虚表地址被改坏导致coredump

Created	@Jun 11, 2020 7:54 PM
Tags	CPP
Updated	@Jun 11, 2020 8:12 PM

1. coredump发生及第一印象

2020-05-27 GarenaTest环境gamesvr进程出现coredump。相关代码如图1，堆栈如图2。从堆栈看进程core在了`CGameLogicMgr::_DoMsgLogicDispatch`函数第1309行，并且从后续其它的coredump文件看虽然都core在了1309行，但是堆栈显示的函数层级却不一样（写这份文档时corefile已经被删除，只能拿到图2这一个当时截的堆栈图）。

这里有两点值得思考的地方：

1. `poIGameLogic`为非空指针（图2），为什么会core在1309行？
2. 为什么虽然都core在1309行，但是多次coredump产生的函数堆栈不一样？

带着这两个问题，继续排查。

```
1292    ····HashMap<uint32_t,CGameLogic*>::iterator·iter·=·m_oCmdToLogicMap.find(dwMsgID);
1293    ····if·(iter·==·m_oCmdToLogicMap.end())
1294    ····{
1295    ········if·(SSID_SVRHEARTBEAT·!=·dwMsgID·&&·CSID_CMD_HEARTBEAT·!=·dwMsgID·&&·CSID_ANTIDATA_REQ·!=·dwMsgID·)
1296    ········{
1297    ············LOGERR("Cannot·Find·MsgHandler·For·MsgID[%u].",·dwMsgID);
1298    ········}
1299    ········return·false;
1300    ····}
1301
1302    ····CGameLogic··*poIGameLogic·=·iter->second;
1303    ····if·(!poIGameLogic)
1304    ····{
1305    ········myAssert();
1306    ········return·false;
1307    ····}
1308
1309    ····poIGameLogic->HandleMsg(poAcnt,·dwMsgID,·sUnpackedPkgBody);
1310
1311    ····return·true
1312    }
```

图1. 出问题的代码

```
(db) bt
#  0x0000000000280c996 in __gnu_cxx::hashtable<std::pair<int const, HashMap<int, ResWrapper<ResData::ResMangoUse
 ResWrapper<ResData::ResMangoUserGroup>::ConfMap, __gnu_cxx::hash<int>, std::equal_to<int> > > >, std::equal_t
_insert (this=0x623f2f8, __obj=...) at /usr/lib/gcc/x86_64-redhat-linux/4.4.6/../../../../include/c++/4.4.6/ba
#  0x000000000280b33d in __gnu_cxx::hash_map<int, HashMap<int, ResWrapper<ResData::ResMangoUserGroup>::ConfMap,
ngoUserGroup>::ConfMap, __gnu_cxx::hash<int>, std::equal_to<int> > > >::operator[] (this=0x623f2f8, __key=@0x7
#  0x0000000002805e36 in ResWrapper<ResData::ResMangoUserGroup>::PostInit (this=0x623f2d0, iLanguage=566670424)
#  0x00000000022f4e37 in CGameLogicMgr::_DoMsgLogicDispatch (this=0x62459e0, poAcnt=0x21c6b458, dwMsgID=1616, s
#  0x00000000022f47fa in CGameLogicMgr::DoMsgLogicDispatch (this=0x62459e0, poAcnt=0x21c6b458, dwMsgID=1616, sU
#  0x00000000022f6f33 in CGameLogicMgr::_DealCltAppPkg (this=0x62459e0, poAcnt=0x21c6b458, sBufferIn=0x605c741
#  0x00000000022f8231 in CGameLogicMgr::HandleCSPkg (this=0x62459e0, stGameHead=..., sBufferReadIn=0x605c741 "
#  0x0000000002b98afd in CGameSvr::_DealCltPkg (this=0x605c6c0) at gamesvr.cpp:1496
#  0x0000000002b9b7e5 in CGameSvr::MainLoop (this=0x605c6c0, sConfFile=0x7fff2960d9a0 "/data/home/user00/sgame_
#  0x0000000002b9d4ba in main (argc=1, argv=0x7fff2960eef8) at gamesvr.cpp:2558
(db) up
#  0x000000000280b33d in __gnu_cxx::hash_map<int, HashMap<int, ResWrapper<ResData::ResMangoUserGroup>::ConfMap,
ngoUserGroup>::ConfMap, __gnu_cxx::hash<int>, std::equal_to<int> > > >::operator[] (this=0x623f2f8, __key=@0x7
7           { return _M_ht.find_or_insert(value_type(__key, _Tp())).second; }
(db)
#  0x0000000002805e36 in ResWrapper<ResData::ResMangoUserGroup>::PostInit (this=0x623f2d0, iLanguage=566670424)
6                   ConfMap& roConfMap = m_oLanguageConfMaps[iLanguage][i];
(db)
#  0x00000000022f4e37 in CGameLogicMgr::_DoMsgLogicDispatch (this=0x62459e0, poAcnt=0x21c6b458, dwMsgID=1616, s
09          poIGameLogic->HandleMsg(poAcnt, dwMsgID, sUnpackedPkgBody);
(db) p poIGameLogic
$1 = (CGameLogic *) 0x623f2d0
(db)
```

