复盘故障，分析异常信息，可以获取以下信息：

1）根据epc信息，查看/proc/251/maps，可以知道异常点发生在libc.so。然后反汇编这个动态链接库，根据偏移地址，可以算出这个故障发生在molloc中。

2）根据错误的地址信息，这个地址是4字节对齐的，说明指令可能没有错误。也即code区被破坏的可能性较小。

3）再次对比/proc/251/maps，查看错误地址信息，可以看到它位于堆中。堆的地址可能发生动态扩张。可以在内核中加个打印，看下异常时堆是怎么扩张的。

4）堆只有malloc分配内存会使用。malloc本身出问题可能性不大，但观察malloc分配地址的过程，发现无论分配内存的大小，在这块内存前面，预留了8个字节，可能是用于管理内存区的。如果这8个字节被破坏掉，那么可能会导致malloc程序出错。

5）这8个字节怎么被破坏呢？有几种可能性：1）前面一块内存操作溢出，破坏了它的内存，2）内存跑飞，被其他的进程随机破坏。由于malloc分配的内存块，大小和位置都比较随机，因此第二种情况的可能性较小。更多是第一种可能性，也即前面的内存溢出，导致被破坏。

6）走查代码，分析cspd进程用户态代码有哪些地方发生malloc。注意到大多数malloc都被封装为getUB。因此我们可以首先在getUB中加个打印，把它分配出去的每一块内存的地址都打印出来。把它每次对内存的释放也打印出来。

7）复现问题，比对出问题的地址，和所有分配出去的内存。查找临近出问题内存块的内存是被谁分配的。

8）至此，问题应该可以缩小到一个较小的范围。

问题最终原因，拷贝陆亦芬的分析：

9）经验：在往用户态拷贝内存时，要特别注意。如果越界，有可能把用户态的堆或者栈破坏掉。

首先，用户态需要传入一个数组，告知内存的头指针，和内存的长度。而不应该仅仅只传递一个指针。

另外，内核态在向用户态空间拷贝内存时，也应该检查待拷贝内存的大小。

10）为什么在8月份已经发现这里有strlen的故障之后，过了两个月仍然在报故障？

因为ioctl中有两个case：BOARD_IOCTL_SET_KERNEL_DIE和BOARD_IOCTL_GET_KERNEL_DIE，但是当时只修改了前面一个set case的strlen为strnlen，没有修改get case的strlen为strnlen，导致了悲剧的一再发生。有开发人员的问题，也有管理的问题。没有对这块代码好好走查。

3、在VOID cspd_misc_main(DWORD dwEvent,VOID *lpMsg, WORD wMsgLen, WORD wState)函数中使用相同方法继续加入跟踪，调用关系如下：

cspd_misc_main->cspd_misc_asynmsg->pdt_proc_init->pdt_processsDieFile->devCtl_getSysDieInfo->devCtl_boardIoctl进入内核态board_ioctl，通过查看这个代码发现如下：

```
  case BOARD_IOCTL_GET_KERNEL_DIE:
      if(copy_from_user((void*)&ctrlParms,(void*)arg,sizeof(ctrlParms))== 0)
{
        unsigned char *SysDieData;
        SysDieData = kzalloc(ctrlParms.strLen, GFP_ATOMIC);
        /*没有判SysDieData 是否为空*/
memcpy((char*)SysDieData, __va((unsigned long)(MEM_DIE_RESERVE_ADDR -
MEM_DIE_RESERVE_SIZE)),MEM_DIE_RESERVE_SIZE);
         __copy_to_user(ctrlParms.string,SysDieData, strlen(SysDieData));
         kfree(SysDieData);
          ret = 0;
   }
```

ctrlParms.string是用户态通过malloc获得的虚拟地址，copy到这个缓冲区的数据长度是根据strlen(SysDieData)这个来计算的，由于MEM_DIE_RESERVE_ADDR - MEM_DIE_RESERVE_SIZE内存中数据并没有初始化，

