方亮的专栏

[原]DllMain中不当操作导致死锁问题的分析--DisableThreadLibraryCalls对DllMain中死锁的影响

2012-11-7 阅读3055 评论2

《windows核心编程》作者在讨论DllMain执行序列化的时候，曾说过一个他的故事：他试图通过调用DisableThreadLibraryCalls以使得新线程不在调用DllMain从而解决死锁问题。但是该方案最后失败了。思考作者的思路，他可能一开始认为：因为线程要调用DllMain而加锁，于是windows在发现DllMain不用调用时就不用加锁了。本文将探讨DisableThreadLibraryCalls对DllMain死锁的影响。首先我们需要定位是什么函数调用了DllMain。（转载请指明出于breaksoftware的csdn博客）为了方便分析，我设计了以下代码

```
// 主程序
while ( cin>>n ) {
        string strDllName;
        DWORD dwSleepTime = 0;
        switch(n) {
        case 0:{
            strDllName = "DllWithDisableThreadLibraryCalls_A";
            dwSleepTime = 100000;
               }break;
……
        case 4:{
            strDllName = "DllWithoutDisableThreadLibraryCalls_A";
            dwSleepTime = 3000;
               }break;
……
        default:
            break;
        }
        HMODULE h = LoadLibraryA(strDllName.c_str());

        Sleep(dwSleepTime);

        if ( NULL != h ) {
            FreeLibrary(h);
        }
    }
```

该过程将根据输入值n决定加载哪个DLL。当输入0时，主线程将加载DllWithDisableThreadLibraryCalls_A.dll。它的DllMain收到DLL_PROCESS_ATTACH时，我们将调用

DisableThreadLibraryCalls以让其不再收到DLL_THREAD_ATTACH和DLL_THREAD_DETACH。

```
case DLL_PROCESS_ATTACH:{
            printf("DLL DllWithDisableThreadLibraryCalls_A:\tProcess attach (tid = %d)\n", tid);
            DisableThreadLibraryCalls(hModule);
            HANDLE hThread = CreateThread(NULL, 0, ThreadCreateInDllMain, NULL, 0, NULL);
            CloseHandle(hThread);
        }break;
```

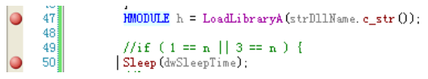
在该例程中，我们要创建一个新的线程。这是为了检测新线程是否会对该DLL有所操作，线程函数很简单。

```
static DWORD WINAPI ThreadCreateInDllMain(LPVOID) {
    return 0;
}
```

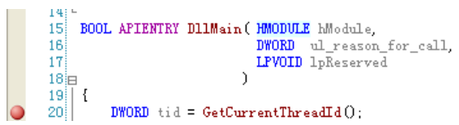
当输入4时，主线程将加载DllWithoutDisableThreadLibraryCalls_A.dll。它的DllMain收到DLL_PROCESS_ATTACH时，将直接启动一个线程（线程函数同上），而不会调用DisableThreadLibraryCalls。这步是为了让我们找出线程创建时是通过什么流程调用到DllMain函数的。

我们先让我们进程加载DllWithoutDisableThreadLibraryCalls_A.dll（输入4）以找到DllMain加载的关键路径。为了达到这个目的，我将设置几个断点：

