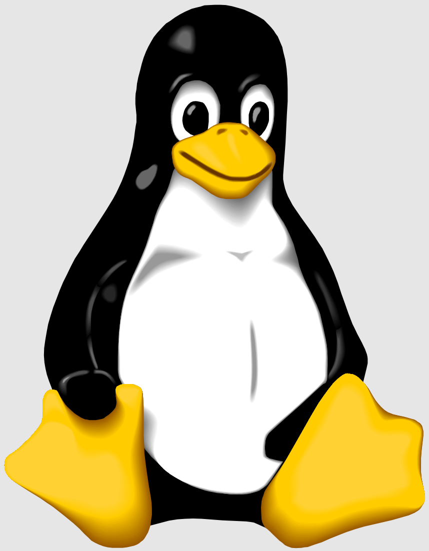
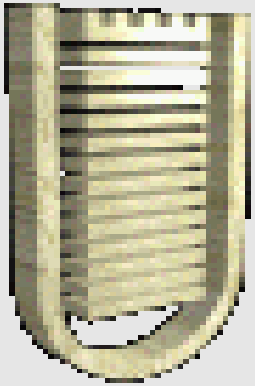
# Linux Kernel Hacking Free Course

3rd edition

G.Grilli, University of Rome “Tor Vergata”

# Profiling and Debugging





## Contents:

- Kernel profiling
  - Introduction to Oprofile
  - Oprofile features
  - Getting started with Oprofile
- Kernel debugging
  - Debugging by printing: printk loglevels and logging process
  - When the system seems to hang: the magic SysRQ key
  - Understanding an Oops output by example
- Useful debugging tools: netconsole and netcat

## What is profiling?

- Profiling is a formal summary or analysis of data, often in the form of a graph or table, representing distinctive performance features or characteristics
- Analyzing the performance of the Linux operating system and application code can be difficult due to unexpected interactions between the hardware and the software, but profiling is one way you can identify such performance problems
- The goal of the profilers is ..provides the percentage and number of samples collected for specified processor events such as the number of cache line misses, Transition Lookaside Buffer (TLB) misses, and so on

## Oprofile

- OProfile is one of several profiling and performance monitoring tools for Linux
- It consists of a loadable kernel module and a system daemon process that collects sample data from a running system (in 2.6 kernels it can be compiled as built-in feature)
- It takes advantage of the hardware performance counters available in today's microprocessors to enable profiling of the entire system
- OProfile is capable of profiling all code including the kernel, kernel modules, kernel interrupt handlers, system shared libraries, and the applications (symbols are retrieved into the System.map of the profiling kernel)

# Oprofile – features (1)

| | |
|---|---|
| ***unobtrusive*** | no special recompilations, wrapper libraries or the like are necessary<br>no kernel patch is needed (or built-in or simply a module) |
| ***system-wide profiling*** | all code running on the system is profiled, enabling analysis of system performance |
| ***performance counter support*** | enables collection of various low-level data, and association with particular sections of code |
| ***call-graph support*** | with an x86 2.6 kernel, OProfile can provide gprof-style call-graph profiling data |

## Oprofile – features (2)

***low overhead***

OProfile has a typical overhead of 1-8%, dependent on sampling frequency and workload

***post-profile analysis***

profile data can be produced on the function-level or instruction-level detail. Source trees annotated with profile information can be created. A hit list of applications and functions that take the most time across the whole system can be produced.

***system support***

OProfile works across a range of CPUs, include the Intel range, AMD's Athlon and AMD64 processors range, the Alpha, and more. OProfile will work against almost any 2.2, 2.4 and 2.6 kernels, and works on both UP and SMP systems from desktops to the scariest NUMAQ boxes.