图2. coredump对应的堆栈

2. 排查过程

2.1 为什么会core在1309行?

要判断这个问题，需要借助于`CGameLogicMgr::_DoMsgLogicDispatch`的汇编代码（gdb执行`disas CGameLogicMgr::_DoMsgLogicDispatch`），因为通过汇编代码，能更清楚的知道程序在发生coredump时正在执行的指令。部分汇编代码如下：

```
   0x00000000022f4d4c <+1238>:  callq  0x22fcb26 <__gnu_cxx::_Hashtable_iterator<std::pair<unsigned int const, CGameLogic*>, un
   0x00000000022f4d51 <+1243>:  mov    0x8(%rax),%rax
   0x00000000022f4d55 <+1247>:  mov    %rax,-0x18(%rbp)
   0x00000000022f4d59 <+1251>:  cmpq   $0x0,-0x18(%rbp)
   0x00000000022f4d5e <+1256>:  jne    0x22f4e11 <CGameLogicMgr::_DoMsgLogicDispatch(CGameAccount*, uint32_t, char*)+1435>
... ... ... ...
   0x00000000022f4e11 <+1435>:  mov    -0x18(%rbp),%rax
   0x00000000022f4e15 <+1439>:  mov    (%rax),%rax
   0x00000000022f4e18 <+1442>:  add    $0x40,%rax
   0x00000000022f4e1c <+1446>:  mov    (%rax),%r8
   0x00000000022f4e1f <+1449>:  mov    -0x64(%rbp),%edx
   0x00000000022f4e22 <+1452>:  mov    -0x70(%rbp),%rcx
   0x00000000022f4e26 <+1456>:  mov    -0x60(%rbp),%rbx
   0x00000000022f4e2a <+1460>:  mov    -0x18(%rbp),%rax
   0x00000000022f4e2e <+1464>:  mov    %rbx,%rsi
   0x00000000022f4e31 <+1467>:  mov    %rax,%rdi
   0x00000000022f4e34 <+1470>:  callq  *%r8
=> 0x00000000022f4e37 <+1473>:  mov    $0x1,%eax
   0x00000000022f4e3c <+1478>:  sub    $0xffffffffffffff80,%rsp
   0x00000000022f4e40 <+1482>:  pop    %rbx
   0x00000000022f4e41 <+1483>:  pop    %r12
   0x00000000022f4e43 <+1485>:  leaveq
   0x00000000022f4e44 <+1486>:  retq
```