Exe中

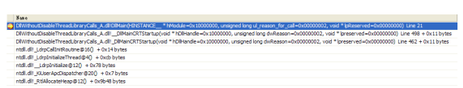


DLL中



在LoadLibrary之后Sleep了一下，是为了让工作线程有机会执行起来。

在执行到Sleep之后，程序中断在DLL中。我们看调用堆栈



我将关注下从ntdll进入DllWithoutDisableThreadLibraryCalls_A.dll的逻辑调用。双击_LdrpCallInitRoutine这行查看其汇编

```
_LdrpCallInitRoutine@16:
7C921176 55                push        ebp
7C921177 8B EC             mov         ebp,esp
7C921179 56                push        esi
7C92117A 57                push        edi
```

```
7C92117B 53              push        ebx
7C92117C 8B F4           mov         esi,esp
7C92117E FF 75 14        push        dword ptr [ebp+14h]
7C921181 FF 75 10        push        dword ptr [ebp+10h]
7C921184 FF 75 0C        push        dword ptr [ebp+0Ch]
7C921187 FF 55 08        call        dword ptr [ebp+8]
7C92118A 8B E6           mov         esp,esi
7C92118C 5B              pop         ebx
7C92118D 5F              pop         edi
7C92118E 5E              pop         esi
7C92118F 5D              pop         ebp
7C921190 C2 10 00        ret         10h
```