## Oprofile – getting started (1)

if we want to profile the linux kernel, we must configure Oprofile this way:

```
# opcontrol --vmlinux=/boot/vmlinux-`uname -r`
```

instead, if you want to profile the application without the kernel:

```
# opcontrol --no-vmlinux
```

now we start the Oprofile daemon to start collecting profile data:

```
# opcontrol --start
```

## Oprofile – getting started (2)

before examining the results we must dump the collected data:

```
# opcontrol --dump
```

in this way we are ready to examine results collected before raising the dump command. Do not forget Oprofile is still capturing data!
In order to completely end the sampling process, use this command:

```
# opcontrol --shutdown
```

if for some reason you want to clear the profile data, at any time you can just do a reset with:

```
# opcontrol --reset
```

## Oprofile – getting started (3)

Once we have collected data of our running application, we can use the `opreport` command to generate a report:

```
# opreport

   CPU: CPU with timer interrupt, speed 0 Mhz
(estimated)
   Profiling through timer interrupt
              TIMER:0|
     samples|       %|
   ------------------
        3122 98.5791 no-vmlinux
          16  0.5052 libc.so.6
           8  0.2526 bash
           4  0.1263 ld-2.3.3.so
           4  0.1263 libgdk_pixbuf-2.0.so.0.400.9
           3  0.0947 libglib-2.0.so.0.400.6
                 ( . . . )
```

all profile data are related to modules and dynamic libraries

## Oprofile – getting started (4)

In this example we collected profile data related to the kernel image (we obtained a very detailed report by using the '-l' option to opreport):

```
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
samples  %        image name                app name                  symbol name
42301    99.0725  vmlinux                   vmlinux                 acpi_processor_idle
59        0.1382  anon (tgid:5973 range:0x8209000-0x88a5000) Xorg   (no symbols)
38        0.0890  libc-2.3.4.so             libc-2.3.4.so             (no symbols)
32        0.0749  opreport                  opreport                  (no symbols)
18        0.0422  libqt-mt.so.3.3.4         libqt-mt.so.3.3.4         (no symbols)
13        0.0304  libglib-2.0.so.0.800.4    libglib-2.0.so.0.800.4    (no symbols)
13        0.0304  libpango-1.0.so.0.1001.1 libpango-1.0.so.0.1001.1 (no symbols)
11        0.0258  libstdc++.so.5.0.7        libstdc++.so.5.0.7        (no symbols)
8         0.0187  gkrellm2                  gkrellm2                  (no symbols)
8         0.0187  ld-2.3.4.so               ld-2.3.4.so               (no symbols)
8         0.0187  libcairo.so.2.2.3         libcairo.so.2.2.3         (no symbols)
7         0.0164  libbfd-2.15.92.0.2.so     libbfd-2.15.92.0.2.so     (no symbols)

                                        ( . . . )
```

(Hint: try to use `opreport --symbols --show-address`)

## Oprofile – hint

You can use Oprofile even continuosly, dumping and resetting data every a certain amount of time (like the command top does):

```
# watch --interval=1 “opcontrol --dump && opreport --symbols \
--show-address -l /usr/src/linux-`uname -r`/vmlinux | \
head -n 20 ; opcontrol --reset”
```

## Linux kernel debugging - why?

There should be no reason to debug the kernel: it is the one part of the system we don't have to worry about because it always works **FALSE!**

- because the kernel is crashing and we don't know why
- because we are modifying the kernel according to a work or school project
- because a driver is not working as well as it should, or is not working at all
- because it is a good way to learn how the kernel works

## Debugging the kernel is an hard task

- the kernel source is BIG (millions of lines)
- the kernel is very complex (multithreaded, hardware-related, ...)
- there's no higher program that monitors it: kernel code cannot be easily executed under a debugger, nor can it be easily traced, because it is a set of functionalities not related to a specific process
  *("User Mode Linux" project addresses this problem)*

## The easiest way: debugging by printing

The most common debugging technique is monitoring. Usually, in applications programming this is done by calling `printf` at suitable points. Now you are debugging kernel code and you can accomplish the same goal with `printk`.

This function lets you classify messages according to their severity by associating different loglevels, or priorities, with the messages. To specify the loglevel you can use a macro which expands to a string.

