H168N挂死故障查证小结

软件九室 陈强 20121019

近期H168N频现单板挂死重启的bug，影响极其严重，给开发、测试及现场人员都带来了巨大的压力。很多问题武研所的同事都需要花费大量的时间和精力去查证，还常常需要求助南京部门同事。现在结合H168N最新出现的一个挂死bug查证的过程，与大家交流一下查证的经验和思路（在此感谢四室彭永超的大力支持帮助）。

故障描述：613001615887

【H168N V1.1.0_HUT2】综合拷机里只开无线下载，单板挂死了，串口下出现很多打印 ZXHN H168N V1.1　ZXHN H168N V1.1.0_HUT2　缺陷/故障

【配置信息】

开启一个ssid，下挂bt下载。bt下载有2个网卡，任务数为40个左右。

查证流程及思路：

1、常规武器——EPC

目前大多数单板挂死，都会产生epc信息，根据epc信息定位故障，是采取的最常规最优先的手段。以这次故障为例，请注意反汇编和定位epc语句：

CPU 0 Unable to handle kernel paging request at virtual address 000000dc, epc == c0640ff8, ra == c0640fe8

--------kfree skb diagnose utility--------

PC[kfree_skb]　　Hits

Oops[#1]:

Cpu 0

$ 0　: 00000000 10008400 829a2280 00000002

$ 4　: 828e0000 0000002e 00000008 83ab2a80

$ 8　: 00000004 0000000c 00000022 00000005

$12　: 0000000d c081d878 00000000 00000001

$16　: 828e0000 c05e6df4 00000002 00000000

$20　: 00000000 00000000 00000002 000000e3

$24　: 00000000 c05e674c

$28　: 83858000 8385bc48 0000005e c0640fe8

Hi　: 00000000

Lo　: 000026de

epc　: c0640ff8 wlc_phy_radio205x_vcocal_nphy+0xe4/0x130 [wl]

　Tainted: P

ra　: c0640fe8 wlc_phy_radio205x_vcocal_nphy+0xd4/0x130 [wl]

...

Call Trace:

[<c0640ff8>] wlc_phy_radio205x_vcocal_nphy+0xe4/0x130 [wl]

[<c0653310>] wlc_phy_chanspec_radio2057_setup+0x370/0x1078 [wl]

[<c06620fc>] wlc_phy_chanspec_set_nphy+0x110/0x1428 [wl]

[<c0650c54>] wlc_phy_aci_scan_nphy+0xd34/0x11e4 [wl]

[<c0651154>] wlc_phy_acimode_upd_nphy+0x50/0x7a8 [wl]

[<c05f1e6c>] wlc_phy_watchdog+0xa04/0xa10 [wl]

[<c06a13a8>] wlc_watchdog_timer+0x138/0x864 [wl]

[<c06f1ce8>] _wl_timer+0xec/0x160 [wl]

[<c06f1d84>] wl_timer_task+0x28/0x6c [wl]

[<8004a60c>] worker_thread+0x1a0/0x2e4

[<8004eef0>] kthread+0x5c/0xb4

[<80014b64>] kernel_thread_helper+0x10/0x18

Code: 2405002e 00000000 00000000 <828000dc> 00000036 2405002e 24060004 0220f809 24070004

[ERROR fap4ke] fapMailBox_irqHandler,197: FAP Mailbox Error <00007>

wl/phy/wlc_phy_n.c中的wlc_phy_radio205x_vcocal_nphy()。

对wl.ko进行反汇编：

/opt/toolchains/uclibc-crosstools-gcc-4.4.2-1/usr/bin/mips-linux-uclibc-objdump -S bcmdrivers/broadcom/net/wl/impl10/build/wlobj-wlconfig_lx_wl_dslcpe_pci_ap_2n/wl.ko > wl.S(注：此处添加 –g可以得到c和汇编在一起的代码可以更加容易定位bug代码的位置)

00082f14 <wlc_phy_radio205x_vcocal_nphy>:

82f14: 27bdffe0 addiu sp,sp,-32

82f18: afb00014 sw s0,20(sp)

...

