方亮的专栏

[原]DllMain中不当操作导致死锁问题的分析--导致DllMain中死锁的关键隐藏因子

2012-11-5 阅读2732 评论6

有了前面两节的基础，我们现在切入正题：研究下DllMain为什么会因为不当操作导致死锁的问题。首先我们看一段比较经典的“DllMain中死锁”代码。（转载请指明出于breaksoftware的csdn博客）

```
//主线程中
HMODULE h = LoadLibraryA(strDllName.c_str());
```

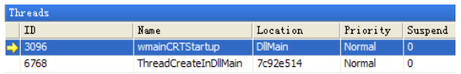
```
// DLL中代码
static DWORD WINAPI ThreadCreateInDllMain(LPVOID) {
    return 0;
}

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                        )
{
    DWORD tid = GetCurrentThreadId();
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH: {
            printf("DLL DllWithoutDisableThreadLibraryCalls_A:\tProcess attach (tid = %d)\n", tid);
            HANDLE hThread = CreateThread(NULL, 0, ThreadCreateInDllMain, NULL, 0, NULL);
            WaitForSingleObject(hThread, INFINITE);
            CloseHandle(hThread);
        }break;
    case DLL_PROCESS_DETACH:
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
        break;
    }
    return TRUE;
}
```

简要说下DLL中逻辑：设计该段代码的同学希望在DLL第一次被映射到进程内存空间时，创建一个工作线程，该工作线程内容可能很简单。为了尽可能简单，我们让这个工作线程直接返回0。这样从逻辑和效率上看，都不会因为我们的工作线程写的有问题而导致死锁。然后我们在DllMain中等待这个线程结束才从返回。

粗略看这个问题，我们很难看出这个逻辑会导致死锁。但是事实就是这样发生了。我们跑一下程序，发现程序输出一下结果

后就停住了，光标在闪动，貌似还是在等待我们输入。可是我们怎么敲击键盘都没有用：它死锁了。

我是在VS2005中调试该程序，于是我们可以Debug->Break All来冻结所有线程。



我们先查看主线程(3096)的堆栈

Name	Language
ntdll.dll!_KiFastSystemCallRet@0()	
ntdll.dll!_NtWaitForSingleObject@12() + 0xc bytes	
kernel32.dll!_WaitForSingleObjectEx@12() + 0x8b bytes	
kernel32.dll!_WaitForSingleObject@8() + 0x12 bytes	
DllWithoutDisableThreadLibraryCalls_A.dll!DllMain(HINSTANCE__ * hModule=0x10000000, unsigned long ul_reason_for_call=1, void * lpReserved=0x00000000) Line 27 + 0xe bytes	C++
DllWithoutDisableThreadLibraryCalls_A.dll!__DllMainCRTStartup(void * hDllHandle=0x10000000, unsigned long dwReason=1, void * lpreserved=0x00000000) Line 498 + 0x11 bytes	C
DllWithoutDisableThreadLibraryCalls_A.dll!_DllMainCRTStartup(void * hDllHandle=0x10000000, unsigned long dwReason=1, void * lpreserved=0x00000000) Line 462 + 0x11 bytes	C
ntdll.dll!_LdrpCallInitRoutine@16() + 0x14 bytes	
ntdll.dll!_LdrpRunInitializeRoutines@4() + 0x205 bytes	
ntdll.dll!_LdrpLoadDll@24() - 0x1b6 bytes	
ntdll.dll!_LdrLoadDll@16() + 0x110 bytes	
kernel32.dll!_LoadLibraryExW@12() + 0xc8 bytes	
kernel32.dll!_LoadLibraryExA@12() + 0x1f bytes	
kernel32.dll!_LoadLibraryA@4() + 0x2d bytes	
DllMainSerial.exe!wmain(int argc=3, wchar_t * * argv=0x003b7000) Line 50 + 0x1b bytes	C++
DllMainSerial.exe!__tmainCRTStartup() Line 594 + 0x19 bytes	C
DllMainSerial.exe!wmainCRTStartup() Line 414	C
kernel32.dll!_BaseProcessStart@4() + 0x23 bytes	

