801ff884 80101cdc
be20: 9ee7be44 9ee7be30 9ee7bf34 00000001 7f000140 9ee43c80 9ee7bf2c 9ee7be48
be40: 8019f188 801ff81c 7f00014c 00007fff 7f000140 8019c5e4 a533cfff a533d000
be60: 807bb7c8 807bb7a0 80a03008 807bb8f0 807bb794 00000000 80703534 7f00014c
be80: 00000000 7f000188 8019c02c 8019bf64 9ee7beb4 9ee7bea0 8023f4d0 8023e7c0
bea0: 00000017 9ee43d80 00000000 00000000 00000000 00000000 00000000 00000000
bec0: 6e72656b 00006c65 00000000 00000000 00000000 00000000 00000000 00000000
bee0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00040901
bf00: 024002c2 000063ec 000d9474 a533c3ec ffffe000 000b0948 00000000 00000051
bf20: 9ee7bfa4 9ee7bf30 8019f918 8019d4d0 80a59598 a5326000 000163ec a533bdfc
bf40: a533bc70 a5337024 00000344 00000394 00000000 00000000 00000000 000005c0
bf60: 00000023 00000024 00000011 00000000 0000000e 00000000 801083c4 000163ec
bf80: 756e694c 00000078 00000080 801083c4 9ee7a000 00000000 00000000 9ee7bfa8
bfa0: 80108220 8019f7c8 000163ec 756e694c 000c3088 000163ec 000b0948 76e65048
bfc0: 000163ec 756e694c 00000078 00000080 00000001 7ee46e3c 76fea000 0009e29a
bfe0: 7ee46b00 7ee46af0 0002808f 76ee9a82 80040030 000c3088 7fffd861 7fffdc61
[<7f002050>] (oops2_init [oops_simple]) from [<80101d34>]
(do_one_initcall+0x64/0x1ac)
[<80101d34>] (do_one_initcall) from [<801ff884>] (do_init_module+0x74/0x1e4)
```

```
[<801ff884>] (do_init_module) from [<8019f188>] (load_module+0x1cc4/0x22f8)
[<8019f188>] (load_module) from [<8019f918>] (SyS_init_module+0x15c/0x17c)
[<8019f918>] (SyS_init_module) from [<80108220>] (ret_fast_syscall+0x0/0x1c)
Code: e30a2bcd e3473f00 e3a00000 e5933000 (e5832024)
---[ end trace 2ee827bca99f88fb ]---
Segmentation fault
ARM #
```

Collect the Oops text into an ASCII text file. Then:

```
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- scripts/decodecode <
            <...>/kernel_oopses/simple_oops2_arm32.txt
Code: e30a2bcd e3473f00 e3a00000 e5933000 (e5832024)
All code
========
   0: e3 0a 2b cd       movw  r2, #43981  ; 0xabcd
   4: e3 47 3f 00       movt  r3, #32512  ; 0x7f00
   8: e3 a0 00 00       mov   r0, #0
   c: e5 93 30 00       ldr   r3, [r3]
  10: *e5 83 20 24      str   r2, [r3, #36]     ; 0x24            <-- trapping
instruction

Code starting with the faulting instruction
===========================================
   0: e5832024    str   r2, [r3, #36]     ; 0x24
$
```

Eg. Oops on Android 4.4.2 IMX6dl: https://community.nxp.com/thread/464888

Can also use some scripts and utils : see the doc on *Kernel Debugging – Tips and Tricks.*

Additional Notes / FAQs

- *How does one see the disassembly?*

```
$ gdb <path/to/>vmlinux
...
(gdb) set disassembly-flavor intel   << do this on an x86 or x86_64 >>
(gdb) disas ip_rcv
Dump of assembler code for function ip_rcv:
   0xc14f0870 <+0>:     push   %ebp
   0xc14f0871 <+1>:     mov    %esp,%ebp
   0xc14f0873 <+3>:     sub    $0x20,%esp
   0xc14f0876 <+6>:     mov    %ebx,-0xc(%ebp)
   0xc14f0879 <+9>:     mov    %esi,-0x8(%ebp)
```

```
   0xc14f087c <+12>:    mov    %edi,-0x4(%ebp)
   0xc14f087f <+15>:    call   0xc15c7c80 <mcount>
```

…

- *How does one see the C source code pertaining to the point where the Oops occurred?*
    - *DO:*
      *(gdb) list *<func_name>+offset*

```
Eg.
(gdb) list *ip_rcv+0xf6
<< why 0xf6 ? Because that's the offset reported in the Oops..
...  EIP is at udp_rcv+0xf6/0x5e0   ...
>>
0xc14f0966 is in ip_rcv (net/ipv4/ip_input.c:460).
455         IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INHDRERRORS);
456   drop:
457         kfree_skb(skb);
458   out:
459         return NET_RX_DROP;
460   }
(gdb)
```

- *(Similar to above, but with objdump)*
  How exactly can one disassemble the kernel image file for a given function (by name)?

  Just use *objdump* on the uncompressed kernel image built with debugging enabled (CONFIG_DEBUG_KERNEL=y) – this essentially uses the compiler's `-g` flag. Do as follows:

```
${CROSS_COMPILE}objdump -d -S <path/to/kernel-src>/vmlinux[-ver#] >
  vmlinux-debug[-ver#].disas
-d, --disassemble   : Display assembler contents of executable sections
-S, --source        : Intermix source code with disassembly
```

Very useful: can use this to locate exact kernel (virtual) addresses, symbol names, etc! (grep-ping this file can help).

Similarly, for kernel modules:
Compile with (in the Makefile):

```
[...]
 $(info Building with ARCH=${ARCH}, KERNELRELEASE=${KERNELRELEASE},
CROSS_COMPILE=${CROSS_COMPILE} EXTRA_CFLAGS=${EXTRA_CFLAGS})
ccflags-y += -DDEBUG -g -ggdb -Og #-Wextra #-Wa,-a,-ad,-ah,-al
[...]
```

and do

```
$ objdump -d -S ./veth.ko > veth.disas
$
```

<<
FYI:
On all these systems, `CONFIG_DEBUG_INFO=y` and `CONFIG_DEBUG_KERNEL=y` !

```
$ grep -w -E "CONFIG_DEBUG_INFO|CONFIG_DEBUG_KERNEL" *
kconfig_oneplus_tab_go:CONFIG_DEBUG_INFO=y
kconfig_oneplus_tab_go:CONFIG_DEBUG_KERNEL=y
kconfig_samsung_flip3_SM_F711B:CONFIG_DEBUG_INFO=y
kconfig_samsung_flip3_SM_F711B:CONFIG_DEBUG_KERNEL=y
kconfig_samsung_glxy_tabA_may20.config:CONFIG_DEBUG_INFO=y
kconfig_samsung_glxy_tabA_may20.config:CONFIG_DEBUG_KERNEL=y
ubuntu_x86_64_config-6.8.0-41-generic:CONFIG_DEBUG_KERNEL=y
ubuntu_x86_64_config-6.8.0-41-generic:CONFIG_DEBUG_INFO=y
```

It's good, and required, to keep some minimal debug info enabled even in production!
>>

The addition of the special linker / assembler flags
`-Wa,-a,-ad,-ah,-al`
to the CFLAGS (**`ccflags-y += <...>`** these days) have the 'make' generate a mixed assembly-source-machine code listing ! Try it.

Details: Producing more verbose information via assembler switches:
Above, we saw the `-Wa,a,-ad,-ah,-al` option switches:

```
From 'man as':
…
 If you are invoking as via the GNU C compiler, you can use the -Wa option to
pass arguments through to the assembler.  The assembler arguments must be
       separated from each other (and the -Wa) by commas.  For example:

               gcc -c -g -O -Wa,-alh,-L file.c

       This passes two options to the assembler: -alh (emit a listing to
standard output with high-level and assembly source) and -L (retain local
symbols in the
       symbol table).
…
   -a[cdghlmns]
           Turn on listings, in any of a variety of ways:
```

```
            -ac omit false conditionals
            -ad omit debugging directives
            -ag include general information, like as version and options passed
            -ah include high-level source
            -al include assembly
            -am include macro expansions
            -an omit forms processing
            -as include symbols
...
```

Q. What if you don't have a vmlinux with debug symbolic information?
A.

Well, *YMMV* !

a) the kernel source now has a script:
scripts/extract-vmlinux

Eg. on x86_64:

```
$ sudo ~/5.4/scripts/extract-vmlinux /boot/vmlinuz-5.8.0-43-generic > vmlinux
$ ls -lh vmlinux
-rw-rw-r-- 1 kaiwan kaiwan 43M Feb 16 18:49 vmlinux
$ file vmlinux
vmlinux: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
BuildID[sha1]=<...>, stripped
```

Also see:
https://stackoverflow.com/questions/12002315/extract-vmlinux-from-vmlinuz-or-bzimage/12002789#12002789

Only works on x86_64 ?? (tried on an ARM kernel image, it failed with "extract-vmlinux: Cannot find vmlinux."

b) Use the ***kdress*** tool from ELFmaster to generate it from the regular vmlinuz !
https://github.com/elfmaster/kdress

(But, kdress seems to *require* a System,map file too).

Also, there's scripts/decode_stacktrace.sh !

```
$ ~/5.4/scripts/decode_stacktrace.sh
Usage:
     /home/kaiwan/5.4/scripts/decode_stacktrace.sh [vmlinux] [base path] [modules
path]
```

Requires the vmlinux image. Could be useful…

- *Regarding the* ***machine code*** *that is listed at the end of an Oops dump:*

  How exactly can one interpret the machine code at the end of an Oops dump? Exactly [this question was posed on the superb StackOverflow website](); a good response:

  "You could use a disassembler. I found [one]() online. Copy, Edit and paste `c3 89 fa 66 ed 0f b7 c0 c3 89 fa ed c3 f0 48 0f`, choose processor family (they have plenty to choose from) and you get the result. … "

  - Modern: use the *scripts/decodecode* script !

- *[Old: From LDD3]*

  ...
  ...when looking at oops listings, always be on the lookout for the "slab poisoning" values discussed at the beginning of this chapter. Thus, for example, if you get a kernel oops where the offending address is 0xa5a5a5a5 , you are almost certainly forgetting to initialize dynamic memory somewhere.
  ...
  << Setting up CONFIG_SLUB_DEBUG helps with exactly this stuff (in kernel config under: Kernel Hacking / Debug slab memory allocations >>

- [Old] Refer to *"LINUX Debugging and Performance Tuning" by Steve Best, Pearson Education Ch 7 "System Error Messages" sections:*

  - "Oops Analysis" (page 190)
  - Generating assembly source (pages 198-200)
  - Actual examples of Oops'es that have been reported on the Linux kernel mailing list (the LKML, accessible at http://groups.google.com/group/linux.kernel?hl=en) make for interesting reading; see the section "Kernel Mailing List Oops" (pages 200-207)

- Misc: From http://kerneltrap.org/news

Lab Assignment

Check for and correct bugs with the *lab_oops_hang_panic LKM*.

Tips-

For an embedded Linux, do always run on the console (connected over the USB-serial interface); else, you may not even see the Oops printk's!

Else, use **netconsole** (YMMV, though). (Else, use **kdump/crash**).

On ARM-32, to figure the Oops bitmask (typically the [D]FSR register content), refer the relevant processor TRM.
Eg. when running it on the Raspberry Pi 4 (in 32-bit mode):
-uses the Broadcom BCM2711 SoC; which employs the Cortex-A72 ARM (quad) core processor.

Processor TRM: https://developer.arm.com/documentation/100095/0003/