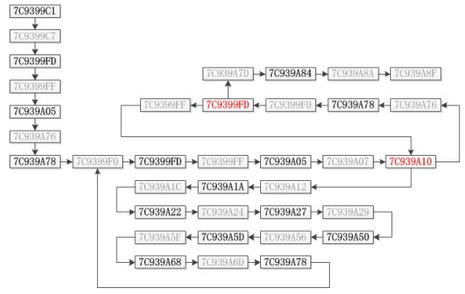
在7C921187这行，我们看到它调用了一个以参数形式传入的函数地址。_LdrpCallInitRoutine的逻辑很简单，只是简单的调用，对我们帮助不大。我们查看_LdrpInitializeThread这个函数。

```
_LdrpInitializeThread@4:
7C9399A2 6A 40           push        40h
7C9399A4 68 D8 99 93 7C  push        7C9399D8h
7C9399A9 E8 1D 4F FF FF  call        __SEH_prolog (7C92E8CBh)
7C9399AE 64 A1 18 00 00 00 mov        eax,dword ptr fs:[00000018h]
7C9399B4 8B 58 30        mov         ebx,dword ptr [eax+30h]
7C9399B7 89 5D DC        mov         dword ptr [ebp-24h],ebx
7C9399BA 80 3D C4 E0 99 7C 00 cmp        byte ptr [_LdrpShutdownInProgress (7C99E0C4h)],0
7C9399C1 0F 85 C3 00 00 00 jne        _LdrpInitializeThread@4+136h (7C939A8Ah)
7C9399C7 E8 59 FF FF FF  call        _LdrpAllocateTls@0 (7C939925h)
7C9399CC 8B 43 0C        mov         eax,dword ptr [ebx+0Ch]
7C9399CF 8B 70 14        mov         esi,dword ptr [eax+14h]
7C9399D2 EB 1C           jmp         _LdrpInitializeThread@4+30h (7C9399F0h)
7C9399D4 90              nop
7C9399D5 90              nop
7C9399D6 90              nop
7C9399D7 90              nop
7C9399D8 ??              db          ffh
7C9399D9 ??              db          ffh
7C9399DA ??              db          ffh
7C9399DB FF 00           inc         dword ptr [eax]
7C9399DD 00 00           add         byte ptr [eax],al
7C9399DF 00 B5 F3 95 7C FF add        byte ptr [ebp-836A0Dh],dh
7C9399E5 ??              db          ffh
7C9399E6 ??              db          ffh
7C9399E7 FF 00           inc         dword ptr [eax]
```

```
7C9399E9 00 00           add         byte ptr [eax],al
7C9399EB 00 62 5C        add         byte ptr [edx+5Ch],ah
7C9399EE 95              xchg        eax,ebp
7C9399EF 7C 89           jl          __LdrpInitialize@12+259h (7C93997Ah)
7C9399F1 75 E4           jne         7C9399D7
7C9399F3 8B 4B 0C        mov         ecx,dword ptr [ebx+0Ch]
7C9399F6 6A 14           push        14h
7C9399F8 58              pop         eax
7C9399F9 03 C8           add         ecx,eax
7C9399FB 3B F1           cmp         esi,ecx
7C9399FD 74 7E           je          _LdrpInitializeThread@4+0E0h (7C939A7Dh)
7C9399FF 8B 4B 08        mov         ecx,dword ptr [ebx+8]
7C939A02 3B 4E 10        cmp         ecx,dword ptr [esi+10h]
7C939A05 74 6F           je          _LdrpInitializeThread@4+0C9h (7C939A76h)
7C939A07 8B 56 2C        mov         edx,dword ptr [esi+2Ch]
7C939A0A F7 C2 00 00 04 00 test       edx,40000h
7C939A10 75 64           jne         _LdrpInitializeThread@4+0C9h (7C939A76h)
7C939A12 8B 4E 14        mov         ecx,dword ptr [esi+14h]
7C939A15 89 4D E0        mov         dword ptr [ebp-20h],ecx
7C939A18 85 C9           test        ecx,ecx
7C939A1A 74 5A           je          _LdrpInitializeThread@4+0C9h (7C939A76h)
7C939A1C F7 C2 00 00 08 00 test       edx,80000h
7C939A22 74 52           je          _LdrpInitializeThread@4+0C9h (7C939A76h)
7C939A24 F6 C2 04        test        dl,4
7C939A27 74 4D           je          _LdrpInitializeThread@4+0C9h (7C939A76h)
7C939A29 89 45 B0        mov         dword ptr [ebp-50h],eax
7C939A2C C7 45 B4 01 00 00 00 mov        dword ptr [ebp-4Ch],1
7C939A33 33 C0           xor         eax,eax
7C939A35 8D 7D B8        lea         edi,[ebp-48h]
7C939A38 AB              stos        dword ptr es:[edi]
7C939A39 AB              stos        dword ptr es:[edi]
7C939A3A AB              stos        dword ptr es:[edi]
7C939A3B FF 76 40        push        dword ptr [esi+40h]
7C939A3E 8D 45 B0        lea         eax,[ebp-50h]
7C939A41 50              push        eax
7C939A42 E8 51 77 FE FF  call        _RtlActivateActivationContextUnsafeFast@8 (7C921198h)
7C939A47 33 FF           xor         edi,edi
7C939A49 89 7D FC        mov         dword ptr [ebp-4],edi
7C939A4C 66 39 7E 32     cmp         word ptr [esi+32h],di
7C939A50 0F 85 93 0C 01 00 jne        _LdrpInitializeThread@4+96h (7C94A6E9h)
```

```
7C939A56 80 3D C4 E0 99 7C 00 cmp         byte ptr [_LdrpShutdownInProgress (7C99E0C4h)],0
7C939A5D 75 0E            jne         _LdrpInitializeThread@4+0C0h (7C939A6Dh)
7C939A5F 57               push        edi
7C939A60 6A 02            push        2
7C939A62 FF 76 10         push        dword ptr [esi+10h]
7C939A65 FF 75 E0         push        dword ptr [ebp-20h]
7C939A68 E8 09 77 FE FF   call        _LdrpCallInitRoutine@16 (7C921176h)
7C939A6D 83 4D FC FF      or          dword ptr [ebp-4],0FFFFFFFFh
7C939A71 E8 1D FF FF FF   call        _LdrpInitializeThread@4+0D6h (7C939993h)
7C939A76 8B 36            mov         esi,dword ptr [esi]
7C939A78 E9 73 FF FF FF   jmp         _LdrpInitializeThread@4+30h (7C9399F0h)
7C939A7D 80 3D DC E0 99 7C 00 cmp         byte ptr [_LdrpImageHasTls (7C99E0DCh)],0
7C939A84 0F 85 7D C1 01 00 jne         _LdrpInitializeThread@4+0E9h (7C955C07h)
7C939A8A E8 77 4E FF FF   call        __SEH_epilog (7C92E906h)
7C939A8F C2 04 00         ret         4
```