Example:

```
printk(KERN_DEBUG "value of cpu_ptr: %i\n", cpu->nr);

printk(KERN_CRIT "critical error! ptr_value: %p\n", ptr);
```

## printk loglevels (1)

There are eight possible loglevels associated to `printk` and defined in `<linux/kernel.h>` header file.

- `KERN_EMERG` → used for emergency messages, usually before a system crash
- `KERN_ALERT` → used for serious problems, when it is needed quick response
- `KERN_CRIT` → critical condition, usually related to hardware or software failure
- `KERN_ERR` → used for conditions, usually related to hardware difficulties
- `KERN_WARNING` → used to warn about problematic situations that are not serious

## printk loglevels (2)

`KERN_NOTICE` → normal situations that requires notification

`KERN_INFO` → informational messages. Many drivers print information about the hardware they find at startup time at this level

`KERN_DEBUG` → used for kernel debugging phase only

- each string expanded by the macro represents a number ranging from 0 to 7, with smaller values representing higher priorities
- if you do not specify any value with printk, the default log level is equal to `DEFAULT_MESSAGE_LOGLEVEL` variable
- `klogd` and `syslogd` display only messages with priority less than or equal to the `DEFAULT_CONSOLE_LOGLEVEL` variable

## How to change the default loglevel

- through the `sys_syslog` system call
- kill `klogd` and then restart it with the `-c` option

From `klogd` version 2.1.31 on it is possible to read and modify the console loglevel using the text file `/proc/sys/kernel/printk`. The file hosts four integer values but we are interested in the first two: the current console loglevel and the default level for messages:

```
# echo 5 > /proc/sys/kernel/printk
```

after this command there will be displayed only messages from loglevel 0 to 4

```
# echo 8 > /proc/sys/kernel/printk
```

after this command there will be displayed all messages

## How the logging process works (1)

1) the printk function writes messages into a circular buffer that is LOG_BUF_LEN (defined in kernel/printk.c) bytes long

2) it then wakes any process that is waiting for messages, that is, any process that is sleeping in the syslog system call or that is reading /proc/kmsg

3) if the circular buffer fills up, printk wraps around and starts adding new data to the beginning of the buffer, overwriting the oldest data

4) if the klogd process is running, it retrieves kernel messages and dispatches them to syslogd, which in turn checks /etc/syslog.conf to find out how to deal with them

## How the logging process works (2)

5) If klogd isn't running, data remains in the circular buffer until someone reads it or the buffer overflows

Example of `/etc/syslog.conf`:

```
(...)

#Kernel logging
kern.=debug;kern.=info;kern.=notice   -/var/log/kernel/info
kern.=warn                            -/var/log/kernel/warnings
kern.err                               /var/log/kernel/errors

(...)
```

*( type* "`man syslog.conf`" *for further informations )*

## When the kernel doesn't respond: the magic SysRQ key

It is a 'magical' key combo you can hit which kernel will respond to regardless of whatever else it is doing, unless it is completely locked up.

To enable this feature you need to say "yes" to 'Magic SysRq key `(CONFIG_MAGIC_SYSRQ)` when configuring the kernel. This option is available starting from 2.1.x kernel version.

On Intel x86 architecture you can use the SysRQ by pressing the key combo:

> ALT + SysRQ + <command key>

If you can't find any key labeled in such way, remember that the 'SysRQ' key is also known as the 'Print Screen' key.

## The magic SysRQ key: command keys

| | |
|---|---|
| "R" | turns off keyboard raw mode and sets it to XLATE |
| "K" | kills all programs on the current virtual console |
| "B" | will immediately reboot the system without syncing or unmounting your disks |
| "O" | will shut your system off via APM (if configured and supported) |
| "S" | will attempt to sync all mounted filesystems |
| "U" | will attempt to remount all mounted filesystems read-only |
| "P" | will dump the current registers and flags to your console |

## The magic SysRQ key: command keys

| | |
|---|---|
| "T" | will dump a list of current tasks and their information to your console |
| "M" | will dump current memory info to your console |
| "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) |
| "E" | send a SIGTERM to all processes, except for init |
| "I" | send a SIGKILL to all processes, except for init |
| "L" | send a SIGKILL to all processes,including init (your system will be non-functional after this) |

## When the system crashes: understanding the Oops output (1)

The “Oops” is a dump of kernel stack and CPU state at an instant and it is shown by the kernel when a serious problem occurs.

