Understanding a Kernel Oops!

opensourceforu.com/2011/01/understanding-a-kernel-oops

Surya Prabhakar December 31, 2010

Developers

By

Surya Prabhakar
-

January 1, 2011
16
100547

Understanding a kernel panic and doing the forensics to trace the bug is considered a hacker's job. This is a complex task that requires sound knowledge of both the architecture you are working on, and the internals of the Linux kernel. Depending on type of error detected by the kernel, panics in the Linux kernel are classified as hard panics (Aiee!) and soft panics (Oops!). This article explains the workings of a Linux kernel 'Oops', helps to create a simple version, and then debug it. It is mainly intended for beginners getting into Linux kernel development, who need to debug the kernel. Knowledge of the Linux kernel, and C programming, is assumed.

An "Oops" is what the kernel throws at us when it finds something faulty, or an exception, in the kernel code. It's somewhat like the segfaults of user-space. An Oops dumps its message on the console; it contains the processor status and the CPU registers of when the fault occurred. The offending process that triggered this Oops gets killed without releasing locks or cleaning up structures. The system may not even resume its normal operations sometimes; this is called an unstable state. Once an Oops has occurred, the system cannot be trusted any further.

Let's try to generate an Oops message with sample code, and try to understand the dump.

Setting up the machine to capture an Oops

The running kernel should be compiled with `CONFIG_DEBUG_INFO`, and `syslogd` should be running. To generate and understand an Oops message, Let's write a sample kernel module, `oops.c`:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
static void create_oops() {
*( int *)0 = 0;
}
static int __init my_oops_init( void ) {
printk( "oops from the module\n" );
create_oops();
return (0);
}
static void __exit my_oops_exit( void ) {
printk( "Goodbye world\n" );
}
module_init(my_oops_init);
module_exit(my_oops_exit);
```

The associated `Makefile` for this module is as follows:

```
obj-m   := oops.o
KDIR    := /lib/modules/ $(shell uname -r) /build
PWD     := $(shell pwd )
SYM=$(PWD)
all:
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

Once executed, the module generates the following Oops:

```
BUG: unable to handle kernel NULL pointer dereference at (null)
```

```
IP: [<ffffffffa03e1012>] my_oops_init+0x12/0x21 [oops]

PGD 7a719067 PUD 7b2b3067 PMD 0

Oops: 0002 [#1] SMP

last sysfs file: /sys/devices/virtual/misc/kvm/uevent

CPU 1

Pid: 2248, comm: insmod Tainted: P           2.6.33.3-85.fc13.x86_64

RIP: 0010:[<ffffffffa03e1012>]  [<ffffffffa03e1012>]
my_oops_init+0x12/0x21 [oops]

RSP: 0018:ffff88007ad4bf08  EFLAGS: 00010292

RAX: 0000000000000018 RBX: ffffffffa03e1000 RCX: 00000000000013b7

RDX: 0000000000000000 RSI: 0000000000000046 RDI: 0000000000000246

RBP: ffff88007ad4bf08 R08: ffff88007af1cba0 R09: 0000000000000004

R10: 0000000000000000 R11: ffff88007ad4bd68 R12: 0000000000000000

R13: 00000000016b0030 R14: 0000000000019db9 R15: 00000000016b0010

FS:  00007fb79dadf700(0000) GS:ffff880001e80000(0000)
knlGS:0000000000000000

CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b

CR2: 0000000000000000 CR3: 000000007a0f1000 CR4: 00000000000006e0

DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000

DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400

Process insmod (pid: 2248, threadinfo ffff88007ad4a000, task
ffff88007a222ea0)

Stack:

ffff88007ad4bf38 ffffffff8100205f ffffffffa03de060 ffffffffa03de060

0000000000000000 00000000016b0030 ffff88007ad4bf78 ffffffff8107aac9

ffff88007ad4bf78 00007fff69f3e814 0000000000019db9 0000000000020000

Call Trace:

[<ffffffff8100205f>] do_one_initcall+0x59/0x154

[<ffffffff8107aac9>] sys_init_module+0xd1/0x230

[<ffffffff81009b02>] system_call_fastpath+0x16/0x1b

Code: <c7> 04 25 00 00 00 00 00 00 00 00 31 c0 c9 c3 00 00 00 00 00 00
00

RIP  [<ffffffffa03e1012>] my_oops_init+0x12/0x21 [oops]

RSP <ffff88007ad4bf08>
```

```
CR2: 0000000000000000
```

Understanding the Oops dump

Let’s have a closer look at the above dump, to understand some of the important bits of information.

```
BUG: unable to handle kernel NULL pointer dereference at (null)
```

The first line indicates a pointer with a NULL value.

```
IP: [<ffffffffa03e1012>] my_oops_init+0x12/0x21 [oops]
```

IP is the instruction pointer.

```
Oops: 0002 [#1] SMP
```

This is the error code value in hex. Each bit has a significance of its own:

- `bit 0` == 0 means no page found, 1 means a protection fault
- `bit 1` == 0 means read, 1 means write
- `bit 2` == 0 means kernel, 1 means user-mode
- `[#1]` — this value is the number of times the Oops occurred. Multiple Oops can be triggered as a cascading effect of the first one.

```
CPU 1
```

This denotes on which CPU the error occurred.

```
Pid: 2248, comm: insmod Tainted: P           2.6.33.3-85.fc13.x86_64
```

The `Tainted` flag points to `P` here. Each flag has its own meaning. A few other flags, and their meanings, picked up from `kernel/panic.c`:

- `P` — Proprietary module has been loaded.
- `F` — Module has been forcibly loaded.
- `S` — SMP with a CPU not designed for SMP.
- `R` — User forced a module unload.
- `M` — System experienced a machine check exception.
- `B` — System has hit bad_page.
- `U` — Userspace-defined naughtiness.
- `A` — ACPI table overridden.