82fe8: 02002021 move a0,s0

82fec: 2405002e li a1,46

82ff0: 24060004 li a2,4

82ff4: 0220f809 jalr s1

82ff8: 00003821 move a3,zero // epc指向这里

82ffc: 02002021 move a0,s0

83000: 2405002e li a1,46

83004: 24060004 li a2,4

83008: 0220f809 jalr s1

通过epc信息，可以定位故障出现在wlc_phy_radio205x_vcocal_nphy函数，通过与C代码对比，有的简单故障可以直接定位到语句并解决，比如数组越界等。但是对于大部分故障，直接对应epc的代码本身并没有错（本例对应的语句为mod_radio_reg，一个单纯的写寄存器的动作，并且寄存器地址和数值都是固定值，代码本身没有问题）。此时就需要更为详细的分析。

将出现异常时打印的Code（其中< >内是epc指向的指令），与反汇编的结果进行比较，很显然代码确实是被破坏掉了，而且有4条指令被破坏！

82ff0: 24060004 li a2,4

82ff4: 0220f809 jalr s1

82ff8: 00003821 move a3,zero // epc指向这里

82ffc: 02002021 move a0,s0

2、查证关键——复现故障

查证故障的关键是稳定快速的复现故障。为了快速稳定的复现故障，首先保持故障出现的环境，随后通过某些手段尽快复现故障。

针对无线+IPTV拷机挂死故障，可以添加多台笔记本进行迅雷下载，同时增大IPTV拷机流量。

为了尽快复现故障，同时考虑是否是由于内存不足导致，添加测试函数，不断申请和释放内存。

2.1 添加脚本反复申请释放内存

通过不断申请，30s后再释放内存，验证故障是否在低内存资源下更容易复现，具体方法如下：

将eatmem.7z放到chip_bcmbcm_412userspaceprivateapps目录下：

修改chip_bcmbcm_412userspaceprivateapps\Makefile：

单独编译用户程序：

make PROFILE=H168NV11_HU userspace

2.2 使用memwatch工具

Memwatch是用来查某一个进程的内存泄露情况的。

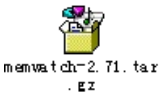
#define malloc(n)　　mwMalloc(n,__FILE__,__LINE__)

Memwatch动态库包括一个c文件memwatch.c和一个头文件memwatch.h。当我们怀疑某个c文件中有内存泄露时，就在该文件中包含头文件memwatch.h，这样，这个文件中引用标准c库中的malloc函数就会被替换成memwatch.c中的mwMalloc函数。

mwMalloc函数的作用就在于，它在调用c库的标准malloc的时候，还会做一些动作，维护一些全局变量，把原本调用malloc的地方在哪个文件的哪一行，申请了多少，分配内存的起始地址记录下来。

同样的道理，mwFree也会记录，如果free的指针在上面的mwMalloc记录的全局变量里找到，则认为相应的分配释放了。但是如果free的指针在mwMalloc里找不到，那么就会认为这是一个“wildfree”，给出警告。为什么会有wildfree呢？因为有的c文件调用了c标准库的一些函数，这些函数内部调用了malloc，但是内部却没有调用free，需要调用者在调用这些函数以后自己调用free释放。这样memwatch就记录不到malloc的过程，而只是记录到了free，所以就认为是“wildfree”了。

当killall 该进程时，memwatch会在var/tmp目录下输出全局变量所记录的信息，通过这些信息，我们就可以查看出堆内存的分配回收情况。



2.3 加快无线校准操作

出故障的地方，位于无线校准区域，为了尽快复现故障，因此添加一些命令，加快无线校准的流程。

```
sprintf(cmd,"wlctl -i wl%d slow_timer 10",idx);
BCMWL_WLCTL_CMD(cmd);
sprintf(cmd,"wlctl -i wl%d fast_timer 5",idx);
BCMWL_WLCTL_CMD(cmd);
sprintf(cmd,"wlctl -i wl%d glacial_timer 15",idx);
BCMWL_WLCTL_CMD(cmd);
```

3、故障分析

3.1、历史经验

很不幸，通过脚本和memwatch都没有查出问题。由于每次epc出现的地方，并不完全相同，主要出现在wlc_phy_radio205x_vcocal_nphy，故考虑更换wlan的模块位置，尝试确认是其它模块越界固定改了某一块区域。

WANKE-log-924.log

epc　: c0641ff8 wlc_phy_clip_det_nphy+0x3c/0x9c [wl]

hut_0926.log

epc　: c0644ffc wlc_phy_rfctrl_override_1tomany_nphy+0x100/0x364 [wl]

hut_0926_nocpy.log：

epc　: c0640ff8 wlc_phy_radio205x_vcocal_nphy+0xe4/0x130 [wl]

hut_0927.log

epc : c0640ff8 wlc_phy_radio205x_vcocal_nphy+0xe4/0x130 [wl]