因此根据'\0'来确定数据的长度并不是一个固定值，况且在每次上电时候里面值是未知的，因此长度可能是一个任意的值。经过试验证实，在出现问题时copy的长度超过了缓冲区的长度，破坏了堆上的数据。

南京陆亦芬的解决方法，和这个思路不太一样。

1）既然malloc的管理结构可能出现问题，那么就频繁地malloc，看有问题。

2）在关键函数前后加打印，判断是哪一次malloc出现的问题。

```
      printf("%d ptPCB->pEntry %p %d\n",__LINE__,ptPCB->pEntry, getpid());
      ptPCB->pEntry(ptMsg->dwMsgId,pMsgData,ptMsg->wMsgLen,ptPCB->wState);
      printf("%d ptPCB->pEntry %p %d\n",__LINE__,ptPCB->pEntry, getpid());
```

3）分析所有的异常点，找到最频繁的一个。继续逼近。由于问题出现频率高，因此可以做很多次实验。

4）最终，肯定可以逼近是哪个函数出现的问题。

5）最开始的第一个步骤，频繁地malloc，可能并没有起到作用，只是一种加速复现的方法。

逼近法。

刘振华分析了上十天，没有进展

1）一开始对epc信息的分析不够。所有的故障现象都指向了malloc和堆。但直到故障的最后阶段，仍然再尝试确认这个现象。没有围绕这个信息思考如何去解决问题。

2）可能不了解堆内存分配的特点。如malloc的管理数据结构，堆溢出的特点等。事实上，我也是现在在清楚地了解到。

H168N_TA版本malloc异常故障查证记录

【作者】　刘振华

【时间】　20120827

【概述】　配置无线的web页面或者拷机过程中导致单板异常重启

故障描述

1. H168N阿根廷TA版本，H168NV1.1T9_TA，基于CSP平台，单号为：613001531222，主题：单板正常操作时自动重启。
2. 问题单描述如下：

测试人员描述

Linux version 2.6.30 (xialei@localhost.localdomain) (gcc version 4.4.2 (Buildroot 2010.02-git)) #1 SMP

测试环境：武研所3楼实验室VD外网拷机环境

测试步骤：

1.恢复默认配置

2.修改单板自带路由连接vdsl_telecom的VLAN为100，修改用户名密码，成功获取外网地址

3.新建PPPOE路由连接vdsl-route1，获取到内网地址

4.开启IGMP PROXY，选择IGMP WAN连接为vdsl-route1

5.将LAN1-4绑定至vdsl-route1。下挂PC可以上网，机顶盒可以正常播放。

6.在WEB中配置DDNS参数后，单板自动重启。

附件为log（故障为最后一次重启处）

开发确认人：刘金成

zyk inet->mark = 11
UTC time: 2012-7-14 1:53:36
do_page_fault() #2: sending SIGSEGV to cspd for invalid write access to
006e222c (epc == 2af47d08, ra == 2af4793c)
1193:01:46 [OSS_cspd][Error] [oss_fault_trap.(252)SigHandler] Process <328> receive signal 11, epc == 2af47d08, ra == 2af4793c
1193:01:46 [OSS_cspd][Error] [oss_fault_trap.(254)SigHandler] Signal information:
do_page_fault() #2: sending SIGSEGV to cspd for invalid read access from
006e222c (epc == 2af47ce0, ra == 2af4793c)
Found userspace die error, prepare to save.....