堆栈不长，我全部列出来

17	**ntdll.dll!_KiFastSystemCallRet@0()**
16	**ntdll.dll!_NtWaitForSingleObject@12()**
15	kernel32.dll!_WaitForSingleObjectEx@12()
14	kernel32.dll!_WaitForSingleObject@8()
13	DllWithoutDisableThreadLibraryCalls_A.dll!DllMain(HINSTANCE__ * hModule=0x10000000, unsigned long ul_reason_for_call=1, void * lpReserved=0x00000000)
12	DllWithoutDisableThreadLibraryCalls_A.dll!__DllMainCRTStartup(void * hDllHandle=0x10000000, unsigned long dwReason=1, void * lpreserved=0x00000000)
11	DllWithoutDisableThreadLibraryCalls_A.dll!_DllMainCRTStartup(void * hDllHandle=0x10000000, unsigned long dwReason=1, void * lpreserved=0x00000000)
10	ntdll.dll!_LdrpCallInitRoutine@16()
9	ntdll.dll!_LdrpRunInitializeRoutines@4()
8	ntdll.dll!_LdrpLoadDll@24()
7	ntdll.dll!_LdrLoadDll@16()
6	kernel32.dll!_LoadLibraryExW@12()

5	kernel32.dll!_LoadLibraryExA@12()
4	kernel32.dll!_LoadLibraryA@4()
3	DllMainSerial.exe!wmain(int argc=3, wchar_t * * argv=0x003b7000)
2	DllMainSerial.exe!__tmainCRTStartup()
1	DllMainSerial.exe!wmainCRTStartup()
0	kernel32.dll!_BaseProcessStart@4()

我们看下这个堆栈。大致我们可以将我们程序分为4段：

0 启动启动我们程序

1~6 我们加载Dll。

7~10 系统为我们准备DLL的加载。

11~17 DLL内部代码执行。

我们关注一下14~17这段对WaitForSingleObject的调用逻辑。15、16步这个过程显示了Kernel32中的WaitForSingleObjectEx在底层是调用了NtDll中的NtWaitForSingleObject。在NtWaitForSingleObject内部，即17步，我们看到的“_KiFastSystemCallRet@0”。这儿要说明下，这个并不是意味着我们程序执行到这个函数。我们看下这个函数的代码

```
_KiFastSystemCall@0:
7C92E510 8B D4           mov         edx, esp
7C92E512 0F 34           sysenter
_KiFastSystemCallRet@0:
7C92E514 C3              ret
7C92E515 8D A4 24 00 00 00 00 lea         esp, [esp]
7C92E51C 8D 64 24 00     lea         esp, [esp]
```

KiFastSystemCallRet函数是内核态(Ring0层)逻辑回到用户态(Ring3层)的着陆点。与之相对应的KiFastSystemCall函数是用户态进入内核态必要的调用方法。因为内核态代码我们是无法查看的，所以动态断点只能设置到KiFastSystemCallRet开始处。所以实际死锁是因为NtWaitForSingleObject在底层调用了KiFastSystemCall进入内核，在内核态中死锁的。

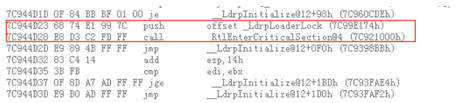
我们在《DllMain中不当操作导致死锁问题的分析--死锁介绍》中介绍过，死锁存在的条件是相互等待。主线程中，我们发现其等待的是工作线程结束。那么工作线程在等待主线程什么呢？我们看下工作线程的调用堆栈

Name	Language
ntdll.dll!_KiFastSystemCallRet@0()	
ntdll.dll!_NtWaitForSingleObject@12() + 0xc bytes	
ntdll.dll!_RtlpWaitForCriticalSection@4() + 0x8c bytes	
ntdll.dll!_RtlEnterCriticalSection@4() + 0x46 bytes	
ntdll.dll!__LdrpInitialize@12() + 0xb4bf bytes	
ntdll.dll!_KiUserApcDispatcher@20() + 0x7 bytes	
ntdll.dll!_RtlAllocateHeap@12() + 0x9b48 bytes	

我们对这个堆栈进行编号

6	**ntdll.dll!_KiFastSystemCallRet@0()**
5	**ntdll.dll!_NtWaitForSingleObject@12() + 0xc bytes**
4	ntdll.dll!_RtlpWaitForCriticalSection@4() + 0x8c bytes
3	ntdll.dll!_RtlEnterCriticalSection@4() + 0x46 bytes
2	ntdll.dll!__LdrpInitialize@12() + 0xb4bf bytes
1	ntdll.dll!_KiUserApcDispatcher@20() + 0x7 bytes
0	ntdll.dll!_RtlAllocateHeap@12() + 0x9b48 bytes

我们看到倒数两步（5、6）和主线程中最后两步（16、17）是相同的，即工作线程也是在进入内核态后死锁的。我们知道主线程在等工作线程结束，那么工作线程在等什么呢？我们追溯栈，请关注“ntdll.dll!__LdrpInitialize@12() + 0xb4bf bytes”处的代码



