

在笔者上一篇文章《内核枚举IoTimer定时器》中我们通过 IoInitializeTimer 这个API函数为跳板，向下扫描特征码获取到了 IopTimerQueueHead 也就是IO定时器的队列头，本章学习的枚举DPC定时器依然使用特征码扫描，唯一不同的是在新版系统中DPC是被异或加密的，想要找到正确的地址，只是需要在找到DPC表头时进行解密操作即可。

进程	驱动模块	内核层	内核钩子	应用层钩子	设置	监控	启动信息	注册表	服务	文件	网络	调试引擎
系统回调	过滤驱动	DPC定时器	IO定时器	系统线程	卸载的驱动							
定时器对象	DPC	触发周期(s)	函数入口	模块路径								
0xFFFFF803161EB1E0	0xFFFFF803161EB1A0	0	0xFFFFF803161E1170	C:\Windows\System32\Drivers\dfsc.sys								
0xFFFFC88CCF6C26D0	0xFFFFC88CCF6C2690	0	0xFFFFF80315F9AA20	C:\Windows\system32\drivers\afd.sys								
0xFFFFF803147323E0	0xFFFFF80314732420	0	0xFFFFF80314724410	C:\Windows\system32\drivers\pdc.sys								
0xFFFFF803100263A0	0xFFFFF803100263E0	0	0xFFFFF8030FCFC250	C:\Windows\system32\ntoskrnl.exe								
0xFFFFF803152E9160	0xFFFFF803152E9100	0	0xFFFFF803150E6840	C:\Windows\System32\drivers\tcpip.sys								
0xFFFFC88CCF318178	0xFFFFC88CCF3181B8	0	0xFFFFF8030FD7EB80	C:\Windows\system32\ntoskrnl.exe								
0xFFFFC88CCF319178	0xFFFFC88CCF3191B8	0	0xFFFFF8030FD7EB80	C:\Windows\system32\ntoskrnl.exe								
0xFFFFC88CCF31B178	0xFFFFC88CCF31B1B8	0	0xFFFFF8030FD7EB80	C:\Windows\system32\ntoskrnl.exe								
0xFFFFC88CCF31C178	0xFFFFC88CCF31C1B8	0	0xFFFFF8030FD7EB80	C:\Windows\system32\ntoskrnl.exe								
0xFFFFC88CCF31E178	0xFFFFC88CCF31E1B8	0	0xFFFFF8030FD7EB80	C:\Windows\system32\ntoskrnl.exe								

DPC定时器的作用是什么？

DPC（Deferred Procedure Call）是一种异步执行的机制。它允许内核代码在不中断当前进程的情况下，延迟执行一些工作。DPC的执行是由内核定时器触发的。内核定时器是一种特殊的内核对象，用于定时执行某个特定的操作。在DPC的上下文中，内核可以安全地访问任何内核数据结构，而不会引起死锁或其他问题。

在内核中可以使用DPC定时器设置任意定时任务，当到达某个节点时自动触发定时回调，定时器的内部使用 KTIMER 对象，当设置任务时会自动插入到 DPC 队列，由操作系统循环读取 DPC 队列并执行任务，枚举 DPC 定时器可得知系统中存在的 DPC 任务。

要想在新版系统中得到DPC定时器则需要执行的步骤有哪些？

- 1.找到 KiProcessorBlock 地址并解析成 \_KPRCB 结构
- 2.在 \_KPRCB 结构中得到 \_KTIMER\_TABLE 偏移
- 3.解析 \_KTIMER\_TABLE\_ENTRY 得到加密后的双向链表

首先 \_KPRCB 这个结构体与CPU内核对应，获取方式可通过一个未导出的变量 nt!KiProcessorBlock 来得到，如下双核电脑，结构体存在两个与之对应的结构地址。

```
lyshark.com 0: kd> dq nt!KiProcessorBlock
fffff807`70a32cc0 fffff807`6f77c180 fffffbe81`3cee0180
fffff807`70a32cd0 00000000`00000000 00000000`00000000
fffff807`70a32ce0 00000000`00000000 00000000`00000000
```

此 KiProcessorBlock 是一个数组，其第一个结构体 TimerTable 则是结构体的偏移。

```
lyshark.com 0: kd> dt _KPRCB fffff807`6f77c180
ntdll!_KPRCB
+0x000 MxCsr           : 0x1f80
+0x3680 TimerTable     : _KTIMER_TABLE （此处）
+0x5880 DpcGate        : _KGATE
```

接下来是把所有的 KTIMER 都枚举出来，KTIMER在 TimerTable 中的存储方式是数组+双向链表。

```

lyshark.com 0: kd> dt _KTIMER_TABLE
ntdll!_KTIMER_TABLE
+0x000 TimerExpiry      : [64] Ptr64 _KTIMER
+0x200 TimerEntries     : [256] _KTIMER_TABLE_ENTRY (此处)