根据前不久陆亦芬查过一个故障，日志模块出现越界错误，表现却是wlan的校准部分出现异常。因为日志存储越界的地方是确定的，紧跟在32M后面，而我们版本每次启动的顺序其实也基本比较固定，而无线模块加载的也总是在32M后面，所以每次都是破坏无线模块的校准部分。

现在这个问题的异常点也是在无线校准部分，而且每次epc指向的代码位置不固定，有时报Reserved instruction，有时报访问无效地址，看起来也很像是被破坏了。尝试下修改启动脚本，把wlan驱动模块的加载位置调整下，改变wlan模快的加载地址，看异常是否还会出现，如果出现，是否还在无线校准部分。

查看H168N代码，发现有两处加载无线模块的地方：

1. 系统启动脚本中：chip_bcm\product\H168NV11_HU\target\fs.src\etc\profile
2. 无线配置模块里也有一次无线模块加载，这个加载由于带上不合法的参数（CSP加载无线驱动模块是会带MAC地址，如果生效需求修改无线驱动代码，但在H168N匈牙利版本无线驱动是没有修改的），加载失败。生效的还是系统启动脚本中的加载。

注释掉系统启动脚本中的无线模块加载，修改无线适配模块加载无线驱动，将非法参数去掉。修改后编出版本，系统启动后对比之前lsmod 查看到wl.ko的加载地址，确实有变化。将修改后的版本在拷机环境中验证，拷机两天未出现问题，因此更加肯定了是被越界修改。

修改前lsmod

```
# lsmod

Module                  Size  Used by
option 14928 0 - Live 0xc0b26000
ftdi_sio 57968 0 - Live 0xc0b0a000
cdc_acm 28816 0 - Live 0xc0ae4000
ohci_hcd 49008 0 - Live 0xc088b000
ehci_hcd 67200 0 - Live 0xc0862000
pwrmngtd 6784 0 - Live 0xc0843000 (P)
wl 2561728 0 - Live 0xc05be000 (P)
bcmarl 7088 0 - Live 0xc02c7000 (P)
usb_storage 79792 0 - Live 0xc02a9000
bcm_enet 178880 1 wl, Live 0xc0257000
adsldd 346480 0 - Live 0xc01c4000 (P)
bcmxtmcfg 67200 1 adsldd, Live 0xc0153000 (P)
bcmfap 71280 1 bcmarl, Live 0xc010f000 (P)
pktflow 73104 1 bcmfap, Live 0xc00c5000 (P)
bcm_bpm 211888 0 [permanent], Live 0xc007c000 (P)
bcm_ingqos 13488 0 - Live 0xc006c000 (P)
```

修改后lsmod:

```
lsmod
Module                  Size  Used by
wl 2561728 0 - Live 0xc08af000 (P)
option 14928 0 - Live 0xc05b6000
ftdi_sio 57968 0 - Live 0xc059a000
cdc_acm 28816 0 - Live 0xc0574000
ohci_hcd 49008 0 - Live 0xc031b000
ehci_hcd 67200 0 - Live 0xc02f2000
pwrmngtd 6784 0 - Live 0xc02d3000 (P)
bcmarl 7088 0 - Live 0xc02c7000 (P)
usb_storage 79792 0 - Live 0xc02a9000
bcm_enet 178880 1 wl, Live 0xc0257000
adsldd 346480 0 - Live 0xc01c4000 (P)
bcmxtmcfg 67200 1 adsldd, Live 0xc0153000 (P)
bcmfap 71280 1 bcmarl, Live 0xc010f000 (P)
pktflow 73104 1 bcmfap, Live 0xc00c5000 (P)
bcm_bpm 211888 0 [permanent], Live 0xc007c000 (P)
bcm_ingqos 13488 0 - Live 0xc006c000 (P)
#
```

3.2、确认原因

通过几次log分析，最近几次检测到的被破坏的地址集中在两处，0xc0640ff0 ~ c0641000 以及 0xc0644ff0 ~ 0xc0645000处。它们的共同点是：都是一个4KB页面的最后16字节，并且其中偏移量为8的地方都被写了0x8280。同时出现问题的地方集中在wlan的phy模块。因此添加debug的思路如下，定义一个数组wlan_text_praram在wlan初始化时保存0xc063c000起始地址的内容，随后在wl_phy_cmn.c文件中添加比较函数，如果发现该地址被修改，则打印；该调试打印可添加在多个函数当中，以下为调试代码之一：

```
  unsigned char* wlan_text_vaddr1 = 0xc063c000;
if(memcmp(wlan_text_praram, wlan_text_vaddr1, 64*1024))
{
    printk("wlan code crashed in wlcphywatchdog\n");
  }
```