这里需要了解的是符号`⇒`表示下一条要执行的指令（也就是说此时`PC`寄存器的值为`0x00000000022f4e37`），所以coredump实际上发生在了`0x00000000022f4e34 <+1470>:  callq  *%r8`这条指令，对应c++代码`poIGameLogic->HandleMsg(poAcnt, dwMsgID, sUnpackedPkgBody)`。这条指令表示跳转到另外一条指令，且目标指令地址保存在r8寄存器。
我们将`r8`寄存器内的值打印出来为`0x000001f0`，可以发现其并不是一个合法的内存地址（64位系统合法的内存地址参考图3），所以这里可以知道为什么进程core在了1309行，因为`callq`调用的指令地址是个非法地址。

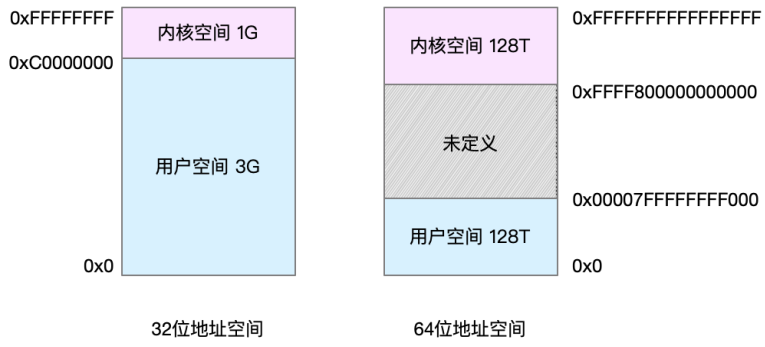

图3. 地址空间

2.2 为什么多次coredump产生的函数堆栈不一样?

将多次coredump对应的r8寄存器内的值打印出来可以发现，r8寄存器的值是不一样的，因此gdb根据寄存器值推导出的函数也就不一样了。

2.3 新的问题

第1节中提出的两个问题都指向了一个原因，即r8寄存器的值是非法的。但是从汇编代码可以看到r8的值是经过一系列计算得出的，并不是导致coredump的根本原因，所以只有找到这个非法值的来源，才能够定位这次coredump产生的根本原因。

```
   //r8寄存器值的来源
   0x00000000022f4e11 <+1435>:  mov    -0x18(%rbp),%rax
   0x00000000022f4e15 <+1439>:  mov    (%rax),%rax
   0x00000000022f4e18 <+1442>:  add    $0x40,%rax
   0x00000000022f4e1c <+1446>:  mov    (%rax),%r8
```