```

到了 `_KTIMER_TABLE_ENTRY` 这里，Entry 开始的双向链表，每一个元素都对应一个 Timer 也就是说我们已经可以遍历所有未解密的 Time 变量了。

```

lyshark.com 0: kd> dt _KTIMER_TABLE_ENTRY 0xffffffff807`6f77c180 + 0x3680
ntdll!_KTIMER_TABLE_ENTRY
+0x000 Lock             : 0
+0x008 Entry            : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`00000000 ]
+0x018 Time             : _ULARGE_INTEGER 0x0

lyshark.com 0: kd> dt _KTIMER_TABLE_ENTRY 0xffffffff807`6f77c180 + 0x3680 + 0x200
ntdll!_KTIMER_TABLE_ENTRY
+0x000 Lock             : 0
+0x008 Entry            : _LIST_ENTRY [ 0xfffffa707`a0d3e1a0 - 0xfffffa707`a0d3e1a0 ]
+0x018 Time             : _ULARGE_INTEGER 0x00000001`a8030353

```

至于如何解密，我们需要得到加密位置，如下通过 `KeSetTimer` 找到 `KeSetTimerEx` 从中得到 DCP 加密流程。

```

lyshark.com 0: kd> u nt!KeSetTimer
nt!KeSetTimer:
fffff803`0fc63a40 4883ec38      sub     rsp,38h
fffff803`0fc63a44 4c89442420     mov     qword ptr [rsp+20h],r8
fffff803`0fc63a49 4533c9        xor     r9d,r9d
fffff803`0fc63a4c 4533c0        xor     r8d,r8d
fffff803`0fc63a4f e80c000000     call    nt!KiSetTimerEx (fffff803`0fc63a60)
fffff803`0fc63a54 4883c438      add     rsp,38h
fffff803`0fc63a58 c3            ret
fffff803`0fc63a59 cc            int     3

0: kd> u nt!KiSetTimerEx 150
nt!KiSetTimerEx:
fffff803`0fc63a60 48895c2408     mov     qword ptr [rsp+8],rbx
fffff803`0fc63a65 48896c2410     mov     qword ptr [rsp+10h],rbp
fffff803`0fc63a6a 4889742418     mov     qword ptr [rsp+18h],rsi
fffff803`0fc63a6f 57            push    rdi
fffff803`0fc63a70 4154          push    r12
fffff803`0fc63a72 4155          push    r13
fffff803`0fc63a74 4156          push    r14
fffff803`0fc63a76 4157          push    r15
fffff803`0fc63a78 4883ec50      sub     rsp,50h
fffff803`0fc63a7c 488b057d0c5100 mov     rax,qword ptr [nt!KiWaitNever (fffff803`10174700)]
fffff803`0fc63a83 488bf9        mov     rdi,rcx
fffff803`0fc63a86 488b35630e5100 mov     rsi,qword ptr [nt!KiWaitAlways (fffff803`101748f0)]
fffff803`0fc63a8d 410fb6e9      movzx   ebp,r9b
fffff803`0fc63a91 4c8bac24a0000000 mov     r13,qword ptr [rsp+0A0h]
fffff803`0fc63a99 458bf8        mov     r15d,r8d