添加代码后，重新拷机，很快就有打印wlan code crashed in wlcphywatchdog出现，再一次确认了我们的判断是正确的，确实somebody修改了wlan的内存空间。

4、查找可疑线程

确定了问题是由写越界造成的，下一步就是确定到底是什么操作导致。因为从epc打印、wlc_phy_radio205x_vcocal_nphy和wlcphywatchdog中都看不出代码本身的问题。只能考虑是在执行某个操作在某个线程中破坏的，因此在队列调度的函数中添加上述打印。方法是在每一次队列调度前都进行比较，如果发现被破坏，则将前一次线程和PID打印出来。关键代码如下：

```
/* 如果比较从0xc063c000 ~ 0xc064c000范围的64*1024字节大小的区域，需要执行几万条指令，会导致不停地进行调度。
* 最近出现的几次异常的epc都指向c0640ff8处，可以只比较0xc0640f00 ~ 0xc0641100范围内的512字节，* 0xc0640f00 - 0xc063c000 = 0x4f00   */
if(bBeginCmp)
{
    /* 只在第一次检测到时告警 */
    if (wlan_crash_warn_count == 0)
  {
      if (cmp_code_area(wlan_text_praram, wlan_text_vaddr, 0x4f00, 512))
   {
         printk("wlan code crashed in __schedule! pid = %d, name = %s\n",                    prev->pid, prev->comm);
        dump_stack();
        wlan_crash_warn_count++;
    }
  }
}
```

最后_schedule中打印wlan code crashed in __schedule! pid = 6, name = sirq-net-rx/0，分析是在sirq-net-rx/0线程中破坏的，考虑是哪个地方的收包流程中破坏的。

如果驱动是使用NAPI机制接收，那应该是在sirq-net-rx/0线程的上下文中执行驱动的napi_poll函数。H168N上的软中断处理已经线程化了。分析了wlan驱动的代码，目前它并没有使用NAPI机制，而是用工作队列或者tasklet。怀疑是ATM和ETH驱动。

（1）以太驱动的NAPI的poll函数是：

chip_bcm\bcm_412\bcmdrivers\opensource\net\enet\impl4\bcmenet.c

bcm63xx_enet_poll_napi()函数

（2）ATM驱动napi_poll函数是：

chip_bcm\bcm_412\bcmdrivers\opensource\net\xtmrt\impl4\bcmxtmrt.c

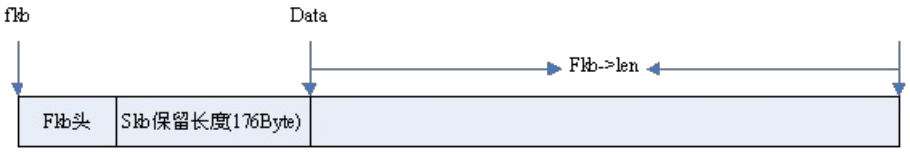
bcmxtmrt_poll_napi()函数

5、定位故障

后续拷机过程中，在__schedule()执行切换前越界打印，又抓到几次，大部分是在sirq-net-rx/0，还有一次是在wps_monitor。

从__schedule()抓到的线程是wps_monitor这点看，不大可能是某个驱动的接收处理函数破坏了wlan代码区。而且抓到的线程并不固定，那么造成破坏的代码应该不是运行在进程上下文中。由于软中断都已线程化了，那么只有可能是中断处理函数，或者是外围的硬件，包括wlan芯片、以太芯片、XTM芯片、加速引擎等所有可能对内存进行访问的。从被破坏的总是4KB页面的最后16个字节看，应该是向前越界造成的。注意内核模块占用的内存是用vmalloc分配的，虽然虚拟地址是连续的，但在物理上并不连续。有可能中间某个物理页面被用来存放报文了，当它越界时，就会破坏模块的代码或数据区。

初步怀疑写入的可能是nbuff头部，根据耿祥玉写的Braodcom nbuff加速浅析里描述，nbuff的结构是：



如果被破坏区域中写的真是nbuff头部，那么根据bcm_412/kernel/linux/include/linux/nbuff.h中定义的fkbuff{}，0x8280开头的应该是指向data的指针。我们以一次异常为例：