3. 继续定位

一时间也没有任何头绪，所以只能做各种尝试及验证。

3.1 模拟r8值的计算过程

根据汇编代码，在gdb中执行 +1435 到 +1446 这4条汇编指令。

a. `0x00000000022f4e11 <+1435>: mov -0x18(%rbp),%rax`

这条指令是通过寄存器rbp计算得到一个新的地址，并存入寄存器rax。通过print命令打印rax的值：

p /x $rax

$rax=0×623f2d0

b. `0x00000000022f4e15 <+1439>: mov (%rax),%rax`

这条指令是访问rax所指向的内存，并把内存中的值再次存入rax。所以通过`x`命令访问rax指向的内存：

x $rax

($rax)=0×04ade5da

c. `0x00000000022f4e18 <+1442>: add $0x40,%rax`

这条指令是将b步骤得出的值加0×40后再次存入rax。计算如下：

p /x 0×04ade5da+0×40

($rax)+0×40=0×4ade61a

d. `0x00000000022f4e1c <+1446>: mov (%rax),%r8`

这条指令是访问c步骤得出的值所指向的内存，然后存入r8。计算如下：

x 0×4ade61a

(0×4ade61a)=0×4ade61a <_ZTV16CGameLogicReward+90>: 0×000001f0

值得注意的是

a. `poIGameLogic`的值为0×623f2d0，与+1435指令执行后rax的值一致

b. `HandleMsg`为虚函数，指令a-b-c-d容易联想到虚函数表相关的内容

c. 出现了两个新的地址`0x04ade5da`和`0x4ade61a`，觉得跟函数调用有关系

3.2 x命令访问各种地址

x命令在本次定位问题的过程中起到了很大的作用，该命令主要用来访问地址所指向的内存，并且可以指定访问的次数，访问的格式及访问的单位。参考：x command

这里通过x命令，访问3.1中出现的几个地址，期待能有新的发现。

3.2.1 访问 0x623f7d0，命令 `x /40ag 0x623f7d0`

```
(gdb) x /40ag 0x623f2d0
0x623f2d0:      0x4ade5d1 <_ZTV16CGameLogicReward+17>    0x4a68270 <_ZTV14CGameLogicMail+16>
0x623f2e0:      0x4a09d50 <_ZTV16CGameLogicLegend+16>    0x5eca45a05ed428e0
0x623f2f0:      0x5ecb9720      0x47fa250 <_ZTV16CGameLogicCredit+16>
```

图4 访问0×623f7d0

图4的输出结果看出红框的内容`_ZTV16CGameLogicReward`的值与其他类有差异，但是此时还不敢确定这个值是错误的，甚至也不确定这个值的含义。但是因为`CGameLogicReward`为单例static对象，所以可以确定对于同一个可执行文件，每个单例对象所在的内存地址是固定的，例如`CGameLogicReward`对象的地址固定为`0x623f7d0`。

所以在另一个环境GarenaTestModifytime（可执行文件一样），访问`0x623f7d0`作为对比。

3.2.2 GarenaTestModifytime 访问 0x623f7d0

```
(gdb) x /30ag 0x623f7d0-0x80
0x623f750:      0x1     0x0
0x623f760:      0x4c48d10 <_ZTV14CGameLogicWeal+16>      0x5ffb63c8
0x623f770:      0x0     0x0
0x623f780:      0x0     0x623f778
0x623f790:      0x623f778       0x0
0x623f7a0:      0x0     0x0
0x623f7b0:      0x0     0x623f7a8
0x623f7c0:      0x623f7a8       0x0
0x623f7d0:      0x4ade5f0 <_ZTV16CGameLogicReward+16>   0x4a68290 <_ZTV14CGameLogicMail+16>
0x623f7e0:      0x4a09d70 <_ZTV16CGameLogicLegend+16>   0x5ff195a05ffb78e0
0x623f7f0:      0x5ff2e720      0x47fa270 <_ZTV16CGameLogicCredit+16>
0x623f800:      0x4647e90 <_ZTV17CGameLogicAccount+16>  0x0
0x623f810:      0x0     0x0
```

图5 GarenaTestModifytime 访问 0×623f7d0

图5的输出结果看出`_ZTV16CGameLogicReward`的值也为16，并且因为其他类的值也为16，所以觉得16应该是正常的值，而GarenaTest的这个值被恶意修改了。

3.2.3 访问 `0x04ade5da`

```
0x4ade5dd:      0x0     0x4adf4a0000000
0x4ade5ed <_ZTV16CGameLogicReward+13>:  0x27dc4f4000000 0x1f00cf2000000
0x4ade5fd <_ZTV16CGameLogicReward+29>:  0x22de8f2000000 0x1f00d16000000
0x4ade60d <_ZTV16CGameLogicReward+45>:  0x1f77a76000000 0x1f77abc000000
0x4ade61d <_ZTV16CGameLogicReward+61>:  0x1f00c6c000000 0x27dc25e000000
0x4ade62d <_ZTV16CGameLogicReward+77>:  0x27dc27c000000 0x1f00cb4000000
0x4ade63d <_ZTV16CGameLogicReward+93>:  0x0     0x4aded80000000
0x4ade64d <_ZTV10ResWrapperIN7ResData17ResMangoUserGroupEE+13>: 0x27f9bb2000000 0x27f9c4e000000
0x4ade65d <_ZTV10ResWrapperIN7ResData17ResMangoUserGroupEE+29>: 0x2805448000000 0x2805c8a000000
0x4ade66d <_ZTV10ResWrapperIN7ResData17ResMangoUserGroupEE+45>: 0x280635e000000 0x0
0x4ade67d:      0x0     0x4adee00000000
0x4ade68d <_ZTVN10ResWrapperIN7ResData17ResMangoUserGroupEE7ConfMapE+13>:       0x28034ba000000 0x28034f4000000
0x4ade69d <_ZTVN10ResWrapperIN7ResData17ResMangoUserGroupEE7ConfMapE+29>:       0x2806370000000 0x2804bea000000
0x4ade6ad <_ZTVN10ResWrapperIN7ResData17ResMangoUserGroupEE7ConfMapE+45>:       0x2806eee000000 0x2806f62000000
0x4ade6bd <_ZTVN10ResWrapperIN7ResData17ResMangoUserGroupEE7ConfMapE+61>:       0x28072b2000000 0x28072c2000000
0x4ade6cd <_ZTVN10ResWrapperIN7ResData17ResMangoUserGroupEE7ConfMapE+77>:       0x28072d2000000 0x28072f6000000
0x4ade6dd <_ZTVN10ResWrapperIN7ResData17ResMangoUserGroupEE7ConfMapE+93>:       0x2807398000000 0x0
0x4ade6ed:      0x0     0x0
```