7C939A68处是我们调用LdrpCallInitRoutine的地方。从_LdrpInitializeThread这个函数名看，它应该是执行一些线程初始化操作，由《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》中我们得知，线程在初始化期间将调用加载的DLL的DllMain。于是我们重点将关注于该函数。我们在所有条件跳转地方设断点。然后让我们程序恢复运行，发现程序分别在7C939A78，7C9399FD，7C939A84中断一次后便进入线程函数中，先不管它。现在我们输入0，我们让我们程序加载DllWithDisableThreadLibraryCalls_A.dll。我们查看调用流程



由以上流程可以发现，标红色的7C939A10前一句和7C9399FD前一句是重要的跳转条件判断。它们分别是

```
7C939A0A F7 C2 00 00 04 00 test        edx,40000h
```

```
7C9399F6 6A 14            push        14h
7C9399F8 58               pop         eax
7C9399F9 03 C8            add         ecx,eax
7C9399FB 3B F1            cmp         esi,ecx
```

而且可以分析出7C939A0A可能是因为没有调用DllMain的主因。即可能是DisableThreadLibraryCalls设置了某结构体的某字段Or 40000h了。以下为了简洁，我不再引入汇编，而使用网上盛传的Win2K中的相关C代码加以说明。

Kernel32中的DisableThreadLibraryCalls底层调用了ntdll中的LdrDisableThreadCalloutsForDll函数。我们看下LdrDisableThreadCalloutsForDll代码

```
NTSTATUS
```

```
LdrDisableThreadCalloutsForDll (
    IN PVOID DllHandle
    )
/*++
Routine Description:
    This function disables thread attach and detach notification
    for the specified DLL.
Arguments:
    DllHandle - Supplies a handle to the DLL to disable.
Return Value:
    TBD
--*/

{
    NTSTATUS st;
    PLDR_DATA_TABLE_ENTRY LdrDataTableEntry;

    st = STATUS_SUCCESS;

    try {
    if ( LdrpInLdrInit == FALSE ) {
        RtlEnterCriticalSection((PRTL_CRITICAL_SECTION)NtCurrentPeb()->LoaderLock);
        }
    if ( LdrpShutdownInProgress ) {
        return STATUS_SUCCESS;
        }

    if (LdrpCheckForLoadedDllHandle(DllHandle, &LdrDataTableEntry)) {
        if ( LdrDataTableEntry->TlsIndex ) {
            st = STATUS_DLL_NOT_FOUND;
            }
        else {
            LdrDataTableEntry->Flags |= LDRP_DONT_CALL_FOR_THREADS;
            }
        }
    }
finally {
    if ( LdrpInLdrInit == FALSE ) {
        RtlLeaveCriticalSection((PRTL_CRITICAL_SECTION)NtCurrentPeb()->LoaderLock);
```

```
            }
        }
    return st;
}
```

我们看第35行，会发现`LdrDataTableEntry->Flags or 了 LDRP_DONT_CALL_FOR_THREADS(0x400000)`。这个也就验证了刚才分析的`_LdrpInitializeThread`逻辑中的没有调用`DllMain`的原因。

