Debugging Kernel Panic

Kernel crash (OOPs) messages interpretation

- The following section briefly describes kernel OOPs message interpretation and identifying the corresponding lines of code executed before kernel crash
- Kernel crash can occur due to Hardware or Software fault
- The following section describes kernel crash investigation due to Software failure

Case Study

- Intentionally generated kernel crash by modifying bridge driver code as shown below

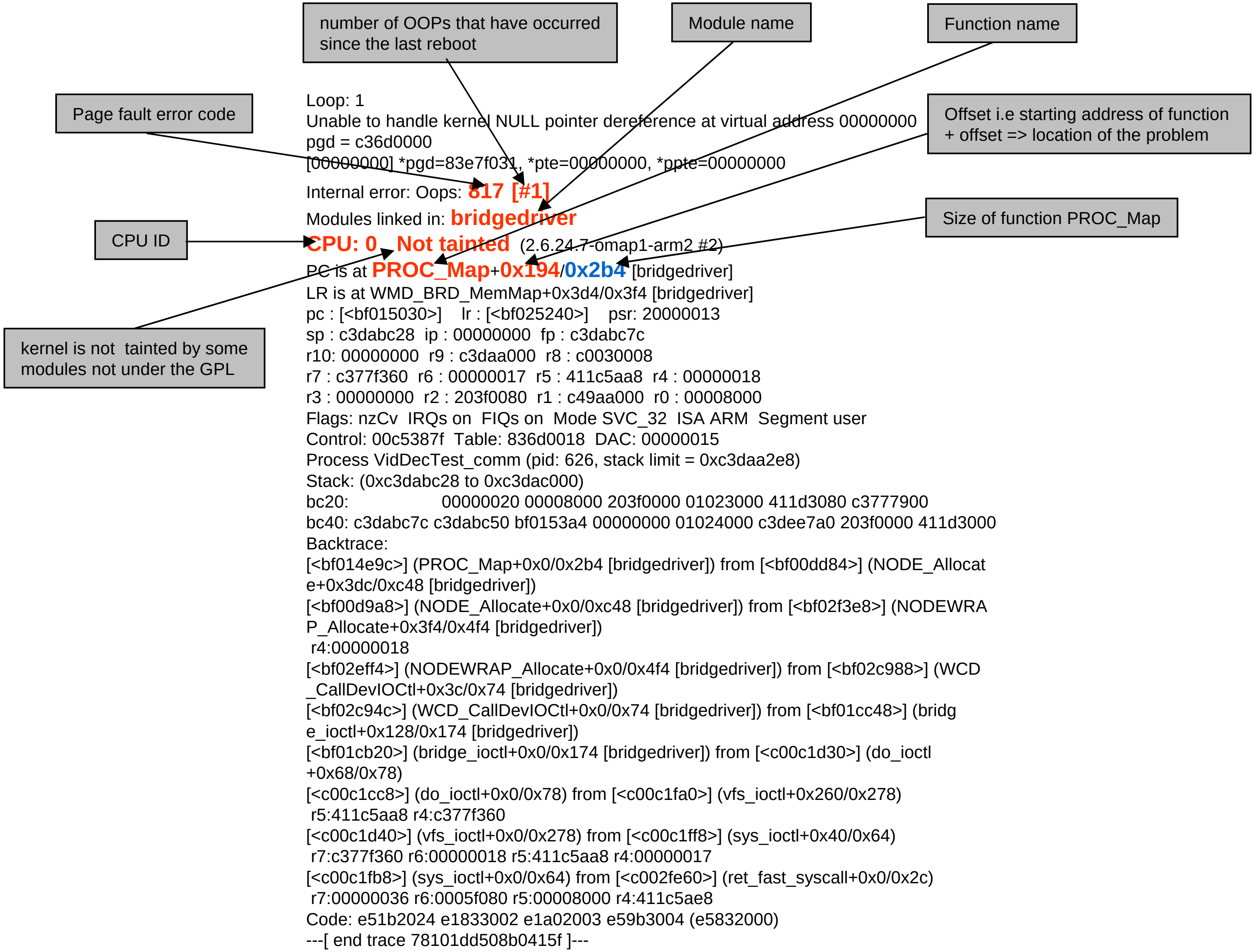
```
if (DSP_SUCCEEDED(status)) {
                /* Mapped address = MSB of VA | LSB of PA */
                ppMapAddr = NULL;
                *ppMapAddr = (void *) (vaAlign | ((u32) pMpuAddr &
                                                (PG_SIZE_4K - 1)));
```