- `W` — Taint on warning.

```
RIP: 0010:[<ffffffffa03e1012>]  [<ffffffffa03e1012>]
my_oops_init+0x12/0x21 [oops]
```

`RIP` is the CPU register containing the address of the instruction that is getting executed. `0010` comes from the code segment register. `my_oops_init+0x12/0x21` is the *<symbol> + the offset/length*.

```
RSP: 0018:ffff88007ad4bf08  EFLAGS: 00010292
RAX: 0000000000000018 RBX: ffffffffa03e1000 RCX: 00000000000013b7
RDX: 0000000000000000 RSI: 0000000000000046 RDI: 0000000000000246
RBP: ffff88007ad4bf08 R08: ffff88007af1cba0 R09: 0000000000000004
R10: 0000000000000000 R11: ffff88007ad4bd68 R12: 0000000000000000
R13: 00000000016b0030 R14: 0000000000019db9 R15: 00000000016b0010
```

This is a dump of the contents of some of the CPU registers.

```
Stack:
ffff88007ad4bf38 ffffffff8100205f ffffffffa03de060 ffffffffa03de060
0000000000000000 00000000016b0030 ffff88007ad4bf78 ffffffff8107aac9
ffff88007ad4bf78 00007fff69f3e814 0000000000019db9 0000000000020000
```

The above is the stack trace.

```
Call Trace:
[<ffffffff8100205f>] do_one_initcall+0x59/0x154
[<ffffffff8107aac9>] sys_init_module+0xd1/0x230
[<ffffffff81009b02>] system_call_fastpath+0x16/0x1b
```

The above is the call trace — the list of functions being called just before the Oops occurred.

```
Code: <c7> 04 25 00 00 00 00 00 00 00 00 31 c0 c9 c3 00 00 00 00 00 00
00
```

The `Code` is a hex-dump of the section of machine code that was being run at the time the Oops occurred.

Debugging an Oops dump

The first step is to load the offending module into the GDB debugger, as follows:

```
[root@DELL-RnD-India oops]# gdb oops.ko
GNU gdb (GDB) Fedora (7.1-18.fc13)
Reading symbols from /code/oops/oops.ko...done.
(gdb) add-symbol-file oops.o 0xffffffffa03e1000
add symbol table from file "oops.o" at
.text_addr = 0xffffffffa03e1000
```

Next, add the symbol file to the debugger. The `add-symbol-file` command's first argument is `oops.o` and the second argument is the address of the text section of the module. You can obtain this address from `/sys/module/oops/sections/.init.text` (where `oops` is the module name):

```
(gdb) add-symbol-file oops.o 0xffffffffa03e1000
add symbol table from file "oops.o" at
.text_addr = 0xffffffffa03e1000
(y or n) y
Reading symbols from /code/oops/oops.o...done.
```

From the `RIP` instruction line, we can get the name of the offending function, and disassemble it.

```
(gdb) disassemble my_oops_init
Dump of assembler code for function my_oops_init:
0x0000000000000038 <+0>:    push   %rbp
0x0000000000000039 <+1>:    mov    $0x0,%rdi
0x0000000000000040 <+8>:    xor    %eax,%eax
0x0000000000000042 <+10>:    mov    %rsp,%rbp
0x0000000000000045 <+13>:    callq  0x4a <my_oops_init+18>
0x000000000000004a <+18>:    movl   $0x0,0x0
0x0000000000000055 <+29>:    xor    %eax,%eax
0x0000000000000057 <+31>:    leaveq
0x0000000000000058 <+32>:    retq
End of assembler dump.
```

Now, to pin point the actual line of offending code, we add the starting address and the offset. The offset is available in the same `RIP` instruction line. In our case, we are adding `0x0000000000000038 + 0x012 =  0x000000000000004a`. This points to the `movl` instruction.

```
(gdb) list *0x000000000000004a
0x4a is in my_oops_init (/code/oops/oops.c:6).
1    #include <linux/kernel.h>
2    #include <linux/module.h>
3    #include <linux/init.h>
4
5    static void create_oops() {
6            *(int *)0 = 0;
7    }
```

This gives the code of the offending function.

References

The kerneloops.org website can be used to pick up a lot of Oops messages to debug. The Linux kernel documentation directory has information about Oops — `kernel/Documentation/oops-tracing.txt`. This, and numerous other online resources, were used while creating this article.