图6 访问0×04ade5da

继续访问3.1出现的地址`0x04ade5da`，出现了ResMangoUserGroup，似乎跟图2的堆栈也联系上了。

3.2.4 访问 `0x04ade5da` 前后几个字节的地址

```
(gdb) x /40ag 0x4ade5d0
0x4ade5d0 <_ZTV16CGameLogicReward+16>:  0x27dc4f0
    <CGameLogicReward::HandleTcapResp(CTCaplusMgr&, TCaplusDBDef::TB_USERBUFFER*, uint64_t, uint32_t, TIME_VAL const&, uint32_t, int32_t)>     0x1f00cf2
    <CGameLogic::HandleTcapRespForDispatcher(CTCaplusMgr&, TCaplusDBDef::TB_USERBUFFER*, uint64_t, uint32_t, TIME_VAL const&, uint32_t, int32_t)>
0x4ade5e0 <_ZTV16CGameLogicReward+32>:  0x22de8f2
    <CGameLogic::HandleTcapTimeout(CTCaplusMgr&, TCaplusDBDef::TB_USERBUFFER*, char const*, TcaplusService::TCaplusApiCmds, uint64_t, uint32_t, TIME_VAL const&, uint32_t)>    0x1f00d16 <CG
ameLogic::HandleTcapTimeoutForDispatcher(CTCaplusMgr&, TCaplusDBDef::TB_USERBUFFER*, char const*, TcaplusService::TCaplusApiCmds, uint64_t, uint32_t, TIME_VAL const&, uint32_t)>
0x4ade5f0 <_ZTV16CGameLogicReward+48>:  0x1f77a76 <CGameLogicReward::~CGameLogicReward()>       0x1f77abc <CGameLogicReward::~CGameLogicReward()>
0x4ade600 <_ZTV16CGameLogicReward+64>:  0x1f00c6c <CGameLogic::InitLogic()>     0x27dc25a <CGameLogicReward::GetHandleMsgIDs(int32_t&)>
0x4ade610 <_ZTV16CGameLogicReward+80>:  0x27dc278 <CGameLogicReward::HandleMsg(CGameAccount*, uint32_t, char*)> 0x1f00cb4 <CGameLogic::HandleMsgForDispatcher(CGameAccount*, uint32_t, char*)
>
0x4ade620 <_ZTV10ResWrapperIN7ResData17ResMangoUserGroupEE>:    0x0     0x4aded60 <_ZTI10ResWrapperIN7ResData17ResMangoUserGroupEE>
```

图7 访问0×04ade5d0

因为图6输出的很多都是+13，+29，+45这类没对齐的字节，所以觉得如果调整访问`0x04ade5da`这段内存前后几个字节的内容并且实现字节对齐后，可能会有其他输出。多次尝试后，输出了图7的结果，`CGameLogicReward`所有虚函数出现在眼前。