Code: 0240f809 00000000 00000000 <828000d0> 00000076 8fbf001c 8fb20018 8fb10014 8fb00010

data指针的值是0x828000d0，在其所在的页面的偏移量是0xd0=208，208 - 176 = 32，而fkbuff{}的大小正好是32字节。按说是足够了，但如果哪地方把保留域长度增加了16字节呢，则Fkb头的仅剩16字节，有可能就向前越界了。

6、代码分析

一般问题定位到最后，就需要进行代码走查和分析了。针对本次故障，重点对ETH和XTH驱动进行分析。

6.1 ETH驱动

=== 接收：

bcm63xx_enet_isr()中：

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,30)
   napi_schedule(&pDevCtrl->napi);
#else
   netif_rx_schedule(pDevCtrl->dev);
#endif
```

采用NAPI机制，poll函数是bcm63xx_enet_poll_napi()，其中调用bcm63xx_rx()。

bcm63xx_rx()中：

```
pFkb = fkb_init(pBuf, RX_ENET_SKB_HEADROOM, pBuf, len - ETH_CRC_LEN );
```

bcm63xx_rx()中：

```
pFkb = fkb_init(pBuf, RX_ENET_SKB_HEADROOM, pBuf, len - ETH_CRC_LEN );
```

bcmdrivers/opensource/include/bcm963xx/bcmPktDma_bds.h中定义：

```
#define RX_ENET_SKB_HEADROOM    176
```

=== 初始化申请RxBuf：

bcm63xx_init_dev()中：

```
       /* initialize the receive buffers */
       if (init_buffers(pDevCtrl, i)) {
          printk("Unable to allocate rx packet buffers \n");
#if defined(CONFIG_BCM_FAP_PWRSAVE)
          bcmPktDma_unforceFapsAwake();
#endif
          return -ENOMEM;
       }
#if (defined(CONFIG_BCM_BPM) || defined(CONFIG_BCM_BPM_MODULE))
       gbpm_resv_rx_buf( GBPM_PORT_ETH, i, rxdma->pktDmaRxInfo.numRxBds,
          (rxdma->pktDmaRxInfo.numRxBds * BPM_ENET_ALLOC_TRIG_PCT/100) );
#endif
```

init_buffers()中申请RxBuf：

```
BufsToAlloc = rxdma->pktDmaRxInfo.numRxBds;
…
```

```
if (enet_bpm_alloc_buf_ring(pDevCtrl, channel, BufsToAlloc) == GBPM_ERROR)
{
    printk(KERN_NOTICE "Eth: Low memory.\n");
    /* release all allocated receive buffers */
    enet_bpm_free_buf_ring(rxdma, channel);
    return -ENOMEM;
    }
```

由于CONFIG_BCM_BPM宏已定义。驱动不是直接调用kmalloc()申请缓冲区，而是通过BPM（Buffer Pool Manager）。

```
static int enet_bpm_alloc_buf_ring(BcmEnet_devctrl *pDevCtrl,
        int channel, uint32 num)
{
    unsigned char *pFkBuf, *pData;
    uint32 context = 0;
    uint32 buf_ix;
    RECYCLE_CONTEXT(context)->channel = channel;
    for (buf_ix=0; buf_ix < num; buf_ix++)
    {
        if ( (pFkBuf = (uint8_t *) gbpm_alloc_buf()) == NULL )
            return GBPM_ERROR;
 /* 注意，在数据区前面，不仅要保留头部空间，还要给fkbuff{}预留32字节空间！ */
       pData =PFKBUFF_TO_PDATA(pFkBuf,RX_ENET_SKB_HEADROOM);

        /* Place a FkBuff_t object at the head of pFkBuf */
        fkb_preinit(pFkBuf, (RecycleFuncP)bcm63xx_enet_recycle, context);

        cache_flush_region(pData, (uint8_t*)pFkBuf + RX_BUF_SIZE);
        bcmPktDma_EthFreeRecvBuf(&pDevCtrl->rxdma[channel]->pktDmaRxInfo, pData);
    }
    return GBPM_SUCCESS;
}
```

而

