Hi, all:

   某线上模块出现了“随机core”，在我们无法查出问题之际，在P、Y同学大神的带领下，Y、J、Z同学协助，一起定位了core的原因。虽然最后的原因查出来是b2log库里面的一个低级buffer长度计算错误的BUG，但是排查的过程让我想起了一句古语：“**言之不足，故嗟叹之，嗟叹之不足，故咏歌之，咏歌之不足，不知手之舞之，足之蹈之也**”!，所以将排查过程整理，分享给大家，尤其是像我这样的菜鸟同学，一起学习、领略高工风采！

       第一：core初步分析

       某线上模块报出core，gdb查看：

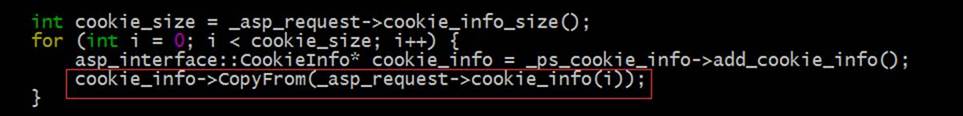
       Program terminated with signal 11, Segmentation fault



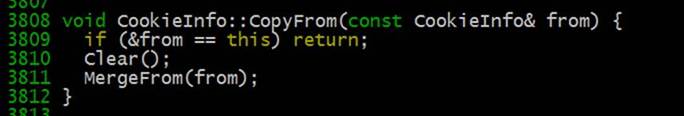
       初步查看，看到某线上模块 core在了一个ProtoBuf的CopyFrom函数，但this指针并不是NULL，因而不是空指针引起的。

       第二：结合代码初步分析：

       某线上模块的相关代码为：



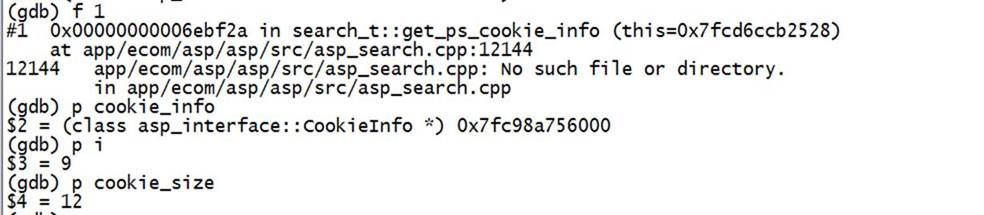
       Pb的相关代码为：



    从代码上看，我们的cookie\_info是由\_ps\_cookie\_info->add\_cookie\_info（）得来，而且并不是NULL，而core又显示了某线上模块 core在cookie\_info->CopyFrom()函数的第一步：Clear()；

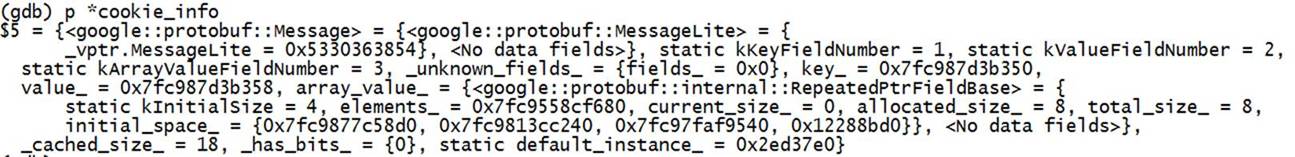
    第三步：进入frame 1继续分析，因为frame 0是protobuf生成的代码区域，而PB有BUG的概率几乎没有，因而我们并不去查看frame 0

    进入frame 1， 首先来查看一下当前的 cookie\_info 这个指针和i这个变量



       从gdb给出的信息来看： \_*某线上模块*\_request总共有 12 个cookie\_info，我们在i=9, 即第9个cookie\_info调用CopyFrom时core