```
Internal error: Oops: a0e [#1] SMP ARM
```

Data Fault Status Register (DFSR) : https://developer.arm.com/documentation/100095/0003/System-Control/AArch32-register-descriptions/Data-Fault-Status-Register?lang=en

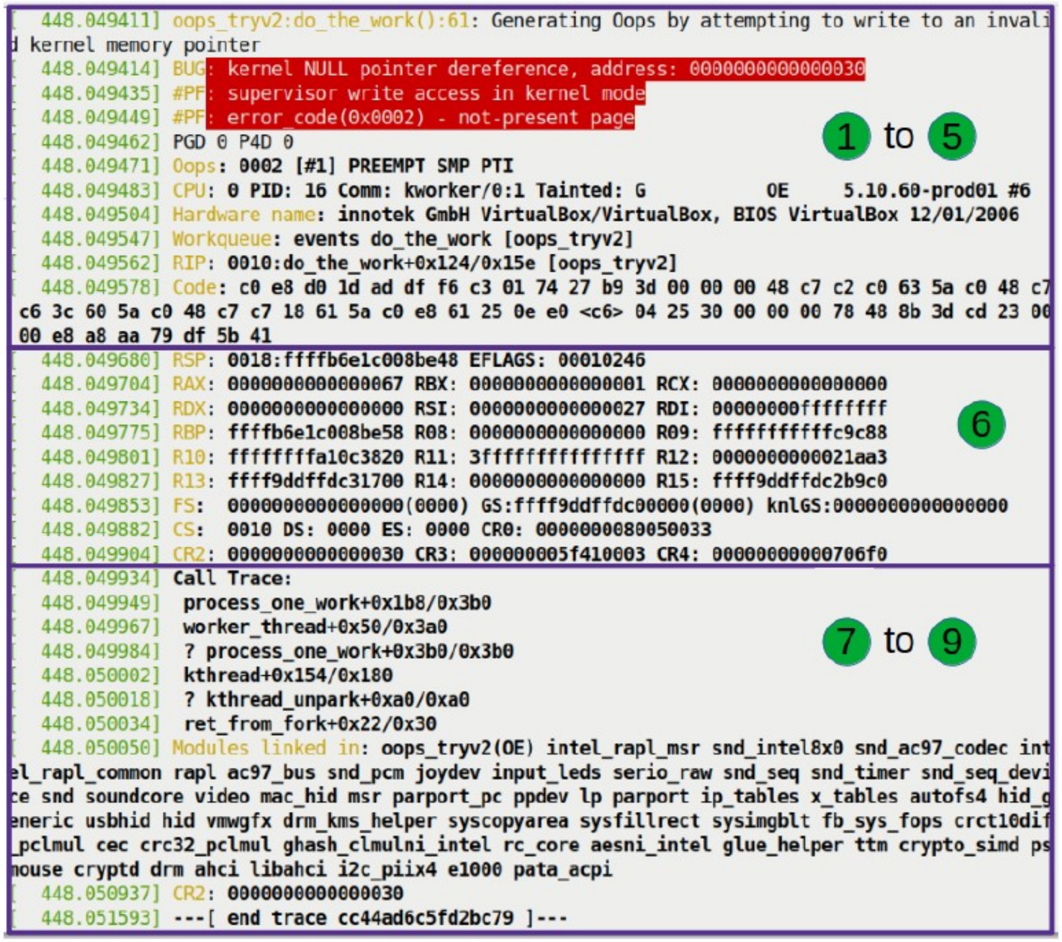
On AArch64, the Oops bitmask seems to be the value of the ESR (**`Exception Syndrome Register`**; see the 'Analyzing a kernel Oops on AArch64' document for more.

Oops 3

<<
The discussion below – *'The foggy crystal ball'* - is useful but pretty outdated!

Please see / use my *Linux Kernel Debugging (Aug 2022)*, *Packt* book *Ch 7 - Oops! Interpreting the Kernel Bug Diagnostic* section *The devil is in the details – decoding the Oops* for a modern take on this key topic.

Screenshot:




```
[  448.049411] oops_tryv2:do_the_work():61: Generating Oops by attempting to write to an invali
d kernel memory pointer
[  448.049414] BUG: kernel NULL pointer dereference, address: 0000000000000030
[  448.049435] #PF: supervisor write access in kernel mode
[  448.049449] #PF: error_code(0x0002) - not-present page
[  448.049462] PGD 0 P4D 0
[  448.049471] Oops: 0002 [#1] PREEMPT SMP PTI
[  448.049483] CPU: 0 PID: 16 Comm: kworker/0:1 Tainted: G           OE     5.10.60-prod01 #6
[  448.049504] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[  448.049547] Workqueue: events do_the_work [oops_tryv2]
[  448.049562] RIP: 0010:do_the_work+0x124/0x15e [oops_tryv2]
[  448.049578] Code: c0 e8 d0 1d ad df f6 c3 01 74 27 b9 3d 00 00 00 48 c7 c2 c0 63 5a c0 48 c7
 c6 3c 60 5a c0 48 c7 c7 18 61 5a c0 e8 61 25 0e e0 <c6> 04 25 30 00 00 00 78 48 8b 3d cd 23 00
 00 e8 a8 aa 79 df 5b 41
[  448.049680] RSP: 0018:ffffb6e1c008be48 EFLAGS: 00010246
[  448.049704] RAX: 0000000000000067 RBX: 0000000000000001 RCX: 0000000000000000
[  448.049734] RDX: 0000000000000000 RSI: 0000000000000027 RDI: 00000000ffffffff
[  448.049775] RBP: ffffb6e1c008be58 R08: 0000000000000000 R09: ffffffffffffc9c88
[  448.049801] R10: fffffffa10c3820 R11: 3fffffffffffffff R12: 0000000000021aa3
[  448.049827] R13: ffff9ddffdc31700 R14: 0000000000000000 R15: ffff9ddffdc2b9c0
[  448.049853] FS:  0000000000000000(0000) GS:ffff9ddffdc00000(0000) knlGS:0000000000000000
[  448.049882] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  448.049904] CR2: 0000000000000030 CR3: 000000005f410003 CR4: 00000000000706f0
[  448.049934] Call Trace:
[  448.049949]  process_one_work+0x1b8/0x3b0
[  448.049967]  worker_thread+0x50/0x3a0
[  448.049984]  ? process_one_work+0x3b0/0x3b0
[  448.050002]  kthread+0x154/0x180
[  448.050018]  ? kthread_unpark+0xa0/0xa0
[  448.050034]  ret_from_fork+0x22/0x30
[  448.050050] Modules linked in: oops_tryv2(OE) intel_rapl_msr snd_intel8x0 snd_ac97_codec int
el_rapl_common rapl ac97_bus snd_pcm joydev input_leds serio_raw snd_seq snd_timer snd_seq_devi
ce snd soundcore video mac_hid msr parport_pc ppdev lp parport ip_tables x_tables autofs4 hid_g
eneric usbhid hid vmwgfx drm_kms_helper syscopyarea sysfillrect sysimgblt fb_sys_fops crct10dif
_pclmul cec crc32_pclmul ghash_clmulni_intel rc_core aesni_intel glue_helper ttm crypto_simd ps
mouse cryptd drm ahci libahci i2c_piix4 e1000 pata_acpi
[  448.050937] CR2: 0000000000000030
[  448.051593] ---[ end trace cc44ad6c5fd2bc79 ]---
```

Figure 7.6 – Annotated (full) screenshot of the Oops output from our oops_tryv2 workqueue (case 3) function bug

>>

The foggy crystal ball: Understanding Oopses

by Olaf Kirch <okir@suse.de>
[URL: http://www.suse.de/~sh/Bugreporting-faq/oops-reading.txt]

<< The above URL is outdated; now, please see similar (but old, 2.4 Linux) content here:

Linux Kernel Debugging Introduction

and

https://en.opensuse.org/openSUSE:Bugreport_kernel
>>

In the introduction, we already covered how to capture an oops. In this document, we will look at the format of an oops in some detail, and demonstrate what can be learned from that information.

```
# insmod ./oops2.ko ; dmesg
Killed
[   70.173154] oops2: module verification failed: signature and/or  required key missing - tainting kernel
[   70.174364] oops2_init:46 : Hello, about to Oops!
[   70.174384] BUG: unable to handle kernel NULL pointer dereference at 0000000000000040
[   70.174389] IP: [<ffffffffa000503c>] oops2_init+0x3c/0x1000 [oops2]
[   70.174396] PGD 37a24067 PUD 37a12067 PMD 0
[   70.174402] Oops: 0002 [#1] SMP
[   70.174406] Modules linked in: oops2(OE+) bnep bluetooth fuse nf_conntrack_netbios_ns nf_conntrack_broadcast ip6t_rpfilter ip6t_REJECT xt
_conntrack cfg80211 rfkill ebtable_nat ebtable_broute bridge stp llc ebtable_filter ebtables ip6table_nat nf_conntrack_ipv6 nf_defrag_ipv6 n
f_nat_ipv6 ip6table_mangle ip6table_security ip6table_raw ip6table_filter ip6_tables iptable_nat nf_conntrack_ipv4 nf_defrag_ipv4 nf_nat_ipv
4 nf_nat nf_conntrack iptable_mangle iptable_security iptable_raw snd_ens1371 gameport snd_rawmidi snd_ac97_codec ac97_bus coretemp crct10di
f_pclmul crc32_pclmul crc32c_intel ppdev snd_seq ghash_clmulni_intel snd_seq_device vmw_balloon snd_pcm serio_raw snd_timer snd soundcore vm
w_vmci shpchp i2c_piix4 parport_pc parport vmwgfx drm_kms_helper mptspi e1000 ttm drm scsi_transport_spi mptscsih
[   70.174466]  mptbase ata_generic pata_acpi
[   70.174473] CPU: 1 PID: 2286 Comm: insmod Tainted: G           OE  3.17.4-301.fc21.x86_64 #1
[   70.174476] Hardware name: VMware, Inc. VMware Virtual Platform/440BX Desktop Reference Platform, BIOS 6.00 07/31/2013
[   70.174479] task: ffff88003adf09d0 ti: ffff88003a098000 task.ti: ffff88003a098000
[   70.174482] RIP: 0010:[<ffffffffa000503c>]  [<ffffffffa000503c>] oops2_init+0x3c/0x1000 [oops2]
[   70.174489] RSP: 0018:ffff88003a09bd10  EFLAGS: 00010282
[   70.174491] RAX: 0000000000000000 RBX: ffffffff81c18040 RCX: 0000000000000025
[   70.174494] RDX: 0000000000000000 RSI: ffff88003f62e6f8 RDI: ffff88003f62e6f8
[   70.174496] RBP: ffff88003a09bd10 R08: 0000000000000092 R09: 0000000000000630
[   70.174499] R10: ffffffff81002138 R11: 0000000000000630 R12: ffff880036578880
[   70.174501] R13: 0000000000000000 R14: ffffffffa0005000 R15: 0000000000000001
[   70.174505] FS:  00007f1e7a296700(0000) GS:ffff88003f620000(0000) knlGS:0000000000000000
[   70.174507] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[   70.174510] CR2: 0000000000000040 CR3: 000000003a062000 CR4: 00000000000407e0
[   70.174637] Stack:
[   70.174641]  ffff88003a09bd88 ffffffff81002148 ffff88003a09bd48 ffffffff811ed35e
```

1. Interpreting Oops messages
==============================

The following will assume that the oops includes symbolic names for all addresses, either because the kernel already supports the kallsyms feature, or because the oops was massaged by syslogd or ksymooops already.

Here is a typical oops from a SLES9 beta kernel:

<< For readability, I've highlighted some places like so (not part of the original article). >>

```
Unable to handle kernel NULL pointer dereference at virtual
address 00000094
 printing eip:
c02b47f6
*pde = 00000000
Oops: 0000 [#1]
CPU:    0
EIP:    0060:[<c02b47f6>]    Tainted: G  U
EFLAGS: 00010202   (2.6.5-7.97-default)
EIP is at udp_rcv+0xf6/0x5e0
```

<< Register dump >>

```
eax: cfb53b80   ebx: df206844   ecx: 0e217d13   edx: 00000020
esi: 00002088   edi: 00882088   ebp: 00000000   esp: c0399e88
ds: 007b   es: 007b   ss: 0068
```

<< Process that got caught in the Oops>>

```
Process swapper (pid: 0, threadinfo=c0398000 task=c033b100)
```

<< Stack dump >>

```
Stack: e1047860 00000000 e10478a0 c0284f0e 00000000 4480b609
       4180b609 df206844 cfb53b80 cfb53b80 c035e0a0 00000000
       00000000 c0294982 00000000 00000001 00000002 c0285bca
       00000000 cfb53b80 c03ffe88 cfb53b80 c856f030 d263f080