```
#define bcmPktDma_EthFreeRecvBuf    bcmPktDma_EthFreeRecvBuf_Dqm
```

bcmPktDma_EthFreeRecvBuf_Dqm调用了 bcmPktDma_dqmXmitMsgHost(rxdma->fapIdx,
DQM_HOST2FAP_ETH_FREE_RXBUF_Q,
DQM_HOST2FAP_ETH_FREE_RXBUF_Q_SIZE, &msg);

将RxBuf交给了FAP。

6.2 XTM驱动

bcmdrivers/opensource/net/xtmrt/impl4/bcmxtmrt.c

=== 接收：

poll函数是bcmxtmrt_poll_napi()，调用bcmxtmrt_rxtask。

bcmxtmrt_rxtask()中：

```
pFkb = fkb_qinit(pBuf, RXBUF_HEAD_RESERVE, pucData, usLength, (uint32_t)rxdma);
#if (defined(CONFIG_BCM_BPM) || defined(CONFIG_BCM_BPM_MODULE)) || (defined(CONFIG_BCM_FAP) ||
defined(CONFIG_BCM_FAP_MODULE))
{
uint32 context = 0;

RECYCLE_CONTEXT(context)->channel = rxdma->pktDmaRxInfo.channel;
```

```
pFkb->recycle_hook = (RecycleFuncP)bcmxtmrt_recycle;
pFkb->recycle_context = context;
}
#endif
bcmxtmrt_process_rx_pkt (pDevCtx, rxdma, pFkb, dmaDesc.status, delLen, trailerDelLen) ;
```

bcmdrivers/opensource/net/xtmrt/impl4/bcmxtmrtimpl.h中定义：

```
#define RXBUF_HEAD_RESERVE          ((176 + 0x3f) & ~0x3f)      // 向上取整为64字节的整数倍
```

RXBUF_HEAD_RESERVE等于192！

=== 初始化：

DoGlobInitReq()中初始化分配Rxbuf，注意CONFIG_BCM_BPM宏已定义。

```
 #if (defined(CONFIG_BCM_BPM) || defined(CONFIG_BCM_BPM_MODULE))
if( (xtm_bpm_alloc_buf_ring(rxdma, BufsToAlloc)) == GBPM_ERROR )
{
/* release all allocated receive buffers */
xtm_bpm_free_buf_ring(rxdma);
kfree(pGi->rxdma[i]);
return -ENOMEM;
}

{
int s;
unsigned char *pSkbuff;
/*
* BCM_BPM_DYNAMIC_SKB_ALLOC assumes that all the skb-buffers
* in 'freeSkbList' belong to the same contiguous address range. So if you do any change to the allocation method below, make sure to rework the
BCM_BPM_DYNAMIC_SKB_ALLOC  sites.
*/
if( (rxdma->skbs_p = kmalloc(
(rxdma->pktDmaRxInfo.numRxBds * SKB_ALIGNED_SIZE) + 0x10,
GFP_KERNEL)) == NULL )
return -ENOMEM;
/*add by qiuxiaoxiao 2012.8.17 解决PPPOE BT下载丢包问题*/
rxdma->skbs_end_p = rxdma->skbs_p
+ (rxdma->pktDmaRxInfo.numRxBds * SKB_ALIGNED_SIZE) + 0x10;
/*end add by qiuxiaoxiao 2012.8.17 */
memset(rxdma->skbs_p, 0,
(rxdma->pktDmaRxInfo.numRxBds * SKB_ALIGNED_SIZE) + 0x10);

/* Chain socket skbs */
for (s = 0, pSkbuff = (unsigned char *)
(((unsigned long) rxdma->skbs_p + 0x0f) & ~0x0f); s < rxdma->pktDmaRxInfo.numRxBds;
s++, pSkbuff += SKB_ALIGNED_SIZE)
{
((struct sk_buff *) pSkbuff)->next_free = rxdma->freeSkbList;
rxdma->freeSkbList = (struct sk_buff *) pSkbuff;
}
}
```

```
gbpm_resv_rx_buf( GBPM_PORT_XTM, i, rxdma->pktDmaRxInfo.numRxBds,
(rxdma->pktDmaRxInfo.numRxBds * BPM_XTM_ALLOC_TRIG_PCT/100) );
#else
...