3.3 初步结论

对前面出现的几个地址做一下总结：
a. `0x623f7d0` 为类`CGameLogicReward`单例对象所在的内存地址
b. `0x04ade5d0`为类`CGameLogicReward`虚表的地址
c. `0x04ade5da`为被修改的虚表地址，该错误地址导致无法正常解析出虚函数地址，从而导致进程coredump

4. 谁修改了虚函数表的地址

至此，我们离真相更近一步了，就是找到谁修改了`CGameLogicReward`的虚表地址。c++的对象模型告诉我们，虚表地址是放在对象的起始空间内，所以如果虚表地址被修改，存在2种可能：
a. 有代码逻辑拿到了单例对象，然后执行了修改操作。可能类似`CGameLogicReward& roGameLogicReward = CGameLogicReward::Instance(); roGameLogicReward++;`这种代码
b. 与单例对象相邻的其它对象写越界了，波及到了本次出问题的`CGameLogicReward`

4.1 可能a，拿到单例对象，然后执行修改

因为相关代码量巨大，并且在Logic对象上直接做算术运算可能极小，所以先跳过了这种可能。

4.2 可能b，相邻对象写越界

首先需要找到与`CGameLogicReward`单例对象相邻的对象是谁。需要再次强调的是，`CGameLogicReward`是`static`对象，所以该对象应该存在于进程的`BSS(Block Started by Symbol)`段，所以我们只需要扫描`gamesvr`二进制文件的符号表，即可知道与`CGameLogicReward`相邻的对象是谁。命令为`nm --numeric-sort -C gamesvr_core_20200527 | grep GameLogicReward | grep instance`，结果如图8。可以看到相邻的对象为`CGameLogicWeal`和`CGameLogicMail`，因为通过图5可以看到`CGameLogicWeal`和`CGameLogicMail`的虚表地址是正常的，所以怀疑是他们写坏了`CGameLogicReward`的虚表地址（因为如果不是他们，那么他们当中肯定也有被写坏的），并且因为`CGameLogicWeal`在低地址，所以为重点怀疑对象。

```
[user00@Garena_Test@Idc ~/sgame/zone/game/gamesvr]$ nm --numeric-sort -C gamesvr_core_20200527 | grep GameLogicReward | grep instance
000000000623f248 u guard variable for Singleton<CGameLogicReward>::Instance()::s_instance
000000000623f2d0 u Singleton<CGameLogicReward>::Instance()::s_instance
00000000140dd568 u guard variable for Singleton<CGameLogicRewardMatch>::Instance()::s_instance
00000000140ddbc0 u Singleton<CGameLogicRewardMatch>::Instance()::s_instance
[user00@Garena_Test@Idc ~/sgame/zone/game/gamesvr]$ nm --numeric-sort -C gamesvr_core_20200527 | grep 000000000623f2d0 -B5 -A5
000000000623f238 u guard variable for Singleton<CGameLogicCredit>::Instance()::s_instance
000000000623f240 u guard variable for Singleton<CGameLogicMail>::Instance()::s_instance
000000000623f248 u guard variable for Singleton<CGameLogicReward>::Instance()::s_instance
000000000623f250 u guard variable for Singleton<CGameLogicWeal>::Instance()::s_instance
000000000623f260 u Singleton<CGameLogicWeal>::Instance()::s_instance
000000000623f2d0 u Singleton<CGameLogicReward>::Instance()::s_instance
000000000623f2d8 u Singleton<CGameLogicMail>::Instance()::s_instance
000000000623f2e0 u Singleton<CGameLogicLegend>::Instance()::s_instance
000000000623f2f8 u Singleton<CGameLogicCredit>::Instance()::s_instance
000000000623f300 u Singleton<CGameLogicAccount>::Instance()::s_instance
000000000623f320 u Singleton<CGameLogicValorPassTask>::Instance()::s_instance
```