分析如下：

通过对log进行分析发现，异常点在malloc函数中，这个故障先前在上海时发现过，当时确认的原因为配置文件加密导致故障出现，但是现在的版本中配置文件是不加密的，因此该故障的根本原因可能没有找到。后续测试的过程中又复现几次，log如下：

前两个异常的进程由于当时在操作WLAN页面通过PID可推断为wlan_config进程，但是第三个log的异常点不好推断，但应该不在wlan_config进程运行过程中。

查证过程

日期（20120826）

初步分析是在malloc内存的过程中导致的异常，且在点击WLAN页面的时候出现的故障频率较高，因此跑IPTV，运行按键精灵，不停的点击WLAN页面复现故障。同时该版本将日志和调试信息打开，便于故障出现后进行分析。

结果：测试12个小时，发现单板并没有异常重启

日期（20120827）

猜想：由于当时的场景是要进行大流量稳定性拷机，会不会再拷机环境下更容易复现，初步想法是在拷机环境中，版本中运行应用测试（先申请一块内存，然后while中不停的申请大内存），同时不停的点击WEB页面进行拷机测试，复现故障。

由于忙其他事情且无拷机环境，暂未进行该项测试。

日期（20120831）

曾静测试DDNS时，配置DDNS的页面会导致单板异常重启，经确认都挂在了malloc函数里边。且重复复现的过程中，复现规律为：单板恢复默认配置，单板重新启动后，等待1~2分钟，然后配置DDNS的web页面，单板会出现异常。

反汇编libc库，找到异常点为malloc函数里边。打开DDNS的打印，跟踪代码发现异常点出现在ddns_mgr.c的ddnsAddClientListNode函数中，如下：

```
if(g_ddns_client_list_init_flag)
{
    /*遍历所有节点,计算当前链表的节点数*/
    list_for_each(ptListItemPtr, &g_ddns_client_list)
    {
        dwNodeCount++;
    }
    OssUserLogInfo("dwNodeCount is:%d,%d", dwNodeCount,sizeof(T_DDNS_CLIENT_LIST));
    /* 创建条目 */
    ptListItemTmp = (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
    if (NULL == ptListItemTmp)
    {
        OssUserLogId(LOGID_PUBLIC_GET_UB_FAIL, "");
        return RET_DDNS_UNKNOW_ERROR;
    }
```

故障确认如下：

1. 怀疑此时系统内存分配有问题。

修改代码如下：

```
if(g_ddns_client_list_init_flag)
{
    /*遍历所有节点,计算当前链表的节点数*/
    list_for_each(ptListItemPtr, &g_ddns_client_list)
    {
        dwNodeCount++;
    }
    OssUserLogInfo("dwNodeCount is:%d,%d", dwNodeCount,sizeof(T_DDNS_CLIENT_LIST));
    pbuf = (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
      OssUserLogInfo("malloc ok =====no free===1====luise 0x%x",pbuf);
    /* 创建条目 */
    ptListItemTmp = (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
    OssUserLogInfo("malloc ok =====4buf===2====luise  0x%x ",(void *)ptListItemTmp);
RetUB(pbuf);
    if (NULL == ptListItemTmp)
    {
        OssUserLogId(LOGID_PUBLIC_GET_UB_FAIL, "");
        return RET_DDNS_UNKNOW_ERROR;
    }
```

测试结果：发现pbuf申请内存后的那句打印可以打印出来，而ptListItemTmp申请内存后的打印语句没有，感觉此时申请内存本身应该没有问题。

2. 怀疑是否由于故障出现时，来不及打印其后的语句，实际是对内存的操作有问题，于是修改代码如下：

```
OssUserLogInfo("malloc ok =====4buf===2====luise  0x%x ",(void *)ptListItemTmp);语句后加延时
usleep(5000);
tcflush();
```