#endif
```

gbpm_resv_rx_buf()定义在kernel/linux/net/core/gbpm.c ，调用gbpm_resv_rx_hook_g指向的函数，实际就是bcmdrivers/broadcom/char/bpm/impl1/bpm.c。

在bmp.h中，定义BPM_XTM_ALLOC_TRIG_PCT等于15。

```
static int xtm_bpm_alloc_buf_ring(BcmXtm_RxDma *rxdma, UINT32 num)
{
    UINT8 *data, *pFkBuf;
    UINT32 context = 0;
    UINT32 buf_ix;

    RECYCLE_CONTEXT(context)->channel = rxdma->pktDmaRxInfo.channel;

    for (buf_ix=0; buf_ix < num; buf_ix++)
    {
        if( (pFkBuf = (uint8_t *) gbpm_alloc_buf()) == NULL )
        {
            printk(KERN_NOTICE CARDNAME ": Low memory.\n");
            return GBPM_ERROR;
        }

        /* Align data buffers on 16-byte boundary - Apr 2010 */
        data = PFKBUFF_TO_PDATA(pFkBuf, RXBUF_HEAD_RESERVE);

        /* Place a FkBuff_t object at the head of pFkBuf */
        fkb_preinit(pFkBuf, (RecycleFuncP)bcmxtmrt_recycle, context);

        cache_flush_region(data, (uint8_t*)pFkBuf + RXBUF_ALLOC_SIZE);
        bcmPktDma_XtmFreeRecvBuf(&rxdma->pktDmaRxInfo, (unsigned char *)data);
    }

    return GBPM_SUCCESS;
}
```

6.3 NBUF&FAP

驱动中通常调用fkb_init()初始化fkbuff：

```
static inline FkBuff_t * _fkb_init(uint8_t * pBuf, uint32_t headroom,
                                   uint8_t * pData, uint32_t len)
{
    FkBuff_t * fkb_p = PDATA_TO_PFKBUFF(pBuf, headroom);

#if defined(CC_FKB_HEADROOM_AUDIT)
    if ( headroom < FKB_HEADROOM )
        printk("NBUFF: Insufficient headroom <%u>, need <%u> %-10s\n",
               headroom, FKB_HEADROOM, __FUNCTION__);
#endif
```

```
    fkb_p->data = pData;
    fkb_p->len  = len;
    fkb_p->ptr  = (void*)NULL;   /* resets dirty_p, blog_p */

    fkb_set_ref( fkb_p, 1 );

    return fkb_p;
}
```

FKB_HEADROOM 等于176，这只是最小值，并不是要求头部大小固定这么大！

```
#define PFKBUFF_PHEAD_OFFSET     sizeof(FkBuff_t)
#define PFKBUFF_TO_PHEAD(pFkb)    ((uint8_t*)((FkBuff_t*)(pFkb) + 1))

#define PDATA_TO_PFKBUFF(pData,headroom)    \
 (FkBuff_t *)((uint8_t*)(pData)-(headroom)-PFKBUFF_PHEAD_OFFSET)

#define PFKBUFF_TO_PDATA(pFkb,headroom)     \
 (uint8_t*)((uint8_t*)(pFkb) + PFKBUFF_PHEAD_OFFSET + (headroom))
```

对于FAP -- bcmdrivers/broadcom/char/fap/impl1/fap_driver.c
重点看一下是怎么申请缓存的：

```
int fapBpm_pendEvtHandle( DQMQueueDataReg_S *msg_p )
{
    uint32 cmd  = msg_p->word1;
    uint8  drv = ((msg_p->word2 >> 28) & 0x0F);
    uint8  channel = ((msg_p->word2 >> 24) & 0x0F);
    uint8  seqId = ((msg_p->word2 >> 16) & 0xFF);
    uint16 numBufs = (msg_p->word2 & 0xFFFF);

    ...
    switch (cmd)
    {
       case HOSTMSG_CMD_ALLOC_BUF_REQT:
       {
          if (drv == FAPMSG_DRV_ENET)
          {
             fapIdx = getFapIdxFromEthRxIudma(channel);
             memAddr = (uint32) &pHostFapSdram(fapIdx)->alloc.bpm.enet[channel].bufAddr[0];
          }
          else
          {
             fapIdx = getFapIdxFromXtmRxIudma(channel);
             memAddr = (uint32) &pHostFapSdram(fapIdx)->alloc.bpm.xtm[channel].bufAddr[0];
          }

          if (gbpm_alloc_mult_buf(numBufs, (uint32_t *)memAddr)  == GBPM_ERROR )
          {
            BCM_LOG_ERROR( BCM_LOG_ID_FAP, "FAP FapIF alloc buf reqt" );
            return GBPM_ERROR;
```