Call Trace:
 [<c0284f0e>] nf_iterate+0x5e/0xb0
 [<c0294982>] ip_local_deliver_finish+0x52/0x1c0
 [<c0285bca>] nf_hook_slow+0xea/0xf0
 [<c0294c5a>] ip_local_deliver+0x16a/0x220
 [<c0294930>] ip_local_deliver_finish+0x0/0x1c0
 [<c0294873>] ip_rcv+0x3c3/0x480
 [<c027cf40>] netif_receive_skb+0x210/0x220
 [<c02799f2>] alloc_skb+0x32/0xd0
 [<e104ce33>] e100_poll+0x2d3/0x650 [e100]
 [<c027bda2>] net_rx_action+0xa2/0xf0
 [<c01231e3>] __do_softirq+0x43/0x90
 [<c0123256>] do_softirq+0x26/0x30
 [<c010a8f5>] do_IRQ+0x125/0x1a0
 [<c0108d48>] common_interrupt+0x18/0x20
 [<c0106290>] default_idle+0x0/0x30
 [<c01062b3>] default_idle+0x23/0x30
 [<c0106dac>] cpu_idle+0x1c/0x40
 [<c039a629>] start_kernel+0x299/0x300
Code: f7 85 94 00 00 00 00 00 00 30 0f 84 62 01 00 00 8b 4c 24
1c
```

How do we interpret such an oops? Let's go through the message in some detail before explaining all the fields in their entirety.

The very first line gives you a clue why the exception happened, in this case the kernel tried to access an invalid address (00000094).

As it's a very small integer, this is less likely to be a corrupted pointer but rather an offset (0x94) added to address 0. This happens if you have a NULL pointer to a struct, say "struct foo *mypointer", any your code tries to access "mypointer→somevar" with struct member somevar at offset 0x94.

Next comes the instruction pointer (EIP on i386) as hex address; and a few lines down the same address displayed relative to the closest symbol known to the kernel.

<< From Documentation/oops-tracing.txt :

[Outdated: see *admin-guide: merge oops-tracing with bug-hunting* (08Nov2016)]

--snip--

- -

Tainted kernels

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

https://kernel.org/doc/html/latest/admin-guide/tainted-kernels.html

https://www.kernel.org/doc/html/latest/admin-guide/tainted-kernels.html#table-for-decoding-tainted-state :

Table for decoding tainted state

Bit	Log	Number	Reason that got the kernel tainted
0	G/P	1	proprietary module was loaded
1	_/F	2	module was force loaded
2	_/S	4	kernel running on an out of specification system
3	_/R	8	module was force unloaded
4	_/M	16	processor reported a Machine Check Exception (MCE)
5	_/B	32	bad page referenced or some unexpected page flags
6	_/U	64	taint requested by userspace application
7	_/D	128	kernel died recently, i.e. there was an OOPS or BUG
8	_/A	256	ACPI table overridden by user
9	_/W	512	kernel issued warning
10	_/C	1024	staging driver was loaded
11	_/I	2048	workaround for bug in platform firmware applied
12	_/O	4096	externally-built (“out-of-tree”) module was loaded
13	_/E	8192	unsigned module was loaded
14	_/L	16384	soft lockup occurred
15	_/K	32768	kernel has been live patched
16	_/X	65536	auxiliary taint, defined for and used by distros
17	_/T	131072	kernel was built with the struct randomization plugin

Note: The character _ is representing a blank in this table to make reading easier.

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 << *unless root overwrites /proc/sys/kernel/tainted !* >>

TIP
It’s just simpler to run a script which will interpret the tainted flags (available on recent kernels); as an

example, an Oops I got had the tainted flags value as 4097 – by looking up **/proc/sys/kernel/tainted**; to interpret this:

```
$ cd <kernel-src-tree-root>
$ cat /proc/sys/kernel/tainted
4097
$ tools/debugging/kernel-chktaint 4097
<...>/tools/debugging/kernel-chktaint: 22: [: 4097x: unexpected operator
<...>/tools/debugging/kernel-chktaint: 22: [: 4097x: unexpected operator
Kernel is "tainted" for the following reasons:
 * proprietary module was loaded (#0)
 * externally-built ('out-of-tree') module was loaded  (#12)
For a more detailed explanation of the various taint flags see
 Documentation/admin-guide/tainted-kernels.rst in the the Linux kernel sources
 or https://kernel.org/doc/html/latest/admin-guide/tainted-kernels.html
Raw taint value as int/string: 4097/'P           O     '
$
```

>>

<<

Oops: 0000 [#1] ← this looks to be an old way of displaying Oops bits; the current manner to interpret bits is shown by the table below:

Table 4-1: Meaning of Page Fault Error Codes on IA-32

Bit	Set (1)	Not set (0)
0	No page present in RAM	Protection fault (insufficient access permission)
1	Read access	Write access
2	Privileged kernel mode	User mode

[#1] => # of times this Oops occurred. Multiple Oops can be triggered as a cascading effect of the first one.

What about the Oops bitmask equivalent on ARM-32?

F.e., the above sample buggy kernel module (oops_simple.ko), when run on an Arm-32 (a Raspberry Pi 0W), showed this:

```
oops2_init:42 : Hello, about to Oops!
<--- cut here ---
Unable to handle kernel NULL pointer dereference at virtual address 00000024
pgd = afac1bc4
[00000024] *pgd=02799831, *pte=00000000, *ppte=00000000
Internal error: Oops: 817 [#1] ARM
Modules linked in: oops_simple(O+) aes_arm aes_generic cmac bnep hci_uart btbcm
bluetooth ecdh_generic ecc libaes 8021q garp stp
```

. . .

Please see my answer to this SO Q&A:

Kernel Oops page fault error codes for ARM

(Regards the ARM MMU FSR (Fault Status Register)

The **FAR (Fault Address Register)** holds the faulting virtual address, equivalent to CR2 on Intel MMUs.

Note: on AArch64 (ARM64), the equivalent register is the **`FAR_ELn`** that holds the faulting virtual address at Exception Level (EL) n ; n = 1, 2, 3. (FYI, EL0 is userspace, EL1 typically the kernel-space, EL2 the hypervisor, if any, and EL3 the Secure Monitor mode).

<<

From the ARMv8A TRM:

“D12.2.39

FAR_EL1, Fault Address Register (EL1)

The FAR_EL1 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction or Data Abort, PC alignment fault and Watchpoint exceptions that are taken to EL1. ...”

>>

These symbols deserve a little explanation. In the example above, the exception happened when executing the instruction at "udp_rcv+0xf6/0x5e0". This indicates an offset of 0xf6 counting from the start of function *udp_rcv*. The second hex number indicates what the kernel thinks is the length of that function. In fact, it's the distance to the next global symbol; there can actually be static functions that follow udp_rcv which do not show up in the symbol table. However, these two numbers can give you a rough indication of where the crash happened. In this case, the problematic statement should be no more than a dozen lines from the start of udp_rcv().

A few lines down there is the pid and name of the process that died. In this case, it's the swapper (pid 0), which is not a regular process. This indicates that we crashed inside an interrupt or a bottom half handler.

Another important piece of information is the dump of all registers. Here the kernel basically displays the content of all registers prior to the execution of the faulting instruction. This can help locate the problem when looking at the disassembly dump. As we're not looking at the disassembly (yet), we'll ignore these for now.

Following the process information, the oops includes a hexdump of the top of the stack. Some of the words on the stack are return addresses, so the kernel also tries to translate map these into symbols. The algorithm used is rather simple-minded; the Suse kernel doesn't use frame pointers, and it doesn't have any other debugging information (such as dwarf) available either. Therefore, it uses a simple heuristic to check whether a given value could be the address of a kernel function or not, and if it thinks so, it will map it to the closes symbol.

This will usually give you a close idea of the call chain, but you should not treat this as 100% reliable. For instance, the stack may contain left-overs from previous function calls that show up as weird blips in the backtrace. Similar, if function pointers are passed around as arguments on the stack, these will show up in the backtrace as well.

In the example above, the function names that stick out are common_interrupt (confirming the fact that we're handling an interrupt), ip_rcv (confirming that we're delivering an IP packet, and nf_hook_slow. The latter is in the packet filtering code, and is never called unless a packet filter has been installed. So we know that the kernel was configured to do packet filtering.

The last line of the oops shows the instruction(s) at the faulting location. Even without much knowledge of i386 assembler, the "94 00 00 00" sticks out, which matches exactly the offset 0x94 we hypothesized above. And indeed, feeding the oops above to ksymoops shows that the instruction is "testl $0x30000000,0x94(%ebp)".

2. Looking at the disassembly
==============================

Now that we know the oops happened in udp_rcv, it may be time to look at the disassembly (we could skip this step and look at the source code directly, but for the sake of a didactic example, please bear with me :).

Probably the easiest way is to use gdb and the "disas" command. If the offending function is in the kernel itself (as opposed to a loadable module), you will need a copy of the uncompressed kernel image. The Suse kernel RPMs all contain a gzipped copy as */boot/vmlinux.gz* (SLES8) or */boot/vmlinux-<kernelversions>.gz*. Simply uncompress this image and run gdb on it.

If the offending function is in a module, you can also use gdb on the module object directly. Alternatively, you can use the objdump utility to obtain a full text dump of the entire module, which is a little more than you actually need, but on the other hand it allows you search this dump more efficiently using grep etc.

Following through with our example, here's the section of code in udp_rcv where the crash happens:

<<
To generate the disassembly dump, first load the module file (.[k]o) using gdb (or in gdb use the "symbol-file <filename>" directive. Then, issue the command "disas 0" at the (gdb) prompt.

>>

```
0xc02b47df <udp_rcv+226>:       push   %eax
0xc02b47e0 <udp_rcv+227>:       push   %edi
0xc02b47e1 <udp_rcv+228>:       push   %ebx
0xc02b47e2 <udp_rcv+229>:       call   0xc02b467d<udp_checksum_init>
0xc02b47e7 <udp_rcv+234>:       add    $0x14,%esp
0xc02b47ea <udp_rcv+237>:       test   %eax,%eax
0xc02b47ec <udp_rcv+239>:       js     0xc02b4746 <udp_rcv+73>
```

```
0xc02b47f2 <udp_rcv+245>:        mov    0x8(%esp,1),%ebp
0xc02b47f6 <udp_rcv+249>:        testl  $0x30000000,0x94(%ebp)
                                    HERE << 0xc02b47f6 is the IP >>

0xc02b47fd <udp_rcv+256>:        je     0xc02601e8 <udp_rcv+296>
0xc02b47ff <udp_rcv+258>:        mov    (%esp,1),%eax
0xc02b4802 <udp_rcv+261>:        push   %eax
```

What we can see here is that the offending instruction immediately follows a function call to udp_checksum_init(). This is easily located in udp_rcv:

```
struct rtable *rt = (struct rtable*) skb->dst;
...
   if (udp_checksum_init(skb, uh, ulen, saddr, daddr) < 0)
            goto csum_error;
   if (rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))
           return udp_v4_mcast_deliver(skb, uh, saddr, daddr);
```

The crash occurs in the second if() statement because the pointer "rt" was NULL, which in turn was initialized from skb->dst at the top of the function. This should never happen...

3. Conclusion
=============

So now we know where it crashed, and what the immediate cause for this crash was. From here on, debugging is mostly about understanding the code, and figuring out what triggers this unexpected condition.

In the case discussed above, the description of what happens is actually not really straightforward, so if you're interested in the bug itself (and the fix), please refer to SUSE bug #42902.

4. Detailed anatomy of an Oops
==============================

5. Special oops features
========================

[Describe version specific oops features, such as cvs timestamps here]

N. Authors
==========

Olaf Kirch <okir@suse.de>

Investigating kernel Oops on the Aarch32 (ARM-32)

Normal faults occur

do_page_fault → __do_page_fault → handle_mm_fault → __handle_mm_fault […] → *[something's wrong!]* arm_notify_die → die → __die

4.16.0
Functions calling this function: arm_notify_die