       第四步：打印当前cookie\_info的详细信息：

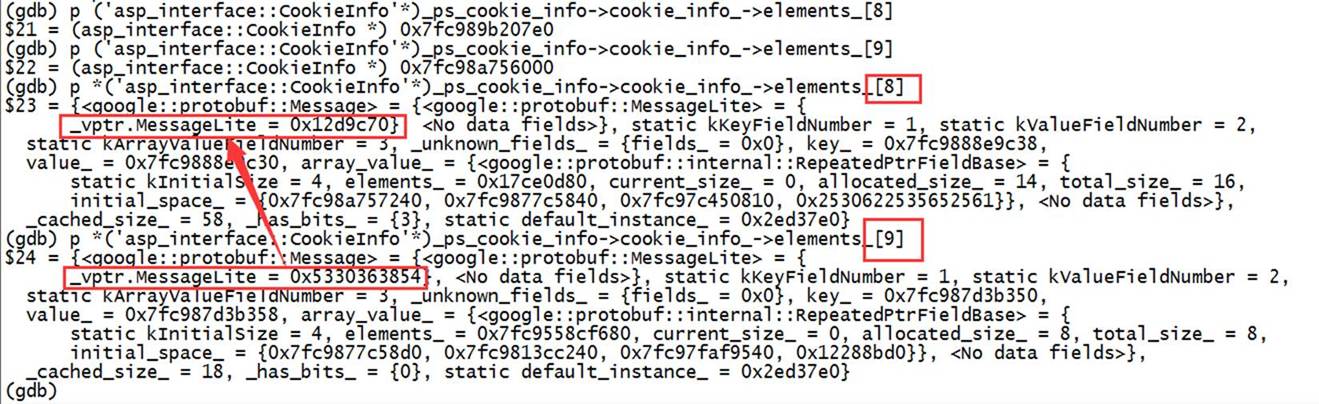


       似乎看起来没什么问题

       第五步：P同学神技：P同学看到了**\_vptr.MessageLite = 0x5330363854**， 敏锐的嗅觉立即产生一个想法，就是虚函数表指针被破坏了，如果虚函数表指针被破坏，那我们的cookie\_info调用Clear()这个继承自protobuf::MessageLite的虚函数时，自然会去尝试在这个“坏了的”虚函数表指针指向的虚函数表找到Clear函数的地址，而这种尝试必然会引发不可预料的错误，比如说core。

       其实，如果我们静下心来观察一下这个虚函数表指针：**0x5330363854就会发现，这是一个很高很高的地址，一般来说我们的程序编译出来的程序对象（包括函数、数据）的地址都是较低的地址**。但在排查core的时候，我就没有考虑到这个问题，而是认为没什么异常~而高工看到地址就会立即意识到这个地址可能有问题！

       为了验证这一个想法，我们打印出其他的cookie\_info的详细信息：

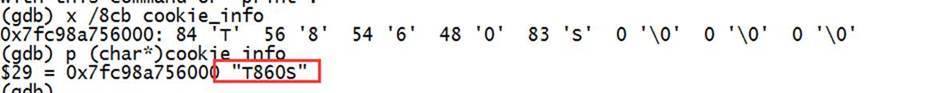


       通过再继续打出\_ps\_cookie\_info的第7，10个cookie\_info的详细信息可以看到，他们的虚函数表指针均为正常的： 0x012d9c70（如前所述，是一个较低的地址），至此，我们可以确认，某线上模块 core是因为一个cookie\_info对象的虚函数表指针被破坏了，导致其调用虚函数Clear()的时候出问题。

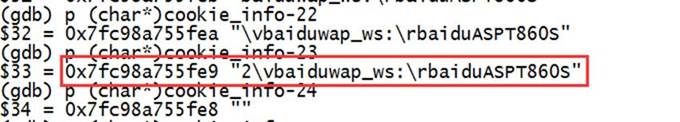
       第六步：既然是虚函数表指针被破坏了，那么究竟是怎么被破坏的。P同学神技：打印出坏了的虚函数表指针的内容是什么先看看！

       这里普及一下知识：在GCC和MVC编译器中，对虚函数的实现采用了**将虚函数表指针放在对象的内存布局的最前面**的方法~，据说还有将虚函数表指针放在对象的内存的最后面的编译器，不过后者没见过~~

       因为我们的机器是64位机器，因而虚函数表指针就是8个字节，所以cookie\_info这个对象内部的虚函数指针就位于这个对象的前8个字节。



       打印出来被破坏的虚函数表指针的内容时，可以看到是一个有意义的字符串！，P同学继续试图打印完整的一个字符串：



       可以看到，这个完整的字符串起始地址是：cookie\_info-23，为：2\vbaiduwap\_ws:\rbaidu某线上模块T860S

       第七步：看到这个字符串，P同学神技：瞬间意识到，这个字符串的编码是ProtoBuf编码！

       我们按照P同学的思路仔细查看这段字符串的内容：’\v’的ASCII码值为11，而其后面的”baiduwap\_ws”的长度就是11，而’\r’的ASCII码值为13，其后面的” baidu某线上模块T860S”长度就为13，这正好是protobuf的编码

       而前面的 “2”和”\r”的ASCII码值分别为：50和58，这两个数字是ProtoBuf的filed\_number以及type的编码，根据[pb的编码](https://developers.google.com/protocol-buffers/docs/encoding" \l "structure)可以计算出：50/8 = 6..2, 即filed number是6而其类型2代表“Length-delimited”即字符串，而58/8=7..2,，即filed\_number是7，类型仍然是“Length-delimited”。

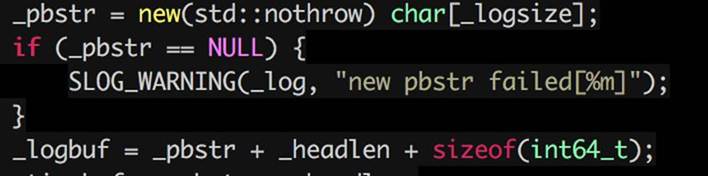
       现在获得的信息是：这个有意义的字符串是来自于一个protobuf，而且其field\_number分别是6和7