结果：故障出现时，依然没有这句打印，应该不是没来及打印。

3. 将1的代码的**RetUB(pbuf);代码注释掉时，出现第一句打印都没有打印出来的情况**，说明第一句申请内存的语句就挂死了，这种情况复现过3次。

4. 修改1的代码在pbuf前加调试代码，申请pbuf1然后释放、申请pbuf2然后释放、申请pbuf3然后释放，这种情况下故障出现时，依然出现在ptListItemTmp这行挂死，代码如下：

```
if(g_ddns_client_list_init_flag)
 {
     /*遍历所有节点,计算当前链表的节点数*/
    list_for_each(ptListItemPtr, &g_ddns_client_list)
   {
       dwNodeCount++;
     }
     OssUserLogInfo("dwNodeCount is:%d,%d", dwNodeCount,sizeof(T_DDNS_CLIENT_LIST));
     pbuf 1= (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
 RetUB(pbuf1);
pbuf 2= (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
RetUB(pbuf2);
pbuf 3= (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
     RetUB(pbuf3);
pbuf = (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
        OssUserLogInfo("malloc ok =====no free===1====luise 0x%x",pbuf);
     /* 创建条目 */
    ptListItemTmp = (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
     OssUserLogInfo("malloc ok =====4buf===2====luise  0x%x ",(void *)ptListItemTmp);
RetUB(pbuf);
     if (NULL == ptListItemTmp)
     {
       OssUserLogId(LOGID_PUBLIC_GET_UB_FAIL, "");
       return RET_DDNS_UNKNOW_ERROR;
     }
```

5. 而这种情况下，还没有出现过，点了有10次左右，因为故障基本必现，所以没有做过多次试验

```
if(g_ddns_client_list_init_flag)
 {
     /*遍历所有节点,计算当前链表的节点数*/
    list_for_each(ptListItemPtr, &g_ddns_client_list)
   {
       dwNodeCount++;
     }
     OssUserLogInfo("dwNodeCount is:%d,%d", dwNodeCount,sizeof(T_DDNS_CLIENT_LIST));
     pbuf 1= (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
pbuf 2= (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
pbuf 3= (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
pbuf = (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
        OssUserLogInfo("malloc ok =====no free===1====luise 0x%x",pbuf);
     /* 创建条目 */
    ptListItemTmp = (T_DDNS_CLIENT_LIST *)GetUB(sizeof(T_DDNS_CLIENT_LIST));
     OssUserLogInfo("malloc ok =====4buf===2====luise  0x%x ",(void *)ptListItemTmp);
RetUB(pbuf1);
RetUB(pbuf2);
```

RetUB(pbuf3);

RetUB(pbuf);

```
if (NULL == ptListItemTmp)
{
    OssUserLogId(LOGID_PUBLIC_GET_UB_FAIL, "");
    return RET_DDNS_UNKNOW_ERROR;
}
```

6. 在测试复现的过程中发现：如果在升级版本前DDNS的配置成功了，然后再升级版本，版本升级成功后重新启动，启动过程中也会出现异常

Log如下：

1192:58:49 [adev_mgr][Info] [access_dev_mgr.(2049)adevReg] moudle 65794 reg adev trigger!

do_page_fault() #2: sending SIGSEGV to cspd for invalid write access to

006ffc34 (epc == 2af2bd08, ra == 2af2b95c)

1192:58:49 [OSS_cspd][Error] [oss_fault_trap.(243)SigHandler] Process <316> receive signal 11, epc == 2af2bd08, ra == 2af2b95c

1192:58:49 [OSS_cspd][Error] [oss_fault_trap.(245)SigHandler] Signal information:

1192:58:49 [OSS_cspd][Error] [oss_fault_trap.(248)SigHandler] Signal number: 11 [SIGSEGV]

1192:58:49 [OSS_cspd][Error] [oss_fault_trap.(249)SigHandler] Signal code: 1

1192:58:49 [OSS_cspd][Error] [oss_fault_trap.(250)SigHandler] Signal value: 2122316128

异常进程是316，是资源管理模块，srm_mgr.c；这个现象也很奇怪，**是不是与申请的内存破坏了其他进程的内存空间有关系。有待验证，这种现象也出现在315和314的进程中。**

经过以上的测试分析发现两个问题需要确认：

1. 对以上的第3步要确认，在正常的代码申请内存前先申请一块代码，然后不对其进行释放，故障点就会出现在加的调试代码申请内存上。如果是的话，可以从这条线索上进行分析。
2. 如果经1的确认后，发现故障点还是出现在**ptListItemTmp**的申请上，那会不会是对该块内存的操作有问题，可以给其多申请些内存进行测试，看故障点是否还存在，再继续分析。