- Rebuild the bridge driver and execute test case like video decoding
- This will cause Kernel crash
- Collect the Kernel OOPs message and lets analyze

Steps involved in analyzing and debugging kernel crash

1. Identify the module from Kernel OOPs message that caused kernel crash
2. Trace function calls leading to kernel crash
3. Identify offset address of the last executed function prior to kernel crash
4. Generate the object dump of the corresponding file/module
5. Identify the line that was executed before kernel crash
6. Identify the root cause of kernel crash

Typical kernel crash(OOPs) message



number of OOPs that have occurred since the last reboot

Module name

Function name

Page fault error code

Offset i.e starting address of function + offset => location of the problem

CPU ID

Size of function PROC_Map

kernel is not tainted by some modules not under the GPL

```
Loop: 1
Unable to handle kernel NULL pointer dereference at virtual address 00000000
pgd = c36d0000
[00000000] *pgd=83e7f031, *pte=00000000, *ppte=00000000
Internal error: Oops: 817 [#1]
Modules linked in: bridgedriver
CPU: 0    Not tainted  (2.6.24.7-omap1-arm2 #2)
PC is at PROC_Map+0x194/0x2b4 [bridgedriver]
LR is at WMD_BRD_MemMap+0x3d4/0x3f4 [bridgedriver]
pc : [<bf015030>]    lr : [<bf025240>]    psr: 20000013
sp : c3dabc28  ip : 00000000  fp : c3dabc7c
r10: 00000000  r9 : c3daa000  r8 : c0030008
r7 : c377f360  r6 : 00000017  r5 : 411c5aa8  r4 : 00000018
r3 : 00000000  r2 : 203f0080  r1 : c49aa000  r0 : 00008000
Flags: nzCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment user
Control: 00c5387f  Table: 836d0018  DAC: 00000015
Process VidDecTest_comm (pid: 626, stack limit = 0xc3daa2e8)
Stack: (0xc3dabc28 to 0xc3dac000)
bc20:                   00000020 00008000 203f0000 01023000 411d3080 c3777900
bc40: c3dabc7c c3dabc50 bf0153a4 00000000 01024000 c3dee7a0 203f0000 411d3000
Backtrace:
[<bf014e9c>] (PROC_Map+0x0/0x2b4 [bridgedriver]) from [<bf00dd84>] (NODE_Allocat
e+0x3dc/0xc48 [bridgedriver])
[<bf00d9a8>] (NODE_Allocate+0x0/0xc48 [bridgedriver]) from [<bf02f3e8>] (NODEWRA
P_Allocate+0x3f4/0x4f4 [bridgedriver])
 r4:00000018
[<bf02eff4>] (NODEWRAP_Allocate+0x0/0x4f4 [bridgedriver]) from [<bf02c988>] (WCD
_CallDevIOCtl+0x3c/0x74 [bridgedriver])
[<bf02c94c>] (WCD_CallDevIOCtl+0x0/0x74 [bridgedriver]) from [<bf01cc48>] (bridg
e_ioctl+0x128/0x174 [bridgedriver])
[<bf01cb20>] (bridge_ioctl+0x0/0x174 [bridgedriver]) from [<c00c1d30>] (do_ioctl
+0x68/0x78)
[<c00c1cc8>] (do_ioctl+0x0/0x78) from [<c00c1fa0>] (vfs_ioctl+0x260/0x278)
 r5:411c5aa8 r4:c377f360
[<c00c1d40>] (vfs_ioctl+0x0/0x278) from [<c00c1ff8>] (sys_ioctl+0x40/0x64)
 r7:c377f360 r6:00000018 r5:411c5aa8 r4:00000017
[<c00c1fb8>] (sys_ioctl+0x0/0x64) from [<c002fe60>] (ret_fast_syscall+0x0/0x2c)
 r7:00000036 r6:0005f080 r5:00008000 r4:411c5ae8
Code: e51b2024 e1833002 e1a02003 e59b3004 (e5832000)
---[ end trace 78101dd508b0415f ]---
```