我们看到，是因为_RtlEnterCriticalSection在底层调用了NtWaitForSingleObject。那么我们关注下_RtlEnterCriticalSection的参数_LdrpLoaderLock，它是什么？我们借助下IDA查看下LdrpInitialize反编译代码

```
……
v4 = *(_DWORD *)(*MK_FP(__FS__, 0x18) + 0x30);
v3 = *MK_FP(__FS__,0x18);
 ……
  *(_DWORD *)(v4 + 0xa0) = &LdrpLoaderLock;
  if ( !(unsigned __int8)RtlTryEnterCriticalSection(&LdrpLoaderLock) )
  {
  ……
    RtlEnterCriticalSection(&LdrpLoaderLock);
  }
  ……
  if ( *( DWORD *)(v4 + 0xc) )
  {
    ……
    LdrpInitializeThread(a1);
  }
  else
  {
……
    v17 = LdrpInitializeProcess(a1, a2, &v11, v14, v15);
……
  }
……
```

由RtlTryEnterCriticalSection 可知LdrpLoaderLock是_RTL_CRITICAL_SECTION类型。在尝试进入临界区之前，LdrpLoaderLock将被保存到某个结构体变量v4的某个字段（偏移0xA0）中。那么v4是什么类型呢？这儿可能要科普下windows x86操作系统的一些知识：

在windows系统中每个用户态线程都有一个记录其执行环境的结构体TEB(Thread Environment Block)。TEB结构体中第一个字段是一个TIB（ThreadInformation Block）结构体，该结构体中保存着异常登记链表等信息。在x86系统中，段寄存器FS总是指向TEB结构。于是FS:[0]指向TEB起始字段，也就是指向TIB结构体。我们用Windbg查看下TEB的结构体,该结构体很大，我只列出我们目前关心的字段

```
lkd> dt _TEB
nt!_TEB
```

```
   +0x000 NtTib             : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId          : _CLIENT_ID
.....
```

NtTib就是TIB结构体对象名。 我们再看下TIB结构体

```
lkd> dt _NT_TIB
nt!_NT_TIB
   +0x000 ExceptionList     : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 StackBase         : Ptr32 Void
   +0x008 StackLimit        : Ptr32 Void
   +0x00c SubSystemTib      : Ptr32 Void
   +0x010 FiberData         : Ptr32 Void
   +0x010 Version           : Uint4B
   +0x014 ArbitraryUserPointer : Ptr32 Void
   +0x018 Self              : Ptr32 _NT_TIB
```

该结构体其他字段不解释，我们只看最后一个字段(FS:[18])指向_NT_TIB结构体的指针Self。正如其名，该字段指向的是TIB结构体在进程空间中的虚拟地址。为什么要指向自己？那我们是否可以直接使用FS:[0]地址？不可以。举个例子：我用windbg挂载到我电脑上一个运行中的calc（计算器）。我们查看fs:[0]指向空间保存的值，7ffdb000是TIB的Self字段。

```
0:002> dd fs:[0]
0038:00000000  00afffe4 00b00000 00aff000 00000000
0038:00000010  00001e00 00000000 7ffdb000 00000000
0038:00000020  00001220 00001ba0 00000000 00000000
0038:00000030  7ffde000 00000000 00000000 00000000
0038:00000040  00000000 00000000 00000000 00000000
0038:00000050  00000000 00000000 00000000 00000000
0038:00000060  00000000 00000000 00000000 00000000
0038:00000070  00000000 00000000 00000000 00000000
```

我们查看TIB结构体去匹配该地址指向的空间的。

```
0:002> dt ntdll!_NT_TIB 7ffdb000
   +0x000 ExceptionList     : 0x00afffe4 _EXCEPTION_REGISTRATION_RECORD
   +0x004 StackBase         : 0x00b00000
   +0x008 StackLimit        : 0x00aff000
   +0x00c SubSystemTib      : (null)
   +0x010 FiberData         : 0x00001e00
   +0x010 Version           : 0x1e00
   +0x014 ArbitraryUserPointer : (null)
   +0x018 Self              : 0x7ffdb000 _NT_TIB
```

可以看到7ffdb000所指向的空间的各字段的值和FS:[0]指向的空间的值一致。但是如果我们这样输入就会失败

```
0:002> dt ntdll!_NT_TIB fs:[0]
Cannot find specified field members.
```

介绍完这些后，我们再回到IDA反汇编的代码中。v4 = *(_DWORD*)(*MK_FP(__FS__, 0x18) + 0x30);这段中MK_FP不是一个函数，是一个宏。它的作用是在基址上加上偏移得出一个地址。于是MK_FP(__FS__, 0x18)就是FS:[0x18]，即TIB的Self字段。在该地址再加上0x30得到的地址已经超过了TIB空间，于是我们继续查看TEB结构体

```
lkd> dt _TEB
nt!_TEB
   +0x000 NtTib             : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId          : _CLIENT_ID
   +0x028 ActiveRpcHandle   : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
```

