Bug hunting

Kernel bug reports often come with a stack dump like the one below:

```
------------[ cut here ]------------
WARNING: CPU: 1 PID: 28102 at kernel/module.c:1108 module_put+0x57/0x70
Modules linked in: dvb_usb_gp8psk(-) dvb_usb dvb_core nvidia_drm(PO) nvidia_modeset(PO) snd_hda_codec_hdmi snd_hda_intel snd_hda_codec s
CPU: 1 PID: 28102 Comm: rmmod Tainted: P        WC O 4.8.4-build.1 #1
Hardware name: MSI MS-7309/MS-7309, BIOS V1.12 02/23/2009
 00000000 c12ba080 00000000 00000000 c103ed6a c1616014 00000001 00006dc6
 c1615862 00000454 c109e8a7 c109e8a7 00000009 ffffffff 00000000 f13f6a10
 f5f5a600 c103ee33 00000009 00000000 00000000 c109e8a7 f80ca4d0 c109f617
Call Trace:
 [<c12ba080>] ? dump_stack+0x44/0x64
 [<c103ed6a>] ? __warn+0xfa/0x120
 [<c109e8a7>] ? module_put+0x57/0x70
 [<c109e8a7>] ? module_put+0x57/0x70
 [<c103ee33>] ? warn_slowpath_null+0x23/0x30
 [<c109e8a7>] ? module_put+0x57/0x70
 [<f80ca4d0>] ? gp8psk_fe_set_frontend+0x460/0x460 [dvb_usb_gp8psk]
 [<c109f617>] ? symbol_put_addr+0x27/0x50
 [<f80bc9ca>] ? dvb_usb_adapter_frontend_exit+0x3a/0x70 [dvb_usb]
 [<f80bb3bf>] ? dvb_usb_exit+0x2f/0xd0 [dvb_usb]
 [<c13d03bc>] ? usb_disable_endpoint+0x7c/0xb0
 [<f80bb48a>] ? dvb_usb_device_exit+0x2a/0x50 [dvb_usb]
 [<c13d2882>] ? usb_unbind_interface+0x62/0x250
 [<c136b514>] ? __pm_runtime_idle+0x44/0x70
 [<c13620d8>] ? __device_release_driver+0x78/0x120
 [<c1362907>] ? driver_detach+0x87/0x90
 [<c1361c48>] ? bus_remove_driver+0x38/0x90
 [<c13d1c18>] ? usb_deregister+0x58/0xb0
 [<c109fbb0>] ? SyS_delete_module+0x130/0x1f0
 [<c1055654>] ? task_work_run+0x64/0x80
 [<c1000fa5>] ? exit_to_usermode_loop+0x85/0x90
 [<c10013f0>] ? do_fast_syscall_32+0x80/0x130
 [<c1549f43>] ? sysenter_past_esp+0x40/0x6a
---[ end trace 6ebc60ef3981792f ]---
```

Such stack traces provide enough information to identify the line inside the Kernel's source code where the bug happened. Depending on the severity of the issue, it may also contain the word **Oops**, as on this one:

```
BUG: unable to handle kernel NULL pointer dereference at   (null)
IP: [<c06969d4>] iret_exc+0x7d0/0xa59
*pdpt = 000000002258a001 *pde = 0000000000000000
Oops: 0002 [#1] PREEMPT SMP
...
```

Despite being an **Oops** or some other sort of stack trace, the offended line is usually required to identify and handle the bug. Along this chapter, we'll refer to "Oops" for all kinds of stack traces that need to be analyzed.

If the kernel is compiled with `CONFIG_DEBUG_INFO`, you can enhance the quality of the stack trace by using file:*scripts/decode_stacktrace.sh*.

Modules linked in

Modules that are tainted or are being loaded or unloaded are marked with "(...)", where the taint flags are described in file:*Documentation/admin-guide/tainted-kernels.rst*, "being loaded" is annotated with "+", and "being unloaded" is annotated with "-".

Where is the Oops message is located?

Normally the Oops text is read from the kernel buffers by klogd and handed to `syslogd` which writes it to a syslog file, typically `/var/log/messages` (depends on `/etc/syslog.conf`). On systems with systemd, it may also be stored by the `journald` daemon, and accessed by running `journalctl` command.

Sometimes `klogd` dies, in which case you can run `dmesg > file` to read the data from the kernel buffers and save it. Or you can `cat /proc/kmsg > file`, however you have to break in to stop the transfer, since `kmsg` is a "never ending file".