日期（20120906）

对先前的两个思路测试如下：

1、在ddns_mgr和wlan_config模块的流程中添加如下的代码，测试发现还是会挂死，应该与该ddns链表节点的操作无关，与申请内存有关。

2、预留一块内存后，没有点出来该故障

但是通过上述两点的分析，应该与链表节点的操作没有关系，应该与申请内存有关

于是了解了缺页处理do_page_fault()的函数实现，

```
if (vma->vm_start <= address)
        goto good_area;
 if (!(vma->vm_flags & VM_GROWSDOWN))
        {
        printk("here3\n");
        goto bad_area;
        }
```

正常的缺页处理有两种情况：

1、触发异常的线性地址处于用户空间的vma中，但还未分配物理页，如果访问权限OK的话内核就给进程分配相应的物理页了

2、触发异常的线性地址不处于用户空间的vma中，这种情况得判断是不是因为用户进程的栈空间消耗完而触发的缺页异常，如果是的话则在用户空间对栈区域进行扩展，并且分配相应的物理页，如果不是则作为一次非法地址访问来处理，内核将终结进程

从打印来看，故障出现时访问的线性地址不属于这两种情况。而且故障出现时log如下

here3

do_page_fault() #2: sending SIGSEGV to cspd for invalid write access to

006edc44 (epc == 2af2bd08, ra == 2af2b93c)

00400000-0067f000 r-xp 00000000 1f:00 34 /bin/cspd

0068f000-00699000 rw-p 0027f000 1f:00 34 /bin/cspd

00699000-006e8000 rwxp 00000000 00:00 0 [heap]

2aaa8000-2aaad000 r-xp 00000000 1f:00 1326 /lib/ld-uClibc.so.0

2aaad000-2aaae000 rw-p 00000000 00:00 0

访问的地址首先不在进程的堆空间里，且

堆（heap）：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用free等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）

应该可以推断出应该是正在申请内存的过程中挂死的

如果分析到这里，那malloc都失败，这条路不好继续分析下去。

因此从以下三点继续分析：

1. 打开coredump功能，确认挂死点是否是在malloc这里
2. 为什么单板重启后要放置一会再点挂的几率大些
3. 将申请内存的size改小，测试是否出现，进行规避操作

对于1的试验结果为：产生故障时没有生成相应的coredump文件，但是自己写的测试程序能产生coredump，将异常添加到该进程也不会产生coredump文件。

对于2目前没有好的想法进行分析，运行一会儿会有内存碎片等？？？？

对于3进行测试，先进行规避操作，如果可行的话先规避实现。

日期（20120926）

今日在H168N 匈牙利版本，复现了此故障。

操作步骤：1、使用龙卷风发送大量的DHCP请求报文 2、插拔LAN数据线

现象：单板出现cspd挂死现象。

Log附件：

背景：武汉H168NV11_HU版本在启动过程中，或者在页面上点击DDNS页面，CSPD进程会出现段错误。

一、 malloc异常问题查证流程如下：

1、根据EPC值找到发生异常出现在malloc函数调用里面，由于malloc函数正常情况不会出现异常（分配失败或成功），因此怀疑异常是由于堆上的malloc的管理结构被破坏，因此考虑在执行每个二级进程之前后进行malloc分配，用于证明堆上的管理结构是否被破坏,构造这个函数是需要技巧的，尽可能将每种大小分配的malloc包括进去，代码如下：

```
static VOID test_malloc_valid(char *s, int line)
{
    void *test_p;
    int size;

    printf("%s enter test %d\n", s,line);
    for(size = 1; size < 16*1024; size++)
    {
        test_p = malloc(size);
        if(test_p){
            free(test_p);
        }
    }
    printf("%s exit test %d\n", s,line);
}
```