图8 CGameLogicReward相邻对象

但是通过阅读代码还是很难定位问题，而且也不敢确定就是`CGameLogicWeal`导致的问题，所以也没有花很多时间在此。

4.3 gdb断点

通过前面一系列分析，可以得出结论，本次coredump的原因是`CGameLogicReward`的虚表地址被错误的从`0x04ade5d0`修改为了`0x04ade5da`或者其他值。

根据c++的对象模型可知，虚表地址正是存储在对象所在内存的首位，也就是本案例中地址`0x623f7d0`所指向的内存空间。

所以，我们只需要通过gdb监听`0x623f7d0`指向内存的修改行为，然后等待断点触发即可，如图9。

```
(gdb) watch *0x623f2d0
Hardware watchpoint 1: *0x623f2d0
(gdb) c
Continuing.
```

图9 监听内存修改

5. 破案

28号早上，断点打上一段时间后，被触发了。如图10

通过现场得知，问题果然出在`CGameLogicWeal`，该类定义了2个map用来记录活动的个数，然后在计数的函数内代码出现了bug，如图11。代码在iter为end()时依然执行了++操作。

```
(gdb) c
Continuing.



Hardware watchpoint 1: *0x623f2d0

Old value = 78505424
New value = 78505425
CGameLogicWeal::StatAcntWealNum (this=0x623f260, roAcnt=...) at weal/gamelogicweal.cpp:612
612     weal/gamelogicweal.cpp: 没有那个文件或目录.
        in weal/gamelogicweal.cpp
```

图10 断点被触发

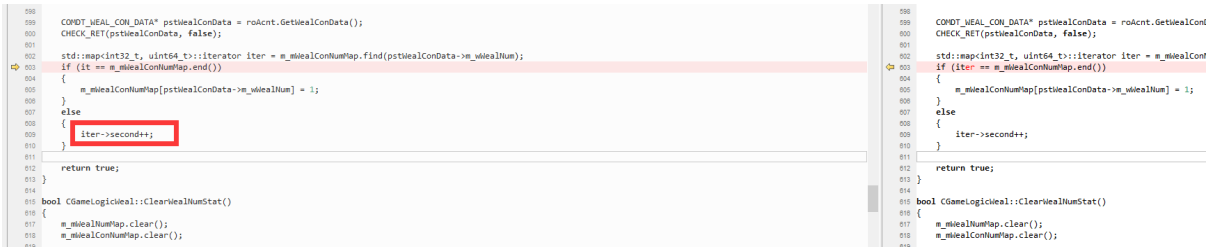

图11 导致问题的代码

6. 问题整理

导致此次coredump的根本原因是`CGameLogicWeal`的逻辑问题，修改了相邻的`CGameLogicReward`的虚表指针，从而导致虚函数调用时无法正常解析出虚函数地址，最终触发coredump。再次梳理后得到整个场景下的内存布局，如图12。

> a. BSS段的`Singleton<CGameLogicReward>::Instance()::s_instance`和`Singleton<CGameLogicWeal>::Instance()::s_instance`地址相邻，其中`CGameLogicReward`的地址为`0x623f7d0`，第一个成员为虚表地址，指向内存`0x04ade5d0`
> b. 虚函数表及虚函数位于`Text Segment`，其中虚函数表的地址为`0x04ade5d0`，其数组元素指向类的虚函数
> c. `CGameLogicWeal`在迭代器为`end`时依然执行`++`操作，导致`CGameLogicReward`虚表地址被修改
> d. 错误的虚表地址，解析到了错误的虚函数地址，导致coredump