If the machine has crashed so badly that you cannot enter commands or the disk is not available then you have three options:

1. Hand copy the text from the screen and type it in after the machine has restarted. Messy but it is the only option if you have not planned for a crash. Alternatively, you can take a picture of the screen with a digital camera - not nice, but better than nothing. If the messages scroll off the top of the console, you may find that booting with a higher resolution (e.g., `vga=791`) will allow you to read more of the text. (Caveat: This needs `vesafb`, so won't help for 'early' oopses.)
2. Boot with a serial console (see :ref:`Documentation/admin-guide/serial-console.rst <serial_console>`), run a null modem to a second machine and capture the output there using your favourite communication program. Minicom works well.

   System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\(linux-master) (Documentation) (admin-guide)bug-hunting.rst`, line 92); *backlink*

   Unknown interpreted text role "ref".

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

Finding the bug's location

Reporting a bug works best if you point the location of the bug at the Kernel source file. There are two methods for doing that. Usually, using `gdb` is easier, but the Kernel should be pre-compiled with debug info.

gdb

The GNU debugger (`gdb`) is the best way to figure out the exact file and line number of the OOPS from the `vmlinux` file.

The usage of gdb works best on a kernel compiled with `CONFIG_DEBUG_INFO`. This can be set by running:

```
$ ./scripts/config -d COMPILE_TEST -e DEBUG_KERNEL -e DEBUG_INFO
```

On a kernel compiled with `CONFIG_DEBUG_INFO`, you can simply copy the EIP value from the OOPS:

```
EIP:    0060:[<c021e50e>]    Not tainted VLI
```

And use GDB to translate that to human-readable form:

```
$ gdb vmlinux
(gdb) l *0xc021e50e
```

If you don't have `CONFIG_DEBUG_INFO` enabled, you use the function offset from the OOPS:

```
EIP is at vt_ioctl+0xda8/0x1482
```

And recompile the kernel with `CONFIG_DEBUG_INFO` enabled:

```
$ ./scripts/config -d COMPILE_TEST -e DEBUG_KERNEL -e DEBUG_INFO
$ make vmlinux
$ gdb vmlinux
(gdb) l *vt_ioctl+0xda8
0x1888 is in vt_ioctl (drivers/tty/vt/vt_ioctl.c:293).
```

```
288   {
289           struct vc_data *vc = NULL;
290           int ret = 0;
291
292           console_lock();
293           if (VT_BUSY(vc_num))
294                   ret = -EBUSY;
295           else if (vc_num)
296                   vc = vc_deallocate(vc_num);
297           console_unlock();
```

or, if you want to be more verbose:

```
(gdb) p vt_ioctl
$1 = {int (struct tty_struct *, unsigned int, unsigned long)} 0xae0 <vt_ioctl>
(gdb) l *0xae0+0xda8
```

You could, instead, use the object file:

```
$ make drivers/tty/
$ gdb drivers/tty/vt/vt_ioctl.o
(gdb) l *vt_ioctl+0xda8
```

If you have a call trace, such as:

```
Call Trace:
 [<ffffffff8802c8e9>] :jbd:log_wait_commit+0xa3/0xf5
 [<ffffffff810482d9>] autoremove_wake_function+0x0/0x2e
 [<ffffffff8802770b>] :jbd:journal_stop+0x1be/0x1ee
 ...
```

this shows the problem likely is in the :jbd: module. You can load that module in gdb and list the relevant code:

```
$ gdb fs/jbd/jbd.ko
(gdb) l *log_wait_commit+0xa3
```

> **Note**
>
> You can also do the same for any function call at the stack trace, like this one:
>
> ```
> [<f80bc9ca>] ? dvb_usb_adapter_frontend_exit+0x3a/0x70 [dvb_usb]
> ```
>
> The position where the above call happened can be seen with:
>
> ```
> $ gdb drivers/media/usb/dvb-usb/dvb-usb.o
> (gdb) l *dvb_usb_adapter_frontend_exit+0x3a
> ```

objdump

To debug a kernel, use objdump and look for the hex offset from the crash output to find the valid line of code/assembler. Without debug symbols, you will see the assembler code for the routine shown, but if your kernel has debug symbols the C code will also be available. (Debug symbols can be enabled in the kernel hacking menu of the menu configuration.) For example:

```
$ objdump -r -S -l --disassemble net/dccp/ipv4.o
```

> **Note**
>
> You need to be at the top level of the kernel tree for this to pick up your C files.