```

```

fffff803`0fc63a9c 4933f5      xor     rsi,r13
fffff803`0fc63a9f 488bda      mov     rbx,rdx
fffff803`0fc63aa2 480fce      bswap   rsi
fffff803`0fc63aa5 4833f1      xor     rsi,rcx
fffff803`0fc63aa8 8bc8        mov     ecx,eax
fffff803`0fc63aaa 48d3ce      ror     rsi,cl
fffff803`0fc63aad 4833f0      xor     rsi,rax
fffff803`0fc63ab0 440f20c1    mov     rcx,cr8
fffff803`0fc63ab4 48898c24a0000000 mov     qword ptr [rsp+0A0h],rcx
fffff803`0fc63abc b802000000    mov     eax,2
fffff803`0fc63ac1 440f22c0    mov     cr8,rax
fffff803`0fc63ac5 8b05dd0a5100 mov     eax,dword ptr [nt!KiIrqlFlags (fffff803`101745a8)]
fffff803`0fc63acb 85c0        test    eax,eax
fffff803`0fc63acd 0f85b72d1a00 jne     nt!KiSetTimerEx+0x1a2e2a (fffff803`0fe0688a)
fffff803`0fc63ad3 654c8b342520000000 mov     r14,qword ptr gs:[20h]
fffff803`0fc63adc 33d2        xor     edx,edx
fffff803`0fc63ade 488bcf      mov     rcx,rdi
fffff803`0fc63ae1 e86aa2fdff    call    nt!KiCancelTimer (fffff803`0fc3dd50)
fffff803`0fc63ae6 440fb6e0    movzx   r12d,a1
fffff803`0fc63aea 48897730    mov     qword ptr [rdi+30h],rsi
fffff803`0fc63aee 33c0        xor     eax,eax
fffff803`0fc63af0 44897f3c    mov     dword ptr [rdi+3Ch],r15d
fffff803`0fc63af4 8b0f        mov     ecx,dword ptr [rdi]
fffff803`0fc63af6 4889442430    mov     qword ptr [rsp+30h],rax
fffff803`0fc63afb 894c2430    mov     dword ptr [rsp+30h],ecx
fffff803`0fc63aff 488bcb      mov     rcx,rbx
fffff803`0fc63b02 48c1e920    shr     rcx,20h
fffff803`0fc63b06 4889442438    mov     qword ptr [rsp+38h],rax
fffff803`0fc63b0b 4889442440    mov     qword ptr [rsp+40h],rax
fffff803`0fc63b10 40886c2431    mov     byte ptr [rsp+31h],bp1
fffff803`0fc63b15 85c9        test    ecx,ecx
fffff803`0fc63b17 0f89c0000000 jns     nt!KiSetTimerEx+0x17d (fffff803`0fc63bdd)
fffff803`0fc63b1d 33c9        xor     ecx,ecx
fffff803`0fc63b1f 8bd1        mov     edx,ecx
fffff803`0fc63b21 40f6c5fc    test    bp1,0FCh
fffff803`0fc63b25 0f85a3000000 jne     nt!KiSetTimerEx+0x16e (fffff803`0fc63bce)
fffff803`0fc63b2b 48894c2420    mov     qword ptr [rsp+20h],rcx
fffff803`0fc63b30 48b80800000080f7ffff mov     rax,0FFFFFF78000000008h
fffff803`0fc63b3a 4d8bc5      mov     r8,r13
fffff803`0fc63b3d 488b00      mov     rax,qword ptr [rax]
fffff803`0fc63b40 804c243340    or      byte ptr [rsp+33h],40h
fffff803`0fc63b45 482bc3      sub     rax,rbx
fffff803`0fc63b48 48894718    mov     qword ptr [rdi+18h],rax
fffff803`0fc63b4c 4803c2      add     rax,rdx
fffff803`0fc63b4f 48c1e812    shr     rax,12h
fffff803`0fc63b53 488bd7      mov     rdx,rdi
fffff803`0fc63b56 440fb6c8    movzx   r9d,a1
fffff803`0fc63b5a 44884c2432    mov     byte ptr [rsp+32h],r9b
fffff803`0fc63b5f 8b442430    mov     eax,dword ptr [rsp+30h]
fffff803`0fc63b63 8907        mov     dword ptr [rdi],eax
fffff803`0fc63b65 894f04      mov     dword ptr [rdi+4],ecx
fffff803`0fc63b68 498bce      mov     rcx,r14
fffff803`0fc63b6b e8209ffdfc    call    nt!KiInsertTimerTable (fffff803`0fc3da90)

```

```

fffff803`0fc63b70 84c0          test     al,al
fffff803`0fc63b72 0f8495000000    je      nt!KiSetTimerEx+0x1ad (fffff803`0fc63c0d)
fffff803`0fc63b78 f705860851000000200 test dword ptr [nt!PerfGlobalGroupMask+0x8
(fffff803`10174408)],20000h
fffff803`0fc63b82 0f852f2d1a00    jne     nt!KiSetTimerEx+0x1a2e57 (fffff803`0fe068b7)
fffff803`0fc63b88 f081277fffffff  lock   and dword ptr [rdi],0FFFFFF7Fh
fffff803`0fc63b8f 488b8424a0000000 mov     rax,qword ptr [rsp+0A0h]
fffff803`0fc63b97 4533c9          xor     r9d,r9d
fffff803`0fc63b9a 33d2            xor     edx,edx
fffff803`0fc63b9c 88442420        mov     byte ptr [rsp+20h],al
fffff803`0fc63ba0 498bce          mov     rcx,r14
fffff803`0fc63ba3 458d4101        lea     r8d,[r9+1]
fffff803`0fc63ba7 e8044efeff      call   nt!KiExitDispatcher (fffff803`0fc489b0)