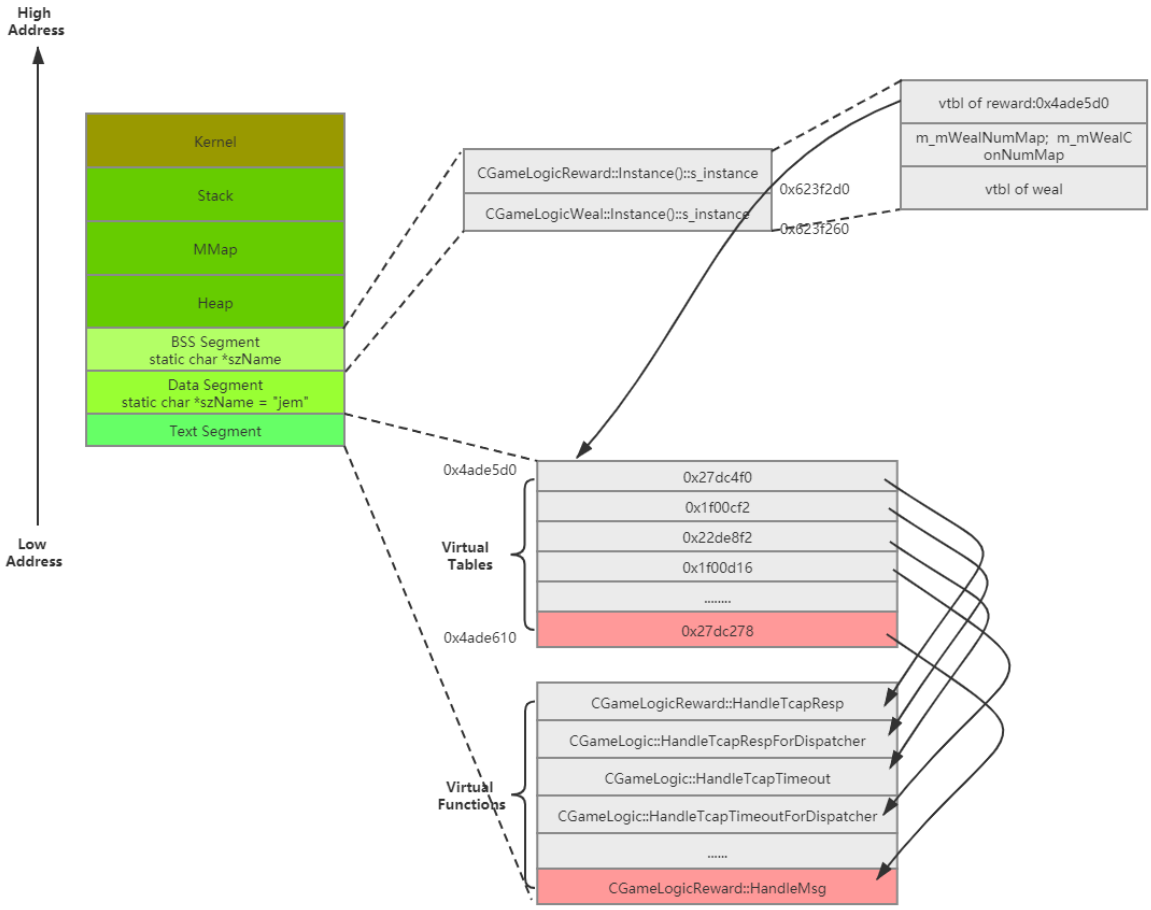
High Address
Low Address
Kernel
Stack
MMap
Heap
BSS Segment
static char *szName
Data Segment
static char *szName = "jem"
Text Segment
CGameLogicReward::Instance()::s_instance
CGameLogicWeal::Instance()::s_instance
0x623f2d0
0x623f260
vtbl of reward:0x4ade5d0
m_mWealNumMap; m_mWealConNumMap
vtbl of weal
0x4ade5d0
Virtual Tables
0x4ade610
0x27dc4f0
0x1f00cf2
0x22de8f2
0x1f00d16
.....
0x27dc278
Virtual Functions
CGameLogicReward::HandleTcapResp
CGameLogic::HandleTcapRespForDispatcher
CGameLogic::HandleTcapTimeout
CGameLogic::HandleTcapTimeoutForDispatcher
.....
CGameLogicReward::HandleMsg

图12 内存概要