If you don't have access to the source code you can still debug some crash dumps using the following method (example crash dump output as shown by Dave Miller):

```
EIP is at  +0x14/0x4c0
 ...
Code: 44 24 04 e8 6f 05 00 00 e9 e8 fe ff ff 8d 76 00 8d bc 27 00 00
00 00 55 57  56 53 81 ec bc 00 00 00 8b ac 24 d0 00 00 00 8b 5d 08
<8b> 83 3c 01 00 00 89 44  24 14 8b 45 28 85 c0 89 44 24 18 0f 85

Put the bytes into a "foo.s" file like this:

       .text
       .globl foo
foo:
       .byte  .... /* bytes from Code: part of OOPS dump */

Compile it with "gcc -c -o foo.o foo.s" then look at the output of
"objdump --disassemble foo.o".

Output:

ip_queue_xmit:
    push       %ebp
    push       %edi
    push       %esi
    push       %ebx
    sub        $0xbc, %esp
    mov        0xd0(%esp), %ebp        ! %ebp = arg0 (skb)
    mov        0x8(%ebp), %ebx         ! %ebx = skb->sk
    mov        0x13c(%ebx), %eax       ! %eax = inet_sk(sk)->opt
```

file:*scripts/decodecode* can be used to automate most of this, depending on what CPU architecture is being debugged.

Reporting the bug

Once you find where the bug happened, by inspecting its location, you could either try to fix it yourself or report it upstream.

In order to report it upstream, you should identify the mailing list used for the development of the affected code. This can be done by using the `get_maintainer.pl` script.

For example, if you find a bug at the gspca's sonixj.c file, you can get its maintainers with:

```
$ ./scripts/get_maintainer.pl -f drivers/media/usb/gspca/sonixj.c
Hans Verkuil <hverkuil@xs4all.nl> (odd fixer:GSPCA USB WEBCAM DRIVER,commit_signer:1/1=100%)
Mauro Carvalho Chehab <mchehab@kernel.org> (maintainer:MEDIA INPUT INFRASTRUCTURE (V4L/DVB),commit_signer:1/1=100%)
Tejun Heo <tj@kernel.org> (commit_signer:1/1=100%)
Bhaktipriya Shridhar <bhaktipriya96@gmail.com> (commit_signer:1/1=100%,authored:1/1=100%,added_lines:4/4=100%,removed_lines:9/9=100%)
linux-media@vger.kernel.org (open list:GSPCA USB WEBCAM DRIVER)
linux-kernel@vger.kernel.org (open list)
```

Please notice that it will point to:

- The last developers that touched the source code (if this is done inside a git tree). On the above example, Tejun and Bhaktipriya (in this specific case, none really involved on the development of this file);
- The driver maintainer (Hans Verkuil);
- The subsystem maintainer (Mauro Carvalho Chehab);
- The driver and/or subsystem mailing list (linux-media@vger.kernel.org);
- the Linux Kernel mailing list (linux-kernel@vger.kernel.org).

Usually, the fastest way to have your bug fixed is to report it to mailing list used for the development of the code (linux-media ML) copying the driver maintainer (Hans).

If you are totally stumped as to whom to send the report, and `get_maintainer.pl` didn't provide you anything useful, send it to linux-kernel@vger.kernel.org.

Thanks for your help in making Linux as stable as humanly possible.

Fixing the bug

If you know programming, you could help us by not only reporting the bug, but also providing us with a solution. After all, open source is about sharing what you do and don't you want to be recognised for your genius?

If you decide to take this way, once you have worked out a fix please submit it upstream.

Please do read :ref:`Documentation/process/submitting-patches.rst <submittingpatches>` though to help your code get accepted.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\(linux-master) (Documentation) (admin-guide)bug-hunting.rst`, line 292); *backlink*
>
> Unknown interpreted text role "ref".

Notes on Oops tracing with `klogd`

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

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

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

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

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

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

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

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

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

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

```
Dr. G.W. Wettstein           Oncology Research Div. Computing Facility
Roger Maris Cancer Center    INTERNET: greg@wind.rmcc.com
820 4th St. N.
Fargo, ND  58122
Phone: 701-234-7556
```