This message can be sent to several destinations:

- **local console**
- **remote console**
  - through serial port
  - through tpc/ip with netcat utility or netconsole
- **kernel ring buffer** (`klogd` pulls it out and sends it to `syslogd`)

## When system crashes: understanding the Oops output (2)

```
Unable to handle kernel NULL pointer dereference at virtual address 0000000c
 EIP  = c0168c7b
 *pde = 00000000
 Oops: 0000 [#1]
 PREEMPT
 Modules linked in:                                                    (...)
 CPU:    0
 EIP:    0060:[<c0168c7b>]  Not tainted VLI
 EFLAGS: 00010246   (2.6.15khc06)
 EIP is at seq_printf+0x7/0x43
 eax: ca809f4c   ebx: 00000000  ecx: 00000000   edx: ca809f4c
 esi: 00000000   edi: ca809f4c  ebp: ca809f0c   esp: ca809f08
 ds: 007bes: 007b   ss: 0068
 Process cat (pid: 5986, threadinfo=ca808000 task=cfa46a50)
 Stack: 00000000 ca809f2c c010447b 00000000 c0393e60 00000000 (...)
 Call Trace:
  [<c0103273>] show_stack+0x7a/0x82
  [<c0103381>] show_registers+0xee/0x157
  [<c0103533>] die+0xd1/0x157
  [<c01102eb>] do_page_fault+0x385/0x4ae
  [<c0102f57>] error_code+0x4f/0x54
  [<c010447b>] show_interrupts+0x21/0x156
  [<c01687be>] seq_read+0xdd/0x24c
  [<c014b9ee>] vfs_read+0x88/0x128
  [<c014bcbc>] sys_read+0x3a/0x61
  [<c0102d2d>] syscall_call+0x7/0xb
 Code: eb 1c 88 1a 42 ff 45 0c 8b 45 0c 0f b6 18 84 db 74 05 3b 55 f0 72 91 2b
 17 31 c0 89 57 0c 5b 5b 5e 5f 5d c3 55 89 e5 53 8b 5d 08 <8b> 4b 0c 8b 53 (...)
```

**Unable to handle kernel NULL pointer dereference at virtual address 0000000c**

**first clue: the kernel was unable to handle a null pointer somewhere in the code**

# When system crashes: understanding the Oops output (3)

```
Unable to handle kernel NULL pointer dereference at virtual address 0000000c
 EIP  = c0168c7b
 *pde = 00000000
 Oops: 0000 [#1]
 PREEMPT
 Modules linked in: pcmcia firmware_class pcmcia_core ee                      ..)
 CPU:    0
 EIP:    0060:[<c0168c7b>]  Not tainted VLI
 EFLAGS: 00010246   (2.6.15khc06)
 EIP is at seq_printf+0x7/0x43
 eax: ca809f4c   ebx: 00000000  ecx: 00000000   edx: ca809f4c
 esi: 00000000   edi: ca809f4c  ebp: ca809f0c   esp: ca809f08
 ds: 007bes: 007b   ss: 0068
 Process cat (pid: 5986, threadinfo=ca808000 task=cfa46a50)
 Stack: 00000000 ca809f2c c010447b 00000000 c0393e60 00000000 (...)
 Call Trace:
  [<c0103273>] show_stack+0x7a/0x82
  [<c0103381>] show_registers+0xee/0x157
  [<c0103533>] die+0xd1/0x157
  [<c01102eb>] do_page_fault+0x385/0x4ae
  [<c0102f57>] error_code+0x4f/0x54
  [<c010447b>] show_interrupts+0x21/0x156
  [<c01687be>] seq_read+0xdd/0x24c
  [<c014b9ee>] vfs_read+0x88/0x128
  [<c014bcbc>] sys_read+0x3a/0x61
  [<c0102d2d>] syscall_call+0x7/0xb
 Code: eb 1c 88 1a 42 ff 45 0c 8b 45 0c 0f b6 18 84 db 74 05 3b 55 f0 72 91 2b
 17 31 c0 89 57 0c 5b 5b 5e 5f 5d c3 55 89 e5 53 8b 5d 08 <8b> 4b 0c 8b 53 (...)
```