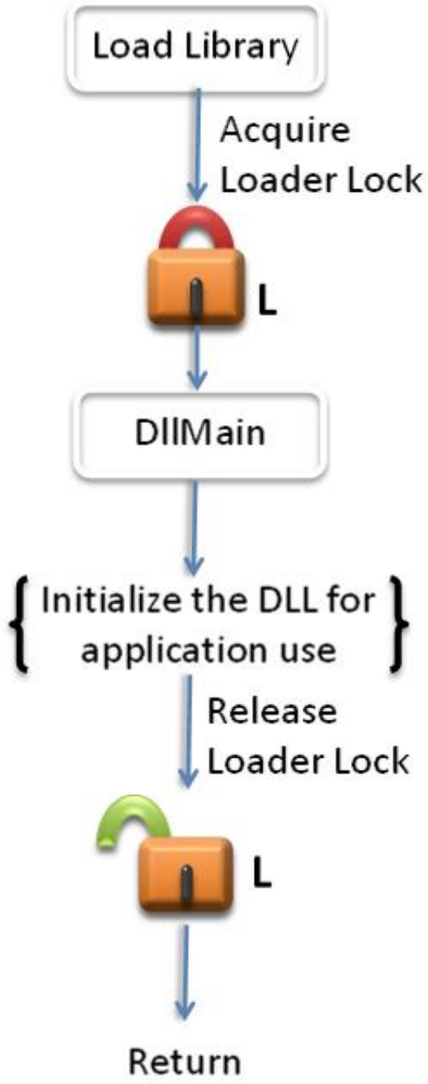
发现0x30偏移的是PEB(Process Environment Block)。

```
lkd> dt _PEB
nt!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
……
+0x09c GdiDCAttributeList : Uint4B
   +0x0a0 LoaderLock       : Ptr32 Void
   +0x0a4 OSMajorVersion   : Uint4B
```

可以发现该结构体偏移0xa0处是一个名字为LoaderLock的变量。

《windows核心编程》中有关于DllMain序列化执行的讲解，大致意思是：线程在调用DllMain之前，要先获取锁，等DllMain执行完再解开这个锁。这样不同线程加载DLL就可以实现序列化操作。而在微软官方文档《Best Practices for Creating DLLs》中也有对这个说法的佐证

```
The DllMain entry-point function. This function is called by the loader when it loads or unloads a DLL. The loader serializes calls to DllMain so
```



其中还有段关于这个锁的介绍

```
The loader lock. This is a process-wide synchronization primitive that the loader uses to ensure serialized loading of DLLs. Any function that mu
```

在该文中多处对这个锁的说明值暗示这个锁是PEB中的LoaderLock。

那么刚才为什么要*(_DWORD *)(v4 + 0xa0) = &LdrpLoaderLock;?因为该LdrpLoaderLock是进程内共享的变量。这样每个线程在执行初期，会先进入该临界区，从而实现在进程内DllMain的执行是序列化的。于是我们得出以下结论:

进程内所有线程共用了同一个临界区来序列化**DllMain**的执行。

结合《DllMain中不当操作导致死锁问题的分析--进程对DllMain函数的调用规律的研究和分析》中介绍的规律

二 线程创建后会调用已经加载了的**DLL**的**DllMain**，且调用原因是**DLL_THREAD_ATTACH**。

我们发现

```
HANDLE hThread = CreateThread(NULL, 0, ThreadCreateInDllMain, NULL, 0, NULL);
WaitForSingleObject(hThread, INFINITE);
```

主线程进入临界区去调用DllMain时进入了临界区，而工作线程也要进入临界区去执行DllMain。但是此时临界区被主线程占用，工作线程便进入等待状态。而主线程却等待工作线程退出才退出临界区。于是这就是死锁产生的原因。

上一篇 下一篇

发表评论

提 交

查看评论

6楼 Breaksoftware 2013-11-09 23:43
[reply]zt_cau[/reply] 工具-选项-调试-符号，勾选Microsoft符号服务器。我在VS2010里这么设置的。

5楼 zt_cau 2013-11-09 20:38
[reply]Breaksoftware[/reply] vs2008

4楼 Breaksoftware 2013-10-14 00:36
[reply]zt_cau[/reply] 请问下你用的是什么工具？

3楼 zt_cau 2013-10-11 10:46
如何能在debug时的调用堆栈看到类似KiFastSystemCallRet()函数名？我这边只能看到一个地址，然后自己去找函数。

2楼 Breaksoftware 2013-06-24 13:56
[reply]yurenchen[/reply] 不客气，希望对你有所帮助。

更多评论（6）

回顶部