我们再看下LdrpInitializeThread的代码

```
VOID LdrpInitializeThread( IN PCONTEXT Context )
{
    PPEB Peb;
    PLDR_DATA_TABLE_ENTRY LdrDataTableEntry;
    PDLL_INIT_ROUTINE InitRoutine;
    PLIST_ENTRY Next;

    Peb = NtCurrentPeb();

    if ( LdrpShutdownInProgress ) {
        return;
    }

LdrpAllocateTls();

Next = Peb->Ldr->InMemoryOrderModuleList.Flink;
while (Next != &Peb->Ldr->InMemoryOrderModuleList) {
    LdrDataTableEntry
        = (PLDR_DATA_TABLE_ENTRY)
        (CONTAINING_RECORD(Next, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks));

    //
    // Walk through the entire list looking for
    // entries. For each entry, that has an init
    // routine, call it.
    //
    if (Peb->ImageBaseAddress != LdrDataTableEntry->DllBase) {
        if ( !(LdrDataTableEntry->Flags & LDRP_DONT_CALL_FOR_THREADS)) {
            InitRoutine = (PDLL_INIT_ROUTINE)LdrDataTableEntry->EntryPoint;
            if (InitRoutine && (LdrDataTableEntry->Flags & LDRP_PROCESS_ATTACH_CALLED) ) {
                if (LdrDataTableEntry->Flags & LDRP_IMAGE_DLL) {
                    if ( LdrDataTableEntry->TlsIndex ) {
```

```
                            if ( !LdrpShutdownInProgress ) {
                                LdrpCallTlsInitializers(LdrDataTableEntry->DllBase,DLL_THREAD_ATTACH);
                            }
                        }

#if defined (WX86)
                        if (!Wx86ProcessInit ||
                            LdrpRunWx86DllEntryPoint(InitRoutine,
                            NULL,
                            LdrDataTableEntry->DllBase,
                            DLL_THREAD_ATTACH,
                            NULL
                            ) ==  STATUS_IMAGE_MACHINE_TYPE_MISMATCH)
#endif
                        {
                            if ( !LdrpShutdownInProgress ) {
                                LdrpCallInitRoutine(InitRoutine,
                                    LdrDataTableEntry->DllBase,
                                    DLL_THREAD_ATTACH,
                                    NULL);
                                }
                            }
                        }
                    }
                }
            }
            Next = Next->Flink;
        }

        //
        // If the image has tls than call its initializers
        //

        if ( LdrpImageHasTls && !LdrpShutdownInProgress ) {
            LdrpCallTlsInitializers(NtCurrentPeb()->ImageBaseAddress,DLL_THREAD_ATTACH);
        }

}
```

这段逻辑的意思是：拿到PEB之后，从PEB的LDR字段中的InMemoryOrderModuleList中获取已经加载进入内存中的DLL信息。枚举这些DLL信息，如果该DLL信息的Flags字段或上LDRP_DONT_CALL_FOR_THREADS（0x40000），则不对其调用LdrpCallInitRoutine，进而不对调用DllMain。这就是为什么DisableThreadLibraryCalls会禁止DllMain的调用的原因。但是目前为止，我们

还没发现哪个逻辑进入临界区而没出来，这样我们就要查看调用LdrpInitializeThread的LdrpInitialize的代码

```
VOID
LdrpInitialize (
                IN PCONTEXT Context,
                IN PVOID SystemArgument1,
                IN PVOID SystemArgument2
                )
{
    ……
        Peb->LoaderLock = (PVOID)&LoaderLock;

    if ( !RtlTryEnterCriticalSection(&LoaderLock) ) {
        if ( LoaderLockInitialized ) {
            RtlEnterCriticalSection(&LoaderLock);
        }
        else {

            //
            // drop into a 30ms delay loop
            //

        DelayValue.QuadPart = Int32x32To64( 30, -10000 );
        while ( !LoaderLockInitialized ) {
            NtDelayExecution(FALSE,&DelayValue);
        }
        RtlEnterCriticalSection(&LoaderLock);
    }
}
……
    LdrpInitializeThread(Context);
……
    RtlLeaveCriticalSection(&LoaderLock);
…
```

我们看到在11或13或25行进入临界区后，在29行调用了LdrpInitializeThread，而在31行退出临界区。这就是说整个LdrpInitializeThread的逻辑都在临界区中执行的，也就是说DisableThreadLibraryCalls将无权干涉是否会进入临界区。这就解释了为什么不能使用DisableThreadLibraryCalls来使上例解决死锁的原因。

上一篇 下一篇

发表评论

提 交

查看评论

2楼 Breaksoftware 2014-11-18 23:41

[reply]f472969530[/reply] 你把AfxMessageBox("测试...加载dll成功");去掉试试。而且dllmain里不要启动线程，具体原因看我这系列的分析博客。其中涉及很多系统方面知识，算是隐性的问题。

1楼 f472969530 2014-11-18 14:05

不是很明白，遇到相似问题 http://bbs.csdn.net/topics/390934786 求解答

更多评论（2）

回顶部