```
  File          Function         Line
0 swp_emulate.c set_segfault  127 arm_notify_die("Illegal memory access", regs,
&info, 0, 0);
1 traps.c   do_undefinstr 491 arm_notify_die("Oops - undefined instruction", regs,
&info, 0, 6);
2 traps.c   bad_syscall  559 arm_notify_die("Oops - bad syscall", regs, &info, n,
0);
3 traps.c   arm_syscall  617 arm_notify_die("branch through zero", regs, &info, 0,
0);
4 traps.c   arm_syscall  691 arm_notify_die("Oops - bad syscall(2)", regs, &info,
no, 0);
5 traps.c   baddataabort     757 arm_notify_die("unknown data abort code", regs,
&info, instr, 0);
6 fault.c   do_DataAbort     564 arm_notify_die("", regs, &info, fsr, 0);
7 fault.c   do_PrefetchAbort 596 arm_notify_die("", regs, &info, ifsr, 0);
```

<<
Prefetch Abort: entered into when the prefetched instruction is invalid
Data Abort: entered into when invalid memory is accessed (various reasons: permissions, invalid reference, … ; *details here* (ARM dev guide)
>>

Src: arch/arm/mm/fault.c (and arch/arm/mm/traps.c) (ver 4.16.0)

"ARM Oops Kernel Messages" Table

Reason / Comment	*Function*	*Leading printk in the Oops message*	***[D]FSR** passed?*	*Oops invoked*

Oops. The kernel tried to access some page that wasn't present.	`static void __do_kernel_fault(struct mm_struct *mm, unsigned long addr, unsigned int fsr, struct pt_regs *regs)`	`pr_alert("Unable to handle kernel %s at virtual address %08lx\n", (addr < PAGE_SIZE) ? "NULL pointer dereference" : "paging request", addr);`	Y	Y die(...)

Something tried to access memory that isn't in our memory map..
User mode accesses just cause a SIGSEGV
	`__do_user_fault`	`#ifdef CONFIG_DEBUG_USER`
		`[...]`
		`printk(KERN_DEBUG "%s: unhandled page fault (%d) at 0x%08lx, code 0x%03x\n", tsk->comm, sig, addr, fsr);`
			Y	N

Dispatch a data abort to the relevant handler.	`do_DataAbort`	`pr_alert("Unhandled fault: %s (0x%03x) at 0x%08lx\n", inf->name, fsr, addr);`	Y	Y arm_notify_die(...)
A data abort trap was taken, but we did not handle the instruction. * Try to abort the user program, or panic if it was the kernel.	`arch/arm/kernel/traps.c:baddataabort`	`#ifdef CONFIG_DEBUG_USER` `if (user_debug & UDBG_BADABORT) { pr_err("[%d] %s: bad data abort: code %d instr 0x%08lx\n", task_pid_nr(current), current->comm, code, instr); ...`	N	N
Prefetch Abort	`do_PrefetchAbort`	`pr_alert("Unhandled prefetch abort: %s (0x%03x) at 0x%08lx\n", inf->name, ifsr, addr);`	Y (ifsr)	Y arm_notify_die(...)
* Abort handler to be used only during first unmasking of asynchronous aborts * on the boot CPU. This makes sure that the machine will not die if the * firmware/bootloader left an imprecise abort pending for us to trip over.	`early_abort_handler`	`pr_warn("Hit pending asynchronous external abort (FSR=0x%08x) during " "first unmask, this is most likely caused by a "` `"firmware/bootloader bug.\n", fsr);`	Y	N

Data abort but no message via the caller – hence we only see the "Internal error: Oops: …" in the kernel log	`arm_notify_die ("", ...)`	`[5.4]:` `arch/arm/mm/fault.c:541:` `arm_notify_die("", regs, inf->sig, inf->code, (void __user *)addr, fsr, 0);` `arch/arm/mm/fault.c-543-}` `arch/arm/mm/fault.c-544-`	Y (fsr)	Y
The *die* code	`traps.c:void die(const char *str, struct pt_regs *regs, int err)`	`if (bug_type != BUG_TRAP_TYPE_NONE) str = "Oops - BUG"; if (__die(str, err, regs)) …`	Y (err)	Y
The *__die* code	`arch/arm/kernel/traps.c:__die`	`pr_emerg("Internal error: %s: %x [#%d]" S_PREEMPT S_SMP S_ISA "\n", str, err, ++die_counter);`	Y (as 'err')	-

Want more details? Do look up *Appendix A :: ARM [D]FSR Register Details*.

<<
An aside: the Raspberry Pi base models use the BCM2835 SoC which is based on the ARM1176JZF-S ARM core.

[1] TRM:
https://developer.arm.com/documentation/ddi0301/h?lang=en

[2] Fault Status Register encoding on the ARM1176JZF-S processor within the TRM:
https://developer.arm.com/documentation/ddi0301/h/memory-management-unit/fault-status-and-address?lang=en

[3] Data Fault Status Register encodings
https://developer.arm.com/documentation/ddi0301/h/system-control-coprocessor/system-control-processor-registers/c5--data-fault-status-register?lang=en

I had an Oops on this SoC with this:

```
Unable to handle kernel NULL pointer dereference at virtual address 00000288
 kernel: pgd = e707950c
 kernel: [00000288] *pgd=06462831, *pte=00000000, *ppte=00000000
 kernel: Internal error: Oops: 17 [#1] ARM
```
…

The number 17 is in hexadecimal. (How do I know?
- because the code that generated the above line is:

```
pr_emerg("Internal error: %s: %x [#%d]" S_PREEMPT S_SMP S_ISA "\n",  str, err,
++die_counter);
```

)

The ‘err’ variable holds the DFSR register value!

Here’s the interpretation on this processor (from [3]):

Differs, depends on value of bit 10.
With bit [10] == 0 or 1: (here’s its 0)

bits [3:0] “Indicates type of fault generated”

[3:0] with bit[10] = 0	Status	Indicates type of fault generated. See *Fault status and address* for t b0000 = no function, reset value b0001 = Alignment fault b0010 = Instruction debug event fault b0011 = Access Bit fault on Section b0100 = Instruction cache maintenance operation fault b0101 = Translation Section fault b0110 = Access Bit fault on Page b0111 = Translation Page fault b1000 = Precise external abort b1001 = Domain Section fault b1010 = no function b1011 = Domain Page fault b1100 = External abort on translation, first level b1101 = Permission Section fault b1110 = External abort on translation, second level b1111 = Permission Page fault.

For us, the bits [3:0] is 0x7 (from 0x17 the LSB 4 bits are the value 0x7) which of course is binary ‘b’0111. Look it up, it’s this row:

b0111 = Translation Page fault

BTW, DFSR value is 0x17; we just saw how to interpret bits [3:0]; what about bit 4? it's the 'domain':

[7:4]	Domain	Indicates the domain from the 16 domains, D15-D0, is accessed when a data fault occurs. Takes values 0-15. The reset value is 0.

here it's the value 1, thus D1.
>>

[Source: "Linux Device Drivers", 3rd Ed. by J Corbet, A Rubini and GK Hartman, O'Reilly.]

System Hangs

Although most bugs in kernel code end up as oops messages, sometimes they can completely hang the system. If the system hangs, no message is printed. For example, if the code enters an endless loop, the kernel stops scheduling,* and the system doesn't respond to any action, including the magic Ctrl-Alt-Del combination. You have two choices for dealing with system hangs—either prevent them beforehand or be able to debug them after the fact.

* Actually, multiprocessor systems still schedule on the other processors, and even a uniprocessor machine might reschedule if kernel preemption is enabled. For the most common case (uniprocessor with preemption disabled), however, the system stops scheduling altogether.

You can prevent an endless loop by inserting schedule invocations at strategic points. The schedule call (as you might guess) invokes the scheduler and, therefore, allows other processes to steal CPU time from the current process. If a process is looping in kernel space due to a bug in your driver, the schedule calls enable you to kill the process after tracing what is happening.

You should be aware, of course, that any call to schedule may create an additional source of reentrant calls to your driver, since it allows other processes to run. This reentrancy should not normally be a problem, assuming that you have used suitable locking in your driver. Be sure, however, not to call schedule any time that your driver is holding a spinlock.

If your driver really hangs the system, and you don't know where to insert schedule calls, the best way to go may be to add some print messages and write them to the console (by changing the console_loglevel value if need be).

Sometimes the system may appear to be hung, but it isn't. This can happen, for example, if the keyboard remains locked in some strange way. These false hangs can be detected by looking at the output of a program you keep running for just this purpose. A clock or system load meter on your display is a good status monitor; as long as it continues to update, the scheduler is working.

Magic SysRq

An indispensable tool for many lockups is the "magic SysRq key," which is available on most architectures. Magic SysRq is invoked with the combination of the Alt and SysRq keys on the PC keyboard, or with other special keys on other platforms (see Documentation/admin-guide/sysrq.rst* for details), and is available on the serial console as well. A third key, pressed along with these two, performs one of a number of useful actions:

<<
* .rst?
.rst files are ReStructuredText format. They look like text files, but can be rendered into HTML with the Python docutils package!
>>

```
How do I enable the magic SysRq key?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

You need to say "yes" to 'Magic SysRq key (CONFIG_MAGIC_SYSRQ)' when
configuring the kernel. When running a kernel with SysRq compiled in,
/proc/sys/kernel/sysrq controls the functions allowed to be invoked via
the SysRq key. The default value in this file is set by the
CONFIG_MAGIC_SYSRQ_DEFAULT_ENABLE config symbol, which itself defaults
to 1. Here is the list of possible values in /proc/sys/kernel/sysrq:

   -  0 - disable sysrq completely
   -  1 - enable all functions of sysrq
   - >1 - bitmask of allowed sysrq functions (see below for detailed function
     description)::

          2 =   0x2 - enable control of console logging level
          4 =   0x4 - enable control of keyboard (SAK, unraw)
          8 =   0x8 - enable debugging dumps of processes etc.
         16 =  0x10 - enable sync command
         32 =  0x20 - enable remount read-only
         64 =  0x40 - enable signalling of processes (term, kill, oom-kill)
        128 =  0x80 - allow reboot/poweroff
        256 = 0x100 - allow nicing of all RT tasks

You can set the value in the file by the following command::

    echo "number" >/proc/sys/kernel/sysrq

<<
To enable all, as root, do:

echo 1 > /proc/sys/kernel/sysrq
>>

[...]


What are the 'command' keys?
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

=========== ================================================================
Command     Function
=========== ================================================================
``b``       Will immediately reboot the system without syncing or unmounting
            your disks.

``c``       Will perform a system crash by a NULL pointer dereference.
            A crashdump will be taken if configured.

``d``       Shows all locks that are held.

``e``       Send a SIGTERM to all processes, except for init.

``f``       Will call the oom killer to kill a memory hog process, but do not
            panic if nothing can be killed.

``g``       Used by kgdb (kernel debugger)

``h``       Will display help (actually any other key than those listed
            here will display help. but ``h`` is easy to remember :-)

``i``       Send a SIGKILL to all processes, except for init.

``j``       Forcibly "Just thaw it" - filesystems frozen by the FIFREEZE ioctl.

``k``       Secure Access Key (SAK) Kills all programs on the current virtual
            console. NOTE: See important comments below in SAK section.

``l``       Shows a stack backtrace for all active CPUs.

``m``       Will dump current memory info to your console.

``n``       Used to make RT tasks nice-able

``o``       Will shut your system off (if configured and supported).

``p``       Will dump the current registers and flags to your console.

``q``       Will dump per CPU lists of all armed hrtimers (but NOT regular
            timer_list timers) and detailed information about all
            clockevent devices.

``r``       Turns off keyboard raw mode and sets it to XLATE.

``s``       Will attempt to sync all mounted filesystems.

``t``       Will dump a list of current tasks and their information to your
            console.

``u``       Will attempt to remount all mounted filesystems read-only.
```

```
``v``       Forcefully restores framebuffer console
``v``       Causes ETM buffer dump [ARM-specific]

``w``       Dumps tasks that are in uninterruptable (blocked) state.

``x``       Used by xmon interface on ppc/powerpc platforms.
            Show global PMU Registers on sparc64.
            Dump all TLB entries on MIPS.

``y``       Show global CPU Registers [SPARC-64 specific]

``z``       Dump the ftrace buffer

``0``-``9`` Sets the console log level, controlling which kernel messages
            will be printed to your console. (``0``, for example would make
            it so that only emergency messages like PANICs or OOPSes would
            make it to your console.)
=========== ===================================================================

Okay, so what can I use them for?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Well, unraw(r) is very handy when your X server or a svgalib program crashes.

sak(k) (Secure Access Key) is useful when you want to be sure there is no
trojan program running at console which could grab your password
when you would try to login. It will kill all programs on given console,
thus letting you make sure that the login prompt you see is actually
the one from init, not some trojan program.

.. important::

   In its true form it is not a true SAK like the one in a
   c2 compliant system, and it should not be mistaken as
   such.

It seems others find it useful as (System Attention Key) which is
useful when you want to exit a program that will not let you switch consoles.
(For example, X or a svgalib program.)

``reboot(b)`` is good when you're unable to shut down. But you should also
``sync(s)`` and ``umount(u)`` first.

``crash(c)`` can be used to manually trigger a crashdump when the system is hung.
Note that this just triggers a crash if there is no dump mechanism available.

``sync(s)`` is great when your system is locked up, it allows you to sync your
disks and will certainly lessen the chance of data loss and fscking. Note
that the sync hasn't taken place until you see the "OK" and "Done" appear
on the screen. (If the kernel is really in strife, you may not ever get the
OK or Done message...)