**EIP = c0168c7b**

**thanks to the EIP register we obtain two important informations:**
**code segment and instruction address**

# When system crashes: understanding the Oops output (4)

```
Unable to handle kernel NULL pointer dereference at virtual address 0000000c
 EIP  = c0168c7b
 *pde = 00000000
 Oops: 0000 [#1]
 PREEMPT
 Modules linked in: pcmcia firmware_class pcmcia_core e ...)
 CPU:    0
 EIP:    0060:[<c0168c7b>]  Not tainted VLI
 EFLAGS: 00010246   (2.6.15khc06)
 EIP is at seq_printf+0x7/0x43
 eax: ca809f4c   ebx: 00000000   ecx: 00000000   edx: ca809f4c
 esi: 00000000   edi: ca809f4c   ebp: ca809f0c   esp: ca809f08
 ds: 007bes: 007b   ss: 0068
 Process cat (pid: 5986, threadinfo=ca808000 task=cfa46a50)
 Stack: 00000000 ca809f2c c010447b 00000000 c0393e60 00000000 (...)
 Call Trace:
  [<c0103273>] show_stack+0x7a/0x82
  [<c0103381>] show_registers+0xee/0x157
  [<c0103533>] die+0xd1/0x157
  [<c01102eb>] do_page_fault+0x385/0x4ae
  [<c0102f57>] error_code+0x4f/0x54
  [<c010447b>] show_interrupts+0x21/0x156
  [<c01687be>] seq_read+0xdd/0x24c
  [<c014b9ee>] vfs_read+0x88/0x128
  [<c014bcbc>] sys_read+0x3a/0x61
  [<c0102d2d>] syscall_call+0x7/0xb
 Code: eb 1c 88 1a 42 ff 45 0c 8b 45 0c 0f b6 18 84 db 74 05 3b 55 f0 72 91 2b
 17 31 c0 89 57 0c 5b 5b 5e 5f 5d c3 55 89 e5 53 8b 5d 08 <8b> 4b 0c 8b 53 (...)
```

**Oops: 0000 [#1]**

**Oops counter: there can be many Oops messages. Trust only the first one, it is more reliable**

## When system crashes: understanding the Oops output (5)

```
Unable to handle kernel NULL pointer dereference at virtual address 0000000c
 EIP  = c0168c7b
 *pde = 00000000
 Oops:
 PREE
 Modu
 CPU:
 EIP:
 EFLA
 EIP
 eax:
 esi:
 ds:
 Proc
 Stack:
 Call Trace:
  [<c0103273>] show_stack+0x7a/0x82
  [<c0103381>] show_registers+0xee/0x157
  [<c0103533>] die+0xd1/0x157
  [<c01102eb>] do_page_fault+0x385/0x4ae
  [<c0102f57>] error_code+0x4f/0x54
  [<c010447b>] show_interrupts+0x21/0x156
  [<c01687be>] seq_read+0xdd/0x24c
  [<c014b9ee>] vfs_read+0x88/0x128
  [<c014bcbc>] sys_read+0x3a/0x61
  [<c0102d2d>] syscall_call+0x7/0xb
 Code: eb 1c 88 1a 42 ff 45 0c 8b 45 0c 0f b6 18 84 db 74 05 3b 55 f0 72 91 2b
 17 31 c0 89 57 0c 5b 5b 5e 5f 5d c3 55 89 e5 53 8b 5d 08 <8b> 4b 0c 8b 53 (...)
```

```
CPU:   0
EIP:   0060:[<c0168c7b>]   Not tainted VLI
EFLAGS: 00010246   (2.6.15khc06)
EIP is at seq_printf+0x7/0x43
eax: ca809f4c   ebx: 00000000 ecx: 00000000   edx: ca809f4c
esi: 00000000   edi: ca809f4c ebp: ca809f0c   esp: ca809f08
ds: 007b  es: 007b   ss: 0068
```