```
        }

#if defined(CC_BPM_DBG)
        {
            memPtr = (uint32 *) memAddr;
            printk( "Bufs Alloc:");
            printk( "\t" );
            for(i=0; i < numBufs; i++, memPtr++)
            printk( "0x%p ", (void *) *memPtr);
            printk( "\n" );
        }
#endif

        /*
         * pFKB to pData conversion
         * BPM stores address to pFKB, whereas FAP used pData,
         * so after allocation of buffers from BPM, pFKB needs to be
         * converted to pData
         * TODO: optimization
         */
        memPtr = (uint32 *) memAddr;
        pBuf = (uint32 *) memAddr;

        for (i=0; i < numBufs; i++, memPtr++, pBuf++)
        {
          *memPtr =(uint32)
          PFKBUFF_TO_PDATA((*pBuf),RX_ENET_SKB_HEADROOM);
        }

        fapBpm_allocBufResp_Dqm( drv, channel, seqId, numBufs, memAddr);
        break;
      }
```

其中是调用gbpm_alloc_mult_buf()申请多个缓存。

注意，其中预留的头部空间大小是RX_ENET_SKB_HEADROOM，即176字节。而XTM驱动要求预留的空间大小是192字节！！！这样，XTM驱动接收报文后，写nbuff头部时就会越界，覆盖掉缓存前面的16个字节。

6.4 主动复现

添加调试代码，提前复现故障，思路是将存储分配内存的第一个指向0x82800000，当XTM收包后直接向前越界到wlan区域；在fapBpm_pendEvtHandle函数中修改的部分代码如下：

```
 if (drv != FAPMSG_DRV_ENET)
{
/* 目前发现破坏0x82800000之前一个页面是在wlan代码区中，尝试分配此处 */
memPtr = (uint32 *) memAddr;
if ((*memPtr) != 0x82800000)
{
    int j;
    uint32_t *tmpptr;

    for (j = 0; j < 4000; j++)
    {
```

```
            tmpptr = gbpm_alloc_buf();
            if ((unsigned int)tmpptr == 0x82800000)
            {
                gbpm_free_buf((void *)*memPtr);
                *memPtr = (uint32)tmpptr;
                printk("Change first buf to %p!\n", tmpptr);
                break;
            }
            else
            gbpm_free_buf(tmpptr);
            }
        }
}
```

同时在各个驱动添加告警打印：正常情况下，放到Rx Ring的地址应该是缓存的起始地址，加上32字节头部，加上保留的头部空间。在所有分配RxBuff的地方，包括ETH、XTM和FAP驱动，都加上检测，看哪地方向Rx Ring放入了错误的地址。例如XTM驱动中，在AssignRxBuffer()入口处加上：

```
if ( ((unsigned int)pucData- ((unsigned int)pucData& 0xffff0000))       <(RXBUF_HEAD_RESERVE + PFKBUFF_PHEAD_OFFSET) )
{
       printk("!!! Warn in AssignRxBuffer! pucData= %p\n", pucData);
                dump_stack();
   }
```

通过主动复现的方式，故障可以很快复现，出现crash打印（当然由于具体每个页面分配的内容不清楚，偶尔有两次复现比较慢，可能是由于越界的那一页正好是空白页，没什么影响）

7、解决故障

通过以上分析，最终定位在XTM驱动接收报文写nbuff头部时越界造成的，在fapBpm_pendEvtHandle的case HOSTMSG_CMD_ALLOC_BUF_REQT里需要对XMT和ETH驱动分别处理。最终修改的改动如下：

```
for (i=0; i < numBufs; i++, memPtr++, pBuf++)
{
    /*  ETH驱动要求预留的头部空间大小是176字节，而XTM驱动要求预留192字节！
* 如果统一预留RX_ENET_SKB_HEADROOM（176）字节的头部空间，
* XTM驱动的接收处理函数在填写nbuffer头部时就会越界。
*/
    if (drv == FAPMSG_DRV_ENET)
    {
        *memPtr =
    (uint32) PFKBUFF_TO_PDATA((*pBuf),RX_ENET_SKB_HEADROOM);
    }   /* RX_ENET_SKB_HEADROOM = 176 */
    else
    {
        *memPtr =
    (uint32) PFKBUFF_TO_PDATA((*pBuf),RX_XTM_SKB_HEADROOM);
    }  /* RX_XTM_SKB_HEADROOM = 192 */

}
```

8、验证故障

最终修改的代码如下，挂机2天未再出现crash打印，无线驱动phy模块没有再出现epc。该无线phy的故障可认为已解决。

正式代码.7z