``umount(u)`` is basically useful in the same ways as ``sync(s)``. I generally
```

```
``sync(s)``, ``umount(u)``, then ``reboot(b)`` when my system locks. It's saved
me many a fsck. Again, the unmount (remount read-only) hasn't taken place until
you see the "OK" and "Done" message appear on the screen.

The loglevels ``0``-``9`` are useful when your console is being flooded with
kernel messages you do not want to see. Selecting ``0`` will prevent all but
the most urgent kernel messages from reaching your console. (They will
still be logged if syslogd/klogd are alive, though.)

``term(e)`` and ``kill(i)`` are useful if you have some sort of runaway process
you are unable to kill any other way, especially if it's spawning other
processes.

"just thaw ``it(j)``" is useful if your system becomes unresponsive due to a
frozen (probably root) filesystem via the FIFREEZE ioctl.
```

...

Note that magic SysRq must be explicitly enabled in the kernel configuration and that most distributions do not enable it, for obvious security reasons. For a system used to develop drivers, however, enabling magic SysRq is worth the trouble of building a new kernel in itself. Magic SysRq may be disabled at runtime with a command such as the following:

```
echo 0 > /proc/sys/kernel/sysrq
```

Tip:

1. Unless the printk loglevel is set low enough, the Magic SysRq messages don't show up on the console (and you'll have to look it up with dmesg). So do:

```
# echo -n “8 4 1 7” > /proc/sys/kernel/printk
```

first.
(Use sysctl to set it up permanently.)

You should consider disabling it if unprivileged users can reach your system keyboard, to prevent accidental or willing damages. Some previous kernel versions had sysrq disabled by default, so you needed to enable it at runtime by writing 1 to that same /proc/sys file.

The sysrq operations are exceedingly useful, so they have been made available to system administrators who can’t reach the console. The file /proc/sysrq-trigger is a writeonly entry point, where you can trigger a specific sysrq action by writing the associated command character; you can then collect any output data from the kernel logs. This entry point to sysrq is always working, even if sysrq is disabled on the console.

If you are experiencing a “live hang,” in which your driver is stuck in a loop but the system as a whole is still functioning, there are a couple of techniques worth knowing. Often, the SysRq p function points the finger directly at the guilty routine. Failing that, you can also use the kernel profiling function.

Build a kernel with profiling enabled, and boot it with profile=2 on the command line. Reset the profile counters with the readprofile utility, then send your driver into its loop. After a little while, use readprofile again to see where the kernel is spending its time. Another more advanced alternative is oprofile, that you may consider as well. The file *Documentation/basic_profiling.txt* tells you everything you need to know to get started with the profilers.

One precaution worth using when chasing system hangs is to mount all your disks as read-only (or unmount them). If the disks are read-only or unmounted, there's no risk of damaging the filesystem or leaving it in an inconsistent state. Another possibility is using a computer that mounts all of its filesystems via NFS, the network file system. The "NFS-Root" capability must be enabled in the kernel, and special parameters must be passed at boot time. In this case, you'll avoid filesystem corruption without even resorting to SysRq, because filesystem coherence is managed by the NFS server, which is not brought down by your device driver.

Implementation

File : [kernel ver 3.10.24] : drivers/tty/sysrq.c

```
...
static void sysrq_handle_crash(int key)
{
    char *killer = NULL;

    panic_on_oops = 1;  /* force panic */
    wmb();
    *killer = 1;
}
static struct sysrq_key_op sysrq_crash_op = {
    .handler    = sysrq_handle_crash,
    .help_msg   = "crash(c)",
    .action_msg = "Trigger a crash",
    .enable_mask    = SYSRQ_ENABLE_DUMP,
};

static void sysrq_handle_reboot(int key)
{
    lockdep_off();
    local_irq_enable();
    emergency_restart();
}
static struct sysrq_key_op sysrq_reboot_op = {
    .handler    = sysrq_handle_reboot,
    .help_msg   = "reboot(b)",
    .action_msg = "Resetting",
    .enable_mask    = SYSRQ_ENABLE_BOOT,
};
...
```

WARNings and BUGs

The Linux kernel provides macros for kernel devs to emit warnings – WARN() and WARN_ON() - and, in really bad cases, a BUG() macro. (The worst case of course, is an irrecoverable situation, where panic() is used – covered next).

NOTE: BUG() calls panic() !

```
include/asm-generic/bug.h
...
#define WARN(condition, format...) ({                    \
    int __ret_warn_on = !!(condition);                  \
    unlikely(__ret_warn_on);                            \
})
```

(In fact, several variations exist:
WARN_ON , *WARN_ONCE* , *WARN_ON_ONCE* , *WARN_ON_SMP* , *WARN_TAINT* , *WARN_TAINT_ONCE* , etc!)

BUG()

```
/*
 * Don't use BUG() or BUG_ON() unless there's really no way out; one
 * example might be detecting data structure corruption in the middle
 * of an operation that can't be backed out of.  If the (sub)system
 * can somehow continue operating, perhaps with reduced functionality,
 * it's probably not BUG-worthy.
 *
 * If you're tempted to BUG(), think again:  is completely giving up
 * really the *only* solution?  There are usually better options, where
 * users don't need to reboot ASAP and can mostly shut down cleanly.
 */
#ifndef HAVE_ARCH_BUG
#define BUG() do { \
    printk("BUG: failure at %s:%d/%s()!\n", __FILE__, __LINE__, __func__); \
    barrier_before_unreachable(); \
    panic("BUG!"); \
} while (0)
#endif

#ifndef HAVE_ARCH_BUG_ON
#define BUG_ON(condition) do { if (unlikely(condition)) BUG(); } while(0)
```

```
#endif
```

Use *cscope* to lookup instances of these macros being used within the kernel codebase.

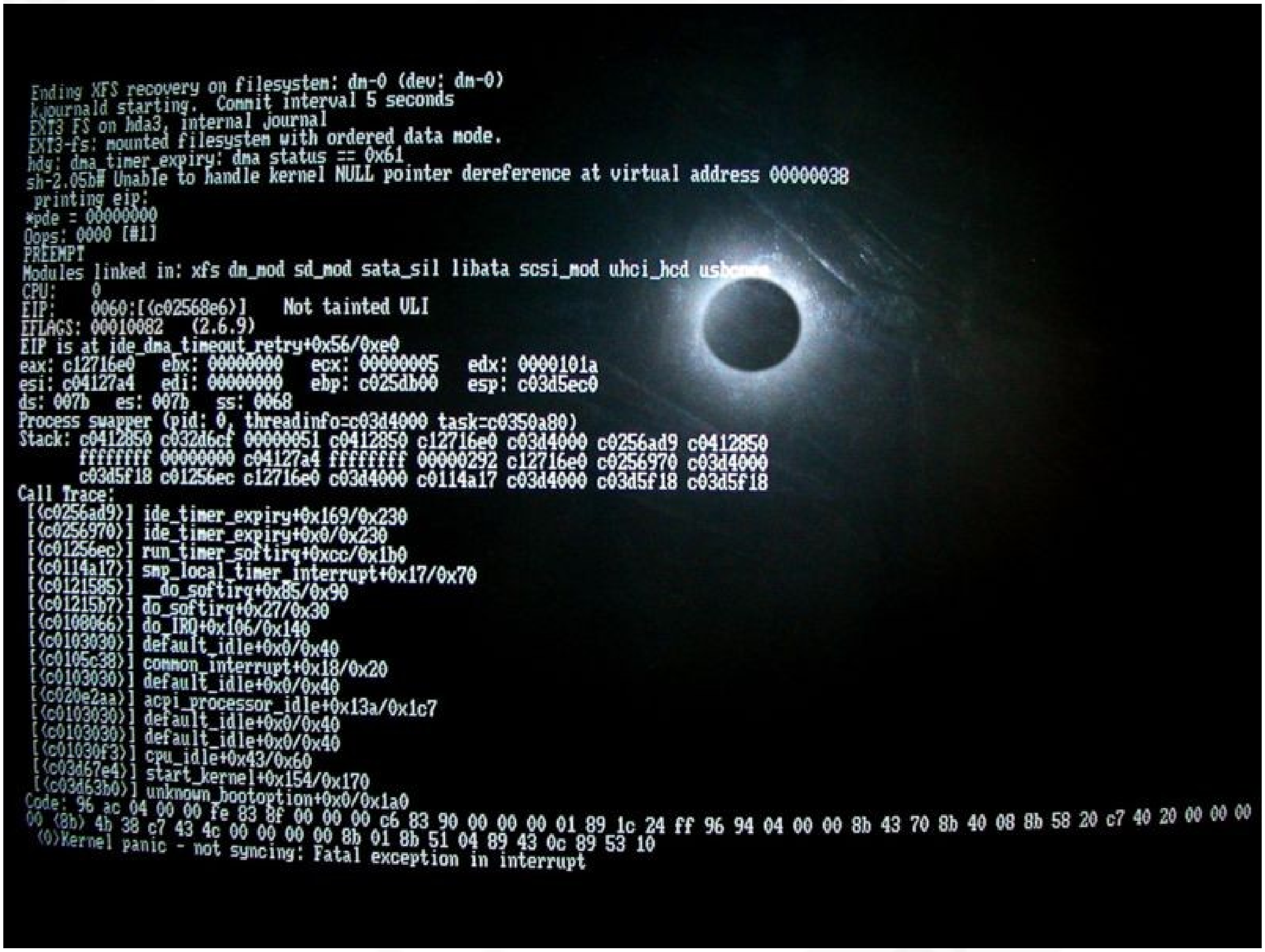
Lab:
Write a kernel module that invokes WARN(), BUG(), etc.
You would be well advised to try it out in a guest VM!

Kernel Panic

A kernel panic occours when the kernel encounters an error condition that it cannot recover from, i.e., it's a fatal condition. The system will need to be restarted. Generally, on a stable system, a kernel panic will be a rare event; hardware failure is generally the cause of a kernel panic.

Also, often during (BSP/platform) development, the kernel panics at boot time when it cannot find it's root filesystem (and therefore cannot mount it and continue).

Sidebar



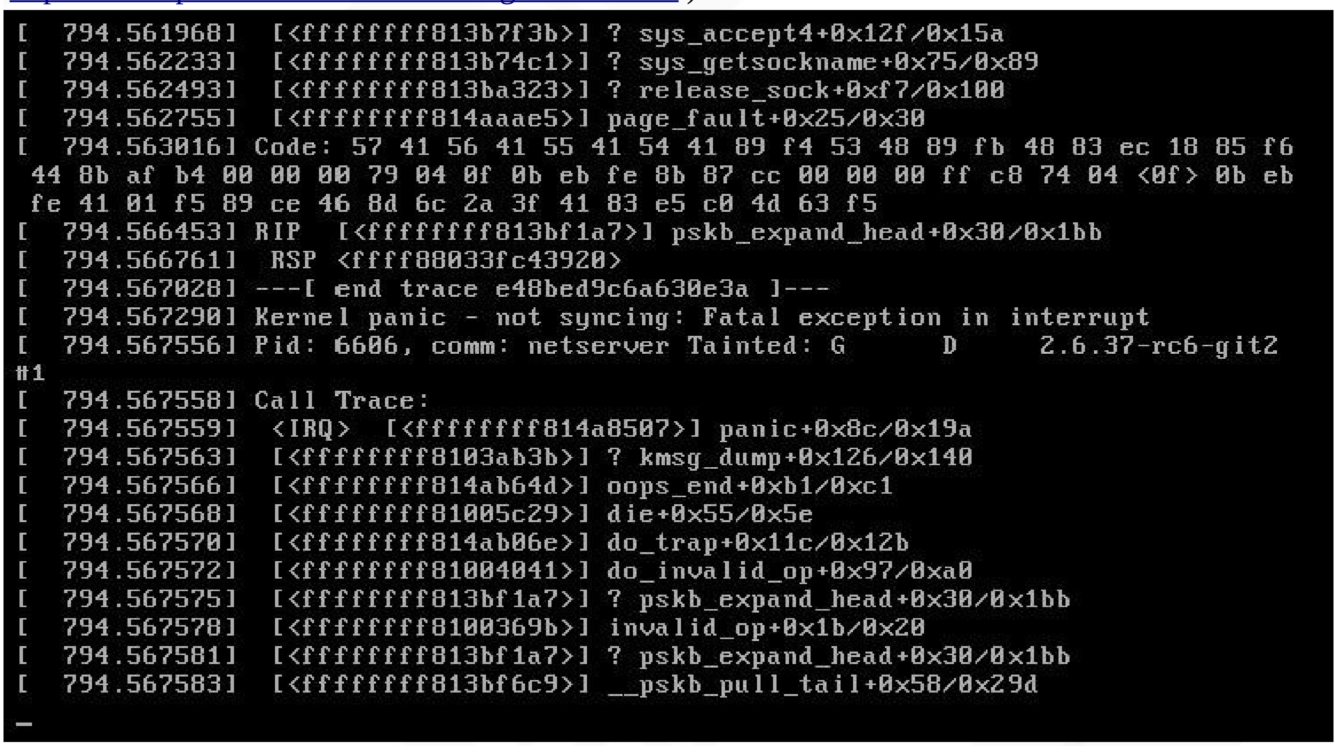
<< See picture : Linux 2.6 kernel panic caused by failing hard drive.
filename: linux_kernel_panic-v2-wikipedia.jpg >>

Source : Edited version of http://commons.wikimedia.org/wiki/Image:Linux_kernel_panic.jpg

Date : Feb 2007

Author : Dolda2000, Adamantios

For your enjoyment, here's another panic pic! (source: http://www.spinics.net/lists/netdev/msg150544.html)