cpu_id, program status, general purpose registers, control registers

## When system crashes: understanding the Oops output (6)

```
Unable to handle kernel NULL pointer dereference at virtual address 0000000c
 EIP  = c0168c7b
 *pde = 00000000
 Oops
 PRE
 Mod
 CPU
 EIP
 EFL
 EIP
 eax
 esi
 ds:
 Pro
 Sta
 Cal
  [<
  [<
  [<
  [<c01102eb>] do_page_fault+0x385/0x4ae
  [<c0102f57>] error_code+0x4f/0x54
  [<c010447b>] show_interrupts+0x21/0x156
  [<c01687be>] seq_read+0xdd/0x24c
  [<c014b9ee>] vfs_read+0x88/0x128
  [<c014bcbc>] sys_read+0x3a/0x61
  [<c0102d2d>] syscall_call+0x7/0xb
 Code: eb 1c 88 1a 42 ff 45 0c 8b 45 0c 0f b6 18 84 db 74 05 3b 55 f0 72 91 2b
 17 31 c0 89 57 0c 5b 5b 5e 5f 5d c3 55 89 e5 53 8b 5d 08 <8b> 4b 0c 8b 53 (...)
```

```
Stack: 00000000 ca809f2c c010447b 00000000 c0393e60 00000000
(...)
 Call Trace:
  [<c0103273>] show_stack+0x7a/0x82
  [<c0103381>] show_registers+0xee/0x157
  [<c0103533>] die+0xd1/0x157
  [<c01102eb>] do_page_fault+0x385/0x4ae
  [<c0102f57>] error_code+0x4f/0x54
  [<c010447b>] show_interrupts+0x21/0x156
  [<c01687be>] seq_read+0xdd/0x24c
```

**process stack and return addresses**

## When system crashes: where is the bug? (1)

1) find out the function where the bug occurred by searching the EIP into the System.map of the running kernel or using the same Oops message (if using 2.6 kernel, the last one is faster):

   **seq_printf+0x7**

2) find out the last suitable function invoked before the crash (searching into the System.map or the Oops message):

   **show_interrupts+0x21**

3) launch the GNU debugger (gdb) on the linux kernel you are examining and disassemble the function found at step 2:

```
# gdb /usr/src/linux-`uname -r`/vmlinux
(gdb) disassemble show_interrupts
```

## When system crashes: where is the bug? (2)

4) Go to the offset found during step 2 (0x21 = 33) :

```
(gdb) disassemble show_interrupts
Dump of assembler code for function show_interrupts:
0xc010445a <show_interrupts+0>: push %ebp
0xc010445b <show_interrupts+1>: mov  %esp,%ebp
0xc010445d <show_interrupts+3>: push %edi
(...)
0xc0104474 <show_interrupts+26>: push $0x0
0xc0104476 <show_interrupts+28>: call 0xc0168c74 <seq_printf>
0xc010447b <show_interrupts+33>: pop  %eax
0xc010447c <show_interrupts+34>: pop  %edx
0xc010447d <show_interrupts+35>: push $0x0
(...)
```

## When system crashes: where is the bug? (3)

5) ok, now let's give a look to the disassembled code of `seq_printf()`:

```
(gdb) disassemble seq_printf
Dump of assembler code for function seq_printf:
0xc0168c74 <seq_printf+0>:  push   %ebp
0xc0168c75 <seq_printf+1>:  mov    %esp,%ebp
0xc0168c77 <seq_printf+3>:  push   %ebx
0xc0168c78 <seq_printf+4>:  mov    0x8(%ebp),%ebx
0xc0168c7b <seq_printf+7>:  mov    0xc(%ebx),%ecx
0xc0168c7e <seq_printf+10>: mov    0x4(%ebx),%edx
0xc0168c81 <seq_printf+13>: cmp    %edx,%ecx
(...)
```