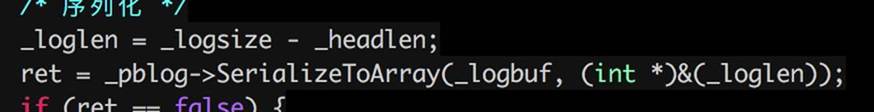
       对某线上模块熟悉的同学，知道这个字符串是计费名和模板名， 根据上面P同学给出的信息查找到一个对应的pb:



       第八步：在知道了这个Pb之后，P同学意识到可能是b2log库的问题，因为某线上模块会写pblog，写之前是使用b2log将一个protobuf对象序列化之后写到文件中的。

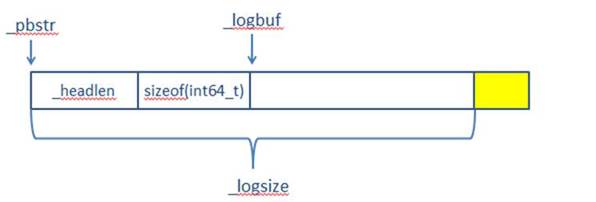
       第九步：Y同学大神闪亮登场，找到了b2log的一个BUG！：





       从上面的代码可以看出，\_pbstr就是b2log应该使用的一个缓冲区，其大小为： \_logsize

       而b2log在使用的时候，这个缓冲区划分成如下的格式：

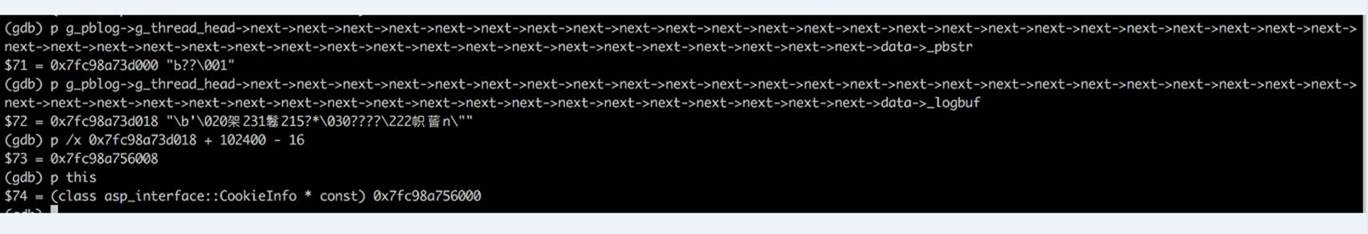


       所以，\_logbuf后面可以使用的空间大小是：\_logsize-\_headlen-sizeof(int64\_t)，而在实际使用的时候，b2log的代码允许pb序列化过程使用\_logbuf起始后面的\_logsize-\_headlen这么长的空间，**多余了8个字节**！

       正是因为这个pb序列化的时候，可能覆盖掉了分配给b2log的缓冲区的后面的8个字节，而这8个字节可能是其他内存对象的区域！这种覆盖的操作的后果是不可预知的，对于本次的core，就是因为pb序列化的时候，刚好超出合法的buffer，多写了至少5个字节到合法buffer的后面，而这5个字节恰好是cookie\_info对象的最前面，即存放虚函数表指针的位置，于是虚函数表指针被破坏了，调用虚函数的时候就发生了core。

       加入上面pb写超的数据只是某个对象的其他数据成员的区域，那么某线上模块就不会core，只是一些计算发生了错误，而这种细小的错误必然是非常难以发现的，想到这一点，就会觉得core了是最好的，要不然线上的问题一直难以发现~

       第十步：Y同学神技：确认是pb写超的原因！



       Y同学大神快速地在b2log的线程数据里面，找到了对应的被写超了的\_logbuf，然后，在打印出了\_logbuf+\_logsize-16的（\_logsize在线上配置是102400）地址发现是：0x7fc98a756000，而这个地址正好对应了我们前面core发生时的那个cookie\_info。

       关于这一点，我确实没有想到Y同学大神是如何快速定位到b2log的对应线程数据的，这可能就是高工的本质~~欢迎Y同学大神补充~

       至此，某线上模块的随机core被两位高工定位到了问题，问题的根源在于b2log的一个低级错误，而排查的过程处处体现了高工们敏锐的嗅觉和超群的技术！

       最后，留下两个思考的问题：

1.      假如在第六步，P同学打印出了虚函数表指针的内容，发现并不是一个有意义的字符串，而是乱七八糟的数据，那这个时候，P同学将如何追查问题呢？这就留在下一次的core中继续由P同学展现他的神技了！

2.      假如我们这种buffer越界写的时候并没有写坏函数的地址或者虚函数表的地址，而仅仅是写坏了别人读取的数据内容，那么就不会发生core，只会发生莫名其妙的不符合预期的计算结果，那面对这种情形，高工们如何分析并定位BUG？这恐怕也只能到时候再次膜拜学习吧~