```
[  794.561968]  [<ffffffff813b7f3b>] ? sys_accept4+0x12f/0x15a
[  794.562233]  [<ffffffff813b74c1>] ? sys_getsockname+0x75/0x89
[  794.562493]  [<ffffffff813ba323>] ? release_sock+0xf7/0x100
[  794.562755]  [<ffffffff814aaae5>] page_fault+0x25/0x30
[  794.563016] Code: 57 41 56 41 55 41 54 41 89 f4 53 48 89 fb 48 83 ec 18 85 f6
 44 8b af b4 00 00 00 79 04 0f 0b eb fe 8b 87 cc 00 00 00 ff c8 74 04 <0f> 0b eb
 fe 41 01 f5 89 ce 46 8d 6c 2a 3f 41 83 e5 c0 4d 63 f5
[  794.566453] RIP  [<ffffffff813bf1a7>] pskb_expand_head+0x30/0x1bb
[  794.566761]  RSP <ffff88033fc43920>
[  794.567028] ---[ end trace e48bed9c6a630e3a ]---
[  794.567290] Kernel panic - not syncing: Fatal exception in interrupt
[  794.567556] Pid: 6606, comm: netserver Tainted: G      D     2.6.37-rc6-git2
#1
[  794.567558] Call Trace:
[  794.567559]  <IRQ>  [<ffffffff814a8507>] panic+0x8c/0x19a
[  794.567563]  [<ffffffff8103ab3b>] ? kmsg_dump+0x126/0x140
[  794.567566]  [<ffffffff814ab64d>] oops_end+0xb1/0xc1
[  794.567568]  [<ffffffff81005c29>] die+0x55/0x5e
[  794.567570]  [<ffffffff814ab06e>] do_trap+0x11c/0x12b
[  794.567572]  [<ffffffff81004041>] do_invalid_op+0x97/0xa0
[  794.567575]  [<ffffffff813bf1a7>] ? pskb_expand_head+0x30/0x1bb
[  794.567578]  [<ffffffff8100369b>] invalid_op+0x1b/0x20
[  794.567581]  [<ffffffff813bf1a7>] ? pskb_expand_head+0x30/0x1bb
[  794.567583]  [<ffffffff813bf6c9>] __pskb_pull_tail+0x58/0x29d
_
```

<<

UPDATE! QR codes on kernel panic from 6.12 – an optional feature

Source …

The kernel panic QR code is a powerful and important tool for figuring out what caused the panic, especially as the current kernel panic screen tends to cut-off the kernel panic message. By adding a QR code, one could just grab their phone and scan the QR code to review the log and see what triggered the panic. ...

>>

The kernel panic code is at *kernel/panic.c* .

```
/**
 *  panic - halt the system
 *  @fmt: The text string to print
 *
 *  Display a message, then perform cleanups.
 *
 *  This function never returns.
 */
void panic(const char *fmt, ...)
{
...
}
```

Panic Handlers

Setting up your own panic handler using the kernel panic chain notifier mechanism
[Notes below adapted from "Building Embedded Linux Systems" by Karim Yaghmour, O'Reilly.]

!Note!

It *is* possible to use this technique (described below) to setup a panic handler within a kernel module. However, we need to use some wrapper functions to register the notifier chain mechanism that we are registering a new handler. This is as only the wrapper functions are exported - as **EXPORT_SYMBOL_GPL** - which also implies we need to release under the GPL license (and possibly another – dual licensing). See the source below.

Additional Resources

[The Crux of Linux Notifier Chains, LFY , Reghupathy, Jan 2009]

[Notification Chains in Linux Kernel - Part 01]

The only means of recovery in case of a kernel panic is a complete system reboot. For this reason, the kernel accepts a boot parameter that indicates the number of seconds it should wait after a kernel panic to reboot. If you would like the kernel to reboot one second after a kernel panic, for instance, you would pass the following sequence as part of the kernel's boot parameters: panic=1.

You can register your own panic function with the kernel. This function will be called by the kernel's panic function in the event of a kernel panic and can be used to carry out such things as signaling an emergency.

<<

*In our case, where we want to try and debug the issue, we could install our own panic handler function which will show the stack and registers, by calling dump_stack() and/or show_registers(struct pt_regs *).*

>>

The list that holds the functions called by the kernel's own panic function is *panic_notifier_list*. The *notifier_chain_register* function is used to add an item to this list. Conversely, *notifier_chain_unregister* is used to remove an item from this list.

<<

You can add your own panic handler either in-tree or out of it via a kernel module. It does require using the GPL license though…

From my *Linux Kernel Debugging* book's GitHub repo, here's a nice and simple module that installs a panic handler (works for 5.14 and above as well):

https://github.com/PacktPublishing/Linux-Kernel-Debugging/tree/main/ch10/panic_notifier

>>

```
$ cat panic_handler_lkm.c
#define pr_fmt(fmt) "%s:%s(): " fmt, KBUILD_MODNAME, __func__
#include <linux/init.h>
#include <linux/module.h>
#include <linux/delay.h>

// see kernel commit f39650de687e35766572ac89dbcd16a5911e2f0a
#include <linux/version.h>
#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 14, 0)
#include <linux/panic_notifier.h>
#else
#include <linux/notifier.h>
#endif

/* The atomic_notifier_chain_[un]register() api's are GPL-exported! */
MODULE_LICENSE("Dual MIT/GPL");

/* Do what's required here for the product/project,
 * but keep it simple. Left essentially empty here..
 */
static void dev_ring_alarm(void)
{
      pr_emerg("!!! ALARM !!!\n");
}

static int mypanic_handler(struct notifier_block *nb, unsigned long val, void
*data)
{
      pr_emerg("\n************ Panic : SOUNDING ALARM ************\n\
val = %lu\n\
```

```
data(str) = \"%s\"\n", val, (char *)data);
      dev_ring_alarm();

      return NOTIFY_OK;
}

static struct notifier_block mypanic_nb = {
      .notifier_call = mypanic_handler,
//    .priority = INT_MAX
};

static int __init panic_notifier_lkm_init(void)
{
      atomic_notifier_chain_register(&panic_notifier_list, &mypanic_nb);
      pr_info("Registered panic notifier\n");

      /*
       * Make #if 1 to have this module panic all by itself :-)
       * Else, we use our ../cause_oops_panic.sh script to trigger an
       * Oops and kernel panic!
       */
#if 0
      mdelay(500);
      panic("Linux Kernel Debugging!");
#endif

      return 0;         /* success */
}

static void __exit panic_notifier_lkm_exit(void)
{
      atomic_notifier_chain_unregister(&panic_notifier_list, &mypanic_nb);
      pr_info("Unregistered panic notifier\n");
}

module_init(panic_notifier_lkm_init);
module_exit(panic_notifier_lkm_exit);
```

…

In case of kernel panic, your panic notification function is called back as part of the kernel's notification of all panic functions. In turn, this function calls on *dump_stack()* (or whatever).

In a similar fashion, one can use the *register_die_notifier()* API to register a handler for an Oops.

Sample Run – on a QEMU emulated ARMv7 running Linux 3.14.34

```
ARM / $ id -u
0
ARM / $ insmod pnc.ko
pnc_init:49 : Regd panic notifier.ARM / $
ARM / $ cat /proc/sys/kernel/sysrq
1              << all sysrq functionality enabled >>
ARM / $ echo c > /proc/sysrq-trigger   << trigger a crash/panic >>

SysRq : Trigger a crash
Unable to handle kernel NULL pointer dereference at virtual address 00000000
pgd = 8f180000
[00000000] *pgd=6fa41831, *pte=00000000, *ppte=00000000
Internal error: Oops: 817 [#1] SMP ARM
Modules linked in: pnc(O)
CPU: 0 PID: 594 Comm: sh Tainted: G           O 3.14.34 #2
task: 8f9ec480 ti: 8fac6000 task.ti: 8fac6000
PC is at sysrq_handle_crash+0x38/0x40
LR is at sysrq_handle_crash+0x30/0x40
pc : [<802b415c>]    lr : [<802b4154>]    psr: 60000093
sp : 8fac7f10  ip : 00000000  fp : 000f7294
r10: 00000000  r9 : 00000000  r8 : 00000008
r7 : 60000013  r6 : 00000063  r5 : 806ab940  r4 : 8069bc68
r3 : 00000000  r2 : 00000001  r1 : a0000093  r0 : 806c715c
Flags: nZCv  IRQs off  FIQs on  Mode SVC_32  ISA ARM  Segment user
Control: 10c53c7d  Table: 6f180059  DAC: 00000015
Process sh (pid: 594, stack limit = 0x8fac6238)
Stack: (0x8fac7f10 to 0x8fac8000)
7f00:                                     802b4124 802b4910 8fac6000 00000002
7f20: 00000001 00000000 00000000 8f878680 000f8650 802b4df0 00000000 8013c2b8
7f40: 8f1eba80 000f8650 8fac7f80 00000002 00000002 800eebc8 000f7294 8002423c
7f60: 00000003 00000000 00000000 8f1eba80 8f1eba80 00000002 000f8650 800ef1ac
7f80: 00000000 00000000 00200200 000f6c80 00000001 000f8650 00000004 8000e344
7fa0: 8fac6000 8000e1c0 000f6c80 00000001 00000001 000f8650 00000002 00000000
7fc0: 000f6c80 00000001 000f8650 00000004 00000020 000f72a8 000f7274 000f7294
7fe0: 00000000 7eb1c624 0000f718 76e987ec 60000010 00000001 00000000 00000000
[<802b415c>] (sysrq_handle_crash) from [<802b4910>] (__handle_sysrq+0xb0/0x17c)
[<802b4910>] (__handle_sysrq) from [<802b4df0>] (write_sysrq_trigger+0x38/0x48)
[<802b4df0>] (write_sysrq_trigger) from [<8013c2b8>] (proc_reg_write+0x58/0x80)
[<8013c2b8>] (proc_reg_write) from [<800eebc8>] (vfs_write+0xac/0x188)
[<800eebc8>] (vfs_write) from [<800ef1ac>] (SyS_write+0x40/0x94)
[<800ef1ac>] (SyS_write) from [<8000e1c0>] (ret_fast_syscall+0x0/0x30)
Code: 0a000000 e12fff33 e3a03000 e3a02001 (e5c32000)
---[ end trace d5cbf74ce6b268eb ]---
Kernel panic - not syncing: Fatal exception
mypanic_handler:...            << our panic handler is called back! >>
Panic !ALARM! ...
```

Additional Resources

Determining cause of Linux kernel panic – on stackexchange

Linux kernel oops on Wikipedia

Kernel Documentation | oops-tracing.txt

Linux kernel OOPS debugging – a real-world example

Debugging Linux Kernel Lockup / Panic / Oops

Kernel Oops Howto

Watchdog Timer

A watchdog is a mechanism to periodically detect that the system is in a healthy state, and if it is deemed not to be, to reboot it.

This is achieved by setting up a (kernel) timer (to say, 60 seconds timeout). If all's well, a watchdog daemon process will consistently cancel and subsequently reenable the timeout; this is known as "petting / feeding the dog". If the daemon does not (due to something going badly wrong and, say, the daemon itself dying/getting killed), the watchdog is annoyed – *not petting / feeding me, I'll show you!* - and reboots the system!

What's watchdog "bark and bite"?, Quora, Aug 2014:
"… The process of expiry of a watch dog timer, which occurs when the watch dog timer is not feed/restarted before the time out period, is called BITE.

When the watchdog timer expires, it sends a signal to the micro controller or the PMU to restart the device as per design. This is what is often referred to as the watch dog BARK."

A pure software watchdog implementation will not be protected against kernel bugs and faults; a hardware watchdog (which latches into the board reset circuitry) will always be able to reboot the system as and when required.

An article explains the details:
Using the Watchdog Timer in Linux

Look up *Documentation/watchdog* too.

A case of a (real, although simple) kernel module bug ! (on x86_64) :-)

Seen when developing a kernel driver for an opensource project – DEVMEM_RW (Device Memory Read-Write) – that I (Kaiwan NB) host and maintain on GitHub:

```
/*
 * rwmem.c
 *
 * Part of the DEVMEM-RW opensource project - a simple
 * utility to read / write [I/O] memory and display it.
 * This is the kernel driver.
 *
 * Project home:
 * https://github.com/kaiwan/device-memory-readwrite
 *
 * Pl see detailed overview and usage PDF doc here:
 * https://github.com/kaiwan/device-memory-readwrite/blob/master/Devmem_HOWTO.pdf
 *
 * License: Dual GPL/MIT.
 * Author: Kaiwan N Billimoria
 *         kaiwanTECH.
 */

...

static int __init rwmem_init_module(void)
{

...

      dbgfs_parent = setup_debugfs_entries();
      if (!dbgfs_parent) {
            pr_alert("%s: debugfs setup failed, aborting...\n", DRVNAME);
            res = PTR_ERR(dbgfs_parent);
            return res;
      }

      // If no IO base start address specified, we're done for now
      if (!iobase_start || !iobase_len) {
            printk(KERN_WARNING
            "%s: Init done. IO base address NOT specified (or len invalid) as
module param; so, not performing any ioremap() ...\n",
                  DRVNAME);
            debugfs_remove_recursive(dbgfs_parent);
            return 0;
      }
...
}

static void __exit rwmem_cleanup_module(void)
{
```

```
	int i=0;

	debugfs_remove_recursive(dbgfs_parent);

	if (iobase_start) {
		iounmap (iobase);
		release_mem_region (iobase_start, iobase_len);
	}

...

}
```

The OOPS:

```
...
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.132739] BUG: unable to handle kernel NULL pointer dereference at 00000000000000a8
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.133409] IP: [<ffffffff817f7ef6>] mutex_lock+0x16/0x40
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.133994] PGD 0
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.134509] Oops: 0002 [#3] SMP
<<
Oops bitmask = 0002 = 0010 binary => Kernel-mode, write, no page found      [see earlier notes]
>>
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.134972] Modules linked in: devmem_rw(OE-) rwmem(OE) vm_img_lkm(OE) xt_CHECKSUM iptable_mangle ipt_MASQUERADE nf_nat_masquerade_ipv4 iptable_nat nf_conntrack_ipv4 nf_defrag_ipv4 nf_nat_ipv4 nf_nat nf_conntrack xt_tcpudp bridge stp llc iptable_filter ip_tables x_tables ntfs pci_stub vboxpci(OE) vboxnetadp(OE) vboxnetflt(OE) vboxdrv(OE) vboxsf(OE) snd_intel8x0 crct10dif_pclmul crc32_pclmul snd_ac97_codec ac97_bus aesni_intel aes_x86_64 snd_pcm lrw snd_seq_midi gf128mul snd_seq_midi_event glue_helper snd_rawmidi snd_seq ablk_helper snd_seq_device cryptd snd_timer input_leds snd serio_raw soundcore joydev 8250_fintek i2c_piix4 mac_hid netconsole configfs parport_pc ppdev lp parport autofs4 hid_generic usbhid hid vboxvideo(OE) psmouse ahci libahci syscopyarea sysfillrect sysimgblt ttm video vboxguest(OE) e1000 drm_kms_helper drm pata_acpi [last unloaded: devmem_rw]
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.140363] CPU: 0 PID: 10949 Comm: rmmod Tainted: G      D W  OE   4.2.0-42-generic #49-Ubuntu      << the process context caught in the Oops >>
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.140896] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.141404] task: ffff8800359d2c40 ti: ffff88007a714000 task.ti: ffff88007a714000
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.141968] RIP: 0010:[<ffffffff817f7ef6>]  [<ffffffff817f7ef6>] mutex_lock+0x16/0x40
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.142593] RSP: 0018:ffff88007a717e78  EFLAGS: 00010246
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.143095] RAX: 0000000000000000 RBX: 00000000000000a8 RCX: 0000000000000008
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.143690] RDX: 0000000080000000 RSI: ffffffffc044b580 RDI: 00000000000000a8
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.144194] RBP: ffff88007a717e88 R08: 0000000000000000 R09: 0000000000000077
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.144700] R10: 8080808080808080 R11: 0000000000000000 R12: ffff8800514160a0
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.145205] R13: ffff880051416058 R14: ffff880051416000 R15: ffff880051416000
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.145709] FS:  00007f623e339700(0000) GS:ffff88007fc00000(0000) knlGS:0000000000000000
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.146225] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.146726] CR2: 00000000000000a8 CR3: 000000007a734000 CR4:
```

```
00000000000406f0
                                         << CR2: virtual address lookup that triggered the fault >>
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.147271] Stack:
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.147786]  0000000000000010 ffffffffc044b240 ffff88007a717ed8
ffffffff8130d735
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.148403]  ffff8800359d2c40 ffff88007a717ef8 00007ffe3360a530
ffffffffc044b240
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.149053]  00007ffe3360a530 00007ffe3360a8f1 0000000000000000
00005587f7a6e1d0
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.149624] Call Trace:
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.150156]  [<ffffffff8130d735>] debugfs_remove_recursive+0x65/0x1c0
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.150691]  [<ffffffffc044988f>] rwmem_cleanup_module+0x10/0x781
[devmem_rw]
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.151182]  [<ffffffff81102bf5>] SyS_delete_module+0x1b5/0x210
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.151743]  [<ffffffff817fa072>] entry_SYSCALL_64_fastpath+0x16/0x75
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.152231] Code: ff 31 c0 87 03 83 f8 01 0f 85 5e ff ff ff eb d5 e8 80 4a
88 ff 0f 1f 44 00 00 55 48 89 e5 53 48 89 fb 48 83 ec 08 e8 ba e2 ff ff <3e> ff 0b 79 08 48 89 df e8 bd fe ff ff
65 48 8b 04 25 80 b9 00
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.154617] RIP  [<ffffffff817f7ef6>] mutex_lock+0x16/0x40
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.155143]  RSP <ffff88007a717e78>
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.155688] CR2: 00000000000000a8
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.156310] fbcon_switch: detected unhandled fb_set_par error, error code
-16
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.157588] fbcon_switch: detected unhandled fb_set_par error, error code
-16
Oct 11 08:04:31 Seawolf-VA kernel: [ 6912.158810] ---[ end trace f785db06dec3b907 ]---
...
```

Look carefully at the code snippet (reproduced below for convenience) :

```
...

static int __init rwmem_init_module(void)
{

...

	dbgfs_parent = setup_debugfs_entries();
	if (!dbgfs_parent) {
		pr_alert("%s: debugfs setup failed, aborting...\n", DRVNAME);
		res = PTR_ERR(dbgfs_parent);
		return res;
	}

	// If no IO base start address specified, we're done for now
	if (!iobase_start || !iobase_len) {
		printk(KERN_WARNING
		"%s: Init done. IO base address NOT specified (or len invalid) as
module param; so, not performing any ioremap() ...\n",
			DRVNAME);
		debugfs_remove_recursive(dbgfs_parent);     << the bug: because, at times,
the if condition above would be true leading to the debug_remove_recursive() being invoked here ...
>>
```

```
                        << Fix: the above code line was removed >>
            return 0;
      }
...
}

static void __exit rwmem_cleanup_module(void)
{
      int i=0;

      debugfs_remove_recursive(dbgfs_parent);    << ... and here! the second time
                                              it's attempted is when the Oops triggered ! >>

      if (iobase_start) {
            iounmap (iobase);
            release_mem_region (iobase_start, iobase_len);
      }
...

}
```

The bug, is that we might inadvertently release the debugfs pointer in the init code itself, and then attempt to (again) release the same pointer in the module exit code!

Appendix A :: ARM-32 (AArch32) [D]FSR Register Details

[D]FSR : [Data] Fault Status Register value

(Usually) the common code, that prints the Oops diagnostics and ultimately *panic*s is:

```
[...]
 show_pte(mm, addr);
 die("Oops", regs, fsr);

void die(const char *str, struct pt_regs *regs, int err);
void arm_notify_die(const char *str, struct pt_regs *regs,
                 struct siginfo *info, unsigned long err, unsigned long trap);
```

(usually 'err' is '[d]fsr' value).

The OS sets up 'hooks' for different types of faults, the signal to send (if in usermode), the error printk to emit, etc.

arch/arm/mm/fault.h

```
/*
 * Fault status register encodings.  We steal bit 31 for our own purposes.
 */
#define FSR_LNX_PF              (1 << 31)    << execute allowed >>
#define FSR_WRITE               (1 << 11)    << write allowed >>
#define FSR_FS4                 (1 << 10)
#define FSR_FS3_0               (15)
#define FSR_FS5_0               (0x3f)

…

struct fsr_info {
        int     (*fn)(unsigned long addr, unsigned int fsr, struct pt_regs *regs);
        int     sig;
        int     code;
        const char *name;
};
```

The hooks are here, in *arch/arm/mm/fsr-2level.c* and *arch/arm/mm/fsr-3level.c*:

arch/arm/mm/fsr-2level.c

```
static struct fsr_info fsr_info[] = {
   /*
    * The following are the standard ARMv3 and ARMv4 aborts.  ARMv5
    * defines these to be "precise" aborts.
    */
   { do_bad,  SIGSEGV, 0,           "vector exception"                 },
   { do_bad,  SIGBUS,  BUS_ADRALN,  "alignment exception"              },
   { do_bad,  SIGKILL, 0,           "terminal exception"               },
```

```
   { do_bad,  SIGBUS,  BUS_ADRALN,    "alignment exception"              },
   { do_bad,  SIGBUS,  0,             "external abort on linefetch"      },
   { do_translation_fault, SIGSEGV, SEGV_MAPERR, "section translation fault"  },
   { do_bad,  SIGBUS,  0,             "external abort on linefetch"      },
   { do_page_fault, SIGSEGV, SEGV_MAPERR,   "page translation fault"     },
   { do_bad, SIGBUS,  0,             "external abort on non-linefetch"  },
   { do_bad, SIGSEGV, SEGV_ACCERR,   "section domain fault"             },
   { do_bad, SIGBUS,  0,             "external abort on non-linefetch"  },
   { do_bad, SIGSEGV, SEGV_ACCERR,   "page domain fault"                },
   { do_bad, SIGBUS,  0,             "external abort on translation"    },
   { do_sect_fault, SIGSEGV, SEGV_ACCERR,   "section permission fault"  },
   { do_bad, SIGBUS,  0,             "external abort on translation"    },
   { do_page_fault, SIGSEGV, SEGV_ACCERR,   "page permission fault"     },
[...]
```

The hooks are setup via *arch/arm/mm/fault.c:hook_fault_code()*.

The ARM Data Fault Status Register (DFSR)

The DFSR is a coprocessor 15 (CP-15) 32-bit register, of which the LSB 4 bits are the FSR encoding. Lookup the Technical Reference Manual (TRM) for the ARM architecture type (ARMv5, v6, v7, v8) appropriate to your target board. Below, we show relevant excerpts from the TRM for the ARMv7.

***Source:* ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition**

<<

Accessing the DFSR

To access the DFSR, software reads or writes the CP15 registers with <opc1> set to 0, <CRn> set to c5, <CRm> set to c0, and <opc2> set to 0. For example:

```
MRC p15, 0, <Rt>, c5, c0, 0      ; Read DFSR into Rt
MCR p15, 0, <Rt>, c5, c0, 0      ; Write Rt to DFSR
```

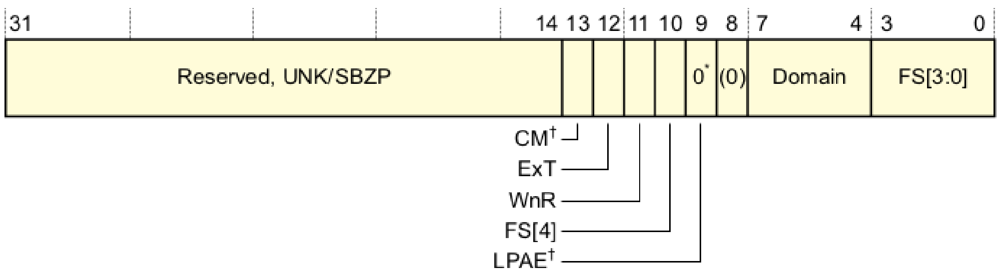
...

>>

Note- The DFSR register description table below applies to "short-descriptor translation table format" implying no LPAE (Large Physical Address Extension).
Page # 1561

DFSR format when using the Short-descriptor translation table format

In a VMSAv7 implementation that does not include the Large Physical Address Extension, or in an implementation that includes the Large Physical Address Extension when address translation is using the Short-descriptor translation table format, the DFSR bit assignments are:



† Only on an implementation that includes the Large Physical Address Extension. For more information, see the field description.

* Returned value, but might be overwritten, because the bit is RW.

Bits[31:14] Reserved, UNK/SBZP.

Bits[31:14] Reserved, UNK/SBZP.

CM, bit[13], if implementation includes the Large Physical Address Extension

Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maintenance operation generated the fault. The possible values of this bit are:

0 Abort not caused by a cache maintenance operation.

1 Abort caused by a cache maintenance operation.

On an asynchronous fault, this bit is UNKNOWN .

Bit[13], if implementation does not include the Large Physical Address Extension

Reserved, UNK/SBZP.

ExT, bit[12] External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

In an implementation that does not provide any classification of external aborts, this bit is UNK/SBZP.

WnR, bit[11] Write not Read bit. On a synchronous exception, indicates whether the abort was caused by a write or a read access. The possible values of this bit are:

0 Abort caused by a read access.

1 Abort caused by a write access.

For synchronous faults on CP15 cache maintenance operations, including the address translation operations, this bit always returns a value of 1.

This bit is UNKNOWN on:

- an asynchronous Data Abort exception
- a Data Abort exception caused by a debug exception.

FS, bits[10, 3:0]

Fault status bits. For the valid encodings of these bits when using the Short-descriptor translation table format, see Table B3-23 on page B3-1415 << *follows* >>. All encodings not shown in the table are reserved.

LPAE, bit[9], if the implementation includes the Large Physical Address Extension

On taking a Data Abort exception, this bit is set to 0 to indicate use of the Short-descriptor translation table formats.
Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation. Unless the register has been updated to report a fault, a subsequent read of the register returns the value written to it.

Bit[9], if the implementation does not include the Large Physical Address Extension

Reserved, UNK/SBZP.

Bit[8] Reserved, UNK/SBZP.

Domain, bits[7:4]

The domain of the fault address.
ARM deprecates any use of this field, see The Domain field in the DFSR on page B3-1415.
This field is UNKNOWN on a Data Abort exception:

- caused by a debug exception
- caused by a Permission fault in an implementation includes the Large Physical Address Extension.

…

[P.T.O.] →

FSR Bits Interpretation (for above non-LPAE case) *[page # 1415]*

Table B3-23 Short-descriptor format FSR encodings

FS	Source		Notes
00001	Alignment fault		DFSR only. Fault on first lookup
00100	Fault on instruction cache maintenance		DFSR only
01100 01110	Synchronous external abort on translation table walk	First level Second level	-
11100 11110	Synchronous parity error on translation table walk	First level Second level	-
00101 00111	Translation fault	First level Second level	MMU fault
00011[a] 00110	Access flag fault	First level Second level	MMU fault
01001 01011	Domain fault	First level Second level	MMU fault
01101 01111	Permission fault	First level Second level	MMU fault
00010	Debug event		See *About debug events* on page C3-2036
01000	Synchronous external abort		-
10000	TLB conflict abort		See *TLB conflict aborts* on page B3-1380
10100	IMPLEMENTATION DEFINED		Lockdown
11010	IMPLEMENTATION DEFINED		Coprocessor abort
11001	Synchronous parity error on memory access		-
10110	Asynchronous external abort[b]		DFSR only
11000	Asynchronous parity error on memory access[c]		DFSR only

a. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in VMSAv7 mean there should be no possibility of confusing the new use of this encoding with its previous use

b. Including asynchronous data external abort on translation table walk or instruction fetch.

c. Including asynchronous parity error on translation table walk.

The above *oops_simple.c* simple kernel module will of cause cause a crash in kernel mode (due to it attempting to reference non-existant memory). We compile for, and run the kernel module on an (Qemu emulated) ARM Versatile Express CA-9 platform – an **ARMv7** processor:

```
ARM # insmod oops_simple.ko
oops_simple: loading out-of-tree module taints kernel.
oops_simple_init:45 : Hello, about to Oops!
Unhandled fault: page domain fault (0x81b) at 0x00000024
pgd = 9ee8c000
[00000024] *pgd=7ee3d831, *pte=00000000, *ppte=00000000
Internal error: : 81b [#1] SMP ARM
Modules linked in: oops2(O+)
CPU: 0 PID: 743 Comm: insmod Tainted: G           O    4.9.1 #4
Hardware name: ARM-Versatile Express
task: 9f53b980 task.stack: 9ee7a000
PC is at oops2_init+0x50/0x5c [oops_simple]
LR is at console_unlock+0x5b0/0x624
pc : [<7f002050>]    lr : [<8016ef0c>]    psr: 60040013
sp : 9ee7bd90  ip : 9ee7bc80  fp : 9ee7bd9c
r10: 80a03008  r9 : 9ee43ca4  r8 : 00000001
r7 : ffffe000  r6 : 7f002000  r5 : 00000000  r4 : 80a03008
r3 : 00000000  r2 : 0000abcd  r1 : 60040013  r0 : 00000000
Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment none
Control: 10c5387d  Table: 7ee8c059  DAC: 00000051
Process insmod (pid: 743, stack limit = 0x9ee7a210)
Stack: (0x9ee7bd90 to 0x9ee7c000)
bd80:                                     9ee7be1c 9ee7bda0 80101d34 7f00200c
bda0: 9ee43ca4 80a03008 9ee7bdcc 9ee7bdb8 806c1364 806c09e0 00000001 024000c0
bdc0: 9ee7bde4 9ee7bdd0 806c13c8 806c1354 9f401f00 024000c0 9ee7be1c 9ee7bde8
bde0: 8024e46c 806c1388 00000001 00000017 9ee43d80 00040901 7f000140 00000001
be00: 7f000140 9ee43d80 00000001 9ee43ca4 9ee7be44 9ee7be20 801ff884 80101cdc
be20: 9ee7be44 9ee7be30 9ee7bf34 00000001 7f000140 9ee43c80 9ee7bf2c 9ee7be48
be40: 8019f188 801ff81c 7f00014c 00007fff 7f000140 8019c5e4 a533cfff a533d000
be60: 807bb7c8 807bb7a0 80a03008 807bb8f0 807bb794 00000000 80703534 7f00014c
be80: 00000000 7f000188 8019c02c 8019bf64 9ee7beb4 9ee7bea0 8023f4d0 8023e7c0
bea0: 00000017 9ee43d80 00000000 00000000 00000000 00000000 00000000 00000000
bec0: 6e72656b 00006c65 00000000 00000000 00000000 00000000 00000000 00000000
bee0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00040901
bf00: 024002c2 000063ec 000d9474 a533c3ec ffffe000 000b0948 00000000 00000051
bf20: 9ee7bfa4 9ee7bf30 8019f918 8019d4d0 80a59598 a5326000 000163ec a533bdfc
bf40: a533bc70 a5337024 00000344 00000394 00000000 00000000 00000000 000005c0
bf60: 00000023 00000024 00000011 00000000 0000000e 00000000 801083c4 000163ec
bf80: 756e694c 00000078 00000080 801083c4 9ee7a000 00000000 00000000 9ee7bfa8
bfa0: 80108220 8019f7c8 000163ec 756e694c 000c3088 000163ec 000b0948 76e65048
bfc0: 000163ec 756e694c 00000078 00000080 00000001 7ee46e3c 76fea000 0009e29a
bfe0: 7ee46b00 7ee46af0 0002808f 76ee9a82 80040030 000c3088 7fffd861 7fffdc61
[<7f002050>] (oops2_init [oops2]) from [<80101d34>] (do_one_initcall+0x64/0x1ac)
[<80101d34>] (do_one_initcall) from [<801ff884>] (do_init_module+0x74/0x1e4)
[<801ff884>] (do_init_module) from [<8019f188>] (load_module+0x1cc4/0x22f8)
[<8019f188>] (load_module) from [<8019f918>] (SyS_init_module+0x15c/0x17c)
[<8019f918>] (SyS_init_module) from [<80108220>] (ret_fast_syscall+0x0/0x1c)
Code: e30a2bcd e3473f00 e3a00000 e5933000 (e5832024)
---[ end trace 2ee827bca99f88fb ]---
Segmentation fault
ARM #
```

Look carefully at the above Oops output:

```
Unhandled fault: page domain fault (0x81b) at 0x00000024
pgd = 9ee8c000
[00000024] *pgd=7ee3d831, *pte=00000000, *ppte=00000000
Internal error: : 81b [#1] SMP ARM
```

Now lookup our “ARM Oops Kernel Messages” table above to find the matching row; it’s this one:

Dispatch a data abort to the relevant handler.	do_DataAbort	`pr_alert("Unhandled fault: %s (0x%03x) at 0x%08lx\n", inf->name, fsr, addr);`	Y	Y arm_notify_die(...)

This clearly reveals that in the line

```
Unhandled fault: page domain fault (0x81b) at 0x00000024
```

Several of the ‘highlighted in bold’ printk’s in the above Oops come from here in the source:

```
arch/arm/kernel/traps.c :
static int __die(const char *str, int err, struct pt_regs *regs)
{
        struct task_struct *tsk = current;
        static int die_counter;
        int ret;

        pr_emerg("Internal error: %s: %x [#%d]" S_PREEMPT S_SMP S_ISA "\n",
                 str, err, ++die_counter);
                     << 'err' is the [D]FSR – the Fault Status Register >>
        /* trap and error numbers are mostly meaningless on ARM */
        ret = notify_die(DIE_OOPS, str, regs, err, tsk->thread.trap_no, SIGSEGV);
        if (ret == NOTIFY_STOP)
                return 1;

        print_modules();
        __show_regs(regs);
        pr_emerg("Process %.*s (pid: %d, stack limit = 0x%p)\n",
                 TASK_COMM_LEN, tsk->comm, task_pid_nr(tsk),
                   end_of_stack(tsk));

        if (!user_mode(regs) || in_interrupt()) {
                dump_mem(KERN_EMERG, "Stack: ", regs->ARM_sp,
                         THREAD_SIZE +
                              (unsigned long)task_stack_page(tsk));
                dump_backtrace(regs, tsk);
                dump_instr(KERN_EMERG, regs);
        }
```

```
        return 0;
}
```

Cause: Data Abort.
ARM Documentation on MMU Aborts:
… If the memory request that aborts is an instruction fetch, then a *Prefetch Abort* exception is raised if and when the processor attempts to execute the instruction corresponding to the aborted access.

If the aborted access is a data access or a cache maintenance operation, a *Data Abort* exception is raised.
All Data Aborts, and aborts caused by cache maintenance operations, cause the *Data Fault Status Register* (DFSR) to be updated so that you can determine the cause of the abort.
…

0x81b is the FSR value, and that `0x00000024` is the faulting virtual address ('addr').

0x81b in binary is `1000 0001.`

From the ARM technical manual pages displayed above (*DFSR Format no LPAE pg 1561*), we can see that bit 8 is a reserved bit, which we shall therefore ignore.
All possible bit settings of the 4-bit FSR register (*FSR Bits Interpretation (for above non-LPAE case) [page # 1415]).*

In our Oops above, the **FSR = 0x81 (= `1000 0001`)** ; thus the following bits are set:
Bit 8 : reserved (ignore)
Bit 1, i.e., `00001`

The very first line of the above table captures the 5 bits 00001:

FS	Source	Notes
00001	Alignment fault	DFSR only. Fault on first lookup

It's a memory fault - "Fault on first lookup" - implying the (virtual) memory address is invalid.

Ref:
what do these kernel panic errors mean? SO, Apr 2013.

AArch64: see the 'Analyzing a kernel Oops on AArch64' doc.

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here.