## When system crashes: where is the bug? (4)

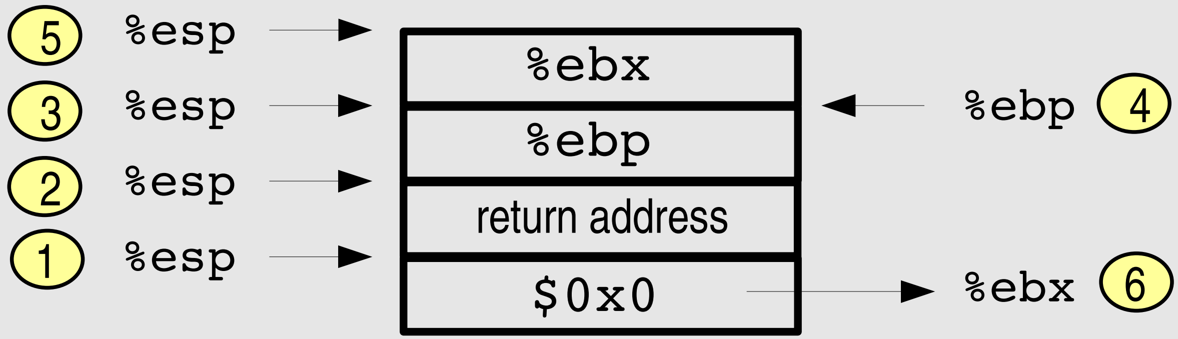
5) that is what happened on the stack:

*show_interrupts()*

```
(1) push $0x0
(2) call 0xc0168c74
```

*seq_printf()*

```
(3) push %ebp
(4) mov  %esp,%ebp
(5) push %ebx
(6) mov  0x8(%ebp),%ebx
(7) mov  0xc(%ebx),%ecx
```




**Unable to handle kernel NULL pointer dereference at virtual address 0000000c**

## When system crashes: where is the bug? (5)

6) It is time to enter the show_interrupts() souce, as we understood the problem must rely on the first parameter passed to seq_printf():

```
int show_interrupts(struct seq_file *p, void *v)
{
   int i = *(loff_t *) v, j;
   struct irqaction * action;
   unsigned long flags;

   if (i == 0) {
      p=0;
      seq_printf(p, "           ");
      for_each_online_cpu(j)
(...)
```

**ok, we found the bug! ;-)**

# Useful debugging tools: netconsole and netcat

## Netconsole (1)

Linux kernel 2.6 support a useful tool used to send console messages from the kernel you are debugging to your host through a simple TCP/IP connection (UDP protocol).

To use Netconsole, simply do the followings:

1) compile the feature in your kernel as module or built in (better)
2) if netconsole is built-in, launch your kernel image at boot in this way:

```
netconsole=4444@10.0.0.1/eth0,5555@10.0.0.2/12:34:56:A1:B2:C3
```

else:

```
insmod netconsole (on the same line)
netconsole=4444@10.0.0.1/eth0,5555@10.0.0.2/12:34:56:A1:B2:C3
```

## Useful debugging tools: netconsole and netcat

### Netconsole (2)

```
netconsole=4444@10.0.0.1/eth0,5555@10.0.0.2/12:34:56:A1:B2:C3
```

source port

source ip address

source network device

destination port

destination ip address

destination mac address

# Useful debugging tools: netconsole and netcat

## Netcat (1)

Netconsole cannot work properly if you are not listening to the port specified in the "destination port" field.
In order to do that, we can use the netcat utility as follows:

```
nc -u -l -p 5555
```

port number netcat will try to open

the transport protocol will be UDP

tells netcat to enter the listening mode