2、由于每次发生异常都出现在cspd里面，并且cspd二级进程处理在以下两个函数里面SynRunProcess、RunProcess，因此在两个函数进行跟踪，跟踪方法主要如下：

```
static VOID RunProcess(THREAD_PCB *ptThreadPCB,T_MSG *ptMsg)
{
    。。。。。。。。。。。。。。。。
    test_malloc_valid(__FUNCTION__,__LINE__);
```

```
    printf("%d ptPCB->pEntry %p %d\n",__LINE__,ptPCB->pEntry, getpid());
    ptPCB->pEntry(ptMsg->dwMsgId,pMsgData,ptMsg->wMsgLen,ptPCB->wState);
    printf("%d ptPCB->pEntry %p %d\n",__LINE__,ptPCB->pEntry, getpid());

    test_malloc_valid(__FUNCTION__,__LINE__);
    。。。。。。。。。。。。。。。。
}

VOID SynRunProcess(T_PCB *ptPCB,T_MSG *ptMsg)
{
    。。。。。。。。。。。。。。。。
    test_malloc_valid(__FUNCTION__,__LINE__);

    printf("%d ptPCB->pEntry %p %d\n",__LINE__,ptPCB->pEntry, getpid());
    ptPCB->pEntry(ptMsg->dwMsgId,pMsgData,ptMsg->wMsgLen,ptPCB->wState);
    printf("%d ptPCB->pEntry %p %d\n",__LINE__,ptPCB->pEntry, getpid());
    test_malloc_valid(__FUNCTION__,__LINE__);

    。。。。。。。。。。。。。。。。
}
```

通过以上代码进行复现可以发现在出现malloc异常地址pEntry通常为cspd_misc_main函数地址，因此在这个函数继续往下跟踪。有时也会发生在dhcp6s/wlan_config线程，由于用户线程可能随时被切换出去，因此发生在其他线程也是可能的，我们在这里紧盯了出现概率高的地方。

3、在VOID cspd_misc_main(DWORD dwEvent,VOID *lpMsg, WORD wMsgLen, WORD wState)函数中使用相同方法继续加入跟踪，调用关系如下：

cspd_misc_main->cspd_misc_asynmsg->pdt_proc_init->pdt_processsDieFile->devCtl_getSysDieInfo->devCtl_boardIoctl进入内核态board_ioctl，通过查看这个代码发现如下：

```
  case BOARD_IOCTL_GET_KERNEL_DIE:
      if(copy_from_user((void*)&ctrlParms,(void*)arg,sizeof(ctrlParms))
== 0)
{
        unsigned char *SysDieData;
        SysDieData = kzalloc(ctrlParms.strLen, GFP_ATOMIC);
        /*没有判SysDieData 是否为空*/
memcpy((char*)SysDieData, __va((unsigned long)(MEM_DIE_RESERVE_ADDR -
MEM_DIE_RESERVE_SIZE)),MEM_DIE_RESERVE_SIZE);

        __copy_to_user(ctrlParms.string,SysDieData, strlen
(SysDieData));
         kfree(SysDieData);
          ret = 0;
   }
```

ctrlParms.string是用户态通过malloc获得的虚拟地址，copy到这个缓冲区的数据长度是根据strlen(SysDieData)这个来计算的，由于MEM_DIE_RESERVE_ADDR - MEM_DIE_RESERVE_SIZE内存中数据并没有初始化，

因此根据'\0'来确定数据的长度并不是一个固定值，况且在每次上电时候里面值是未知的，因此长度可能是一个任意的值。经过试验证实，在出现问题时copy的长度超过了缓冲区的长度，破坏了堆上的数据。

4、这段代码属于H168N转到武汉之前，在上海开发的“死机文件”功能。上次陆亦芬和彭克青查的WLAN代码段被破坏的故障，也是这个“死机文件”功能引入的。建议我们再详细走查一下这个代码。