Step 1. Identify the module

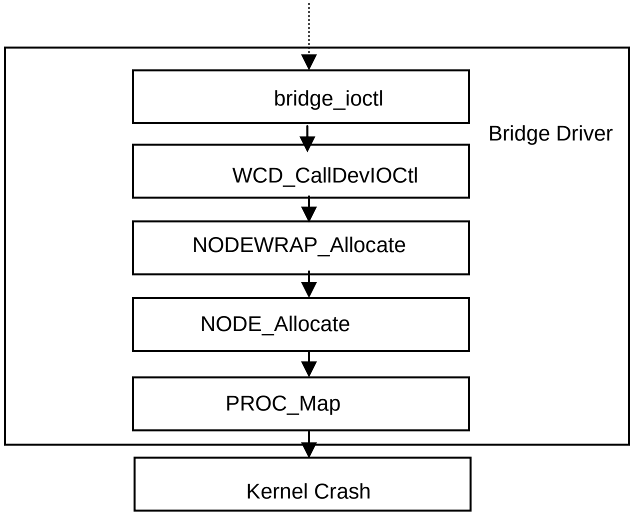
- The module name is listed in back trace and PC section of Kernel OOP’s message as shown below
  - **Backtrace:**

    *[<bf014e9c>] (PROC_Map+0x0/0x2b4 [bridgedriver]) from [<bf00dd84>] (NODE_Allocate+0x3dc/0xc48 [***bridgedriver***])*
  - **PC:**

    *PC is at PROC_Map+0x194/0x2b4 [***bridgedriver***]*

Step 2. Trace function calls leading to kernel crash

- The Backtrace traces last few function calls leading to kernel crash as shown below



- Backtrace and PC also points to last function call executed prior to crash
  - **Backtrace:**
    *[<bf014e9c>] (**PROC_Map**+0x0/0x2b4 [bridgedriver]) from [<bf00dd84>] (NODE_Allocate+0x3dc/0xc48 [bridgedriver])*
  - **PC:**
    *PC is at **PROC_Map**+0x194/0x2b4 [bridgedriver]*

Step 3. Identify offset address of the last executed function prior to kernel crash

- The function offset is specified next to function name as shown below
  - *PC is at PROC_Map+**0x194**/0x2b4 [bridgedriver]*

Step 4. Generate the object dump of the corresponding file/module

- Rebuild the corresponding module/file with –g option

- Generate the object dump of the corresponding module/file something like this
  - *arm-none-linux-gnueabi-objdump -d -S bridgedriver.ko > bridgeObjdump*

- Open the bridge driver object dump file and search for function PROC_Map. It will look something like this

```
DSP_STATUS PROC_Map(DSP_HPROCESSOR hProcessor, void *pMpuAddr,
   u32 ulSize, void *pReqAddr, void **ppMapAddr, u32 ulMapAttr)
{
                14e9c:      e1a0c00d       mov     ip, sp
                14ea0:      e92dd800       push    {fp, ip, lr, pc}
                14ea4:      e24cb004       sub     fp, ip, #4       ; 0x4
```

- Note starting address of the function i.e. **0x14e9c** in this case

Step 5. Identify the source code line that was executed before kernel crash

- Add the offset value as mentioned in step 3 to starting address of the function as mentioned in step 4 as shown below

  0x14e96+0x194=1502a

- Go to offset 1502a in bridge driver object dump file. The line before this address is the line executed before the crash as shown below

```
/* Mapped address = MSB of VA | LSB of PA */
        ppMapAddr = NULL;
 1500c:     e3a03000      mov     r3, #0  ; 0x0
 15010:     e58b3004      str     r3, [fp, #4]
        *ppMapAddr = (void *) (vaAlign | ((u32) pMpuAddr &
 15014:     e51b3044      ldr     r3, [fp, #-68]
 15018:     e1a03a03      lsl     r3, r3, #20
 1501c:     e1a03a23      lsr     r3, r3, #20
 15020:     e51b2024      ldr     r2, [fp, #-36]
 15024:     e1833002      orr     r3, r3, r2
 15028:     e1a02003      mov     r2, r3
 1502c:     e59b3004      ldr     r3, [fp, #4]
 15030:     e5832000      str     r2, [r3]
```

Kernel crashed executing this line

Step 6. Identify the root cause of kernel crash

- If the last line is variable assignment, then check the address and size of the associated variable
- If the last line is a function call from our module to third party module, then check the argument variables that is being passed

Debug environment

- Make sure that the environment is same, else the address will not match.
- If the environment is not the same, then you can guesstimate by looking at the offset value and inspecting the code around that area

Conclusion

- The previous slides just explains as how to get to the line of code which were executed prior to Kernel panic due to software bug
- Sometime, it is not possible to get to the root cause with previous analysis and may need further investigation