```

如上汇编代码 KiSetTimerEx 中就是DPC加密细节，如果需要解密只需要逆操作即可，此处我就具体分析下加密细节，分析这个东西我建议你使用记事本带着色的。

分析思路是这样的，首先这里要传入待加密的DPC数据，然后经过 KiWaitNever 和 KiWaitAlways 对数据进行 xor,ror,bswap 等操作。

```

lyshark.md
1  0: kd> u nt!KiSetTimerEx 150
2  nt!KiSetTimerEx:
3  fffff803`0fc63a60 48895c2408      mov     qword ptr [rsp+8],rbx
4  fffff803`0fc63a65 48896c2410      mov     qword ptr [rsp+10h],rbp
5  fffff803`0fc63a6a 4889742418      mov     qword ptr [rsp+18h],rsi
6  fffff803`0fc63a6f 57              push    rdi
7  fffff803`0fc63a70 4154            push    r12
8  fffff803`0fc63a72 4155            push    r13
9  fffff803`0fc63a74 4156            push    r14
10 fffff803`0fc63a76 4157            push    r15
11 fffff803`0fc63a78 488bec50        sub     rsp,50h
12 fffff803`0fc63a7c 488b057d0c5100 mov     rax,qword ptr [nt!KiWaitNever (fffff803`10174700)]
13 fffff803`0fc63a83 488bf9          mov     rdi,rcx
14 fffff803`0fc63a86 488b35630e5100 mov     rsi,qword ptr [nt!KiWaitAlways (fffff803`101748f0)]
15 fffff803`0fc63a8d 410fb6e9        movzx   ebp,r9b
16 fffff803`0fc63a91 4c8bac24a0000000 mov     r13,qword ptr [rsp+0A0h]
17 fffff803`0fc63a99 458bf8          mov     r15d,r8d
18 fffff803`0fc63a9c 4933f5          xor     rsi,r13
19 fffff803`0fc63a9f 488bda          mov     rbx,rdx
20 fffff803`0fc63aa2 480fce          bswap   rsi
21 fffff803`0fc63aa5 4833f1          xor     rsi,rcx
22 fffff803`0fc63aa8 8bc8            mov     ecx,eax
23 fffff803`0fc63aaa 48d3ce          ror     rsi,cl
24 fffff803`0fc63aad 4833f0          xor     rsi,rax
25 fffff803`0fc63ab0 440f20c1        mov     rcx,cr8

```

将如上所示的汇编解密流程通过C语言的方式实现，解密函数 DPC\_Print 过程可以被总结为如下几个流程：

- 首先获取定时器结构体中 Dpc 成员的地址，将其转换为 ULONG\_PTR 类型的指针 ptrDpc。
- 然后将 ptrDpc 异或上一个常量 p2dq(ptrKiWaitNever)，这个常量是 KiWaitNever 的指针地址强制转换为 ULONG\_PTR 类型后的结果，相当于异或上一个随机值来进行简单的加密。
- 然后将 ptrDpc 循环左移 nShift 位，其中 nShift 的值为 KiWaitNever 指针值的低8位，即取最后一个字节。
- 接着将 ptrDpc 异或上定时器结构体的地址，相当于对加密结果进行一个简单的混淆。然后对 ptrDpc 进行字节交换，相当于将 ptrDpc 的字节序进行翻转，以便在后面的代码中能够正确地解密DPC结构体。最后将 ptrDpc 异或上一个常量 p2dq(ptrKiWaitAlways)，这个常量是 KiWaitAlways 的指针地址强制转换为 ULONG\_PTR 类型后的结果，相当于再进行一次简单的加密。

最后，如果解密得到的DPC结构体指针 DecDpc 是一个有效的内核地址，就输出该DPC的地址和它的延迟函数地址。其中 DeferredRoutine 是KDPC结构体的一个成员，用于保存DPC的回调函数地址，将上述流程通过代码方式实现则如下所示；

```
#include <ntddk.h>
#include <ntstrsafe.h>

// 解密DPC
void DPC_Print(PKTIMER ptrTimer)
{
    ULONG_PTR ptrDpc = (ULONG_PTR)ptrTimer->Dpc;
    KDPC* DecDpc = NULL;
    DWORD nShift = (p2dq(ptrKiWaitNever) & 0xFF);

    // _RSI->Dpc = (_KDPC *)v19;
    // _RSI = Timer;
    ptrDpc ^= p2dq(ptrKiWaitNever);          // v19 = KiWaitNever ^ v18;
    ptrDpc = _rotl64(ptrDpc, nShift);        // v18 = __ROR8__((unsigned __int64)Timer ^ _RBX,
    KiWaitNever);
    ptrDpc ^= (ULONG_PTR)ptrTimer;
    ptrDpc = _byteswap_uint64(ptrDpc);       // __asm { bswap  rbx }
    ptrDpc ^= p2dq(ptrKiWaitAlways);        // _RBX = (unsigned __int64)DPC ^ KiWaitAlways;

    // real DPC
    if (MmIsAddressValid((PVOID)ptrDpc))
    {
        DecDpc = (KDPC*)ptrDpc;
        DbgPrint("DPC = %p | routine = %p \n", DecDpc, DecDpc->DeferredRoutine);
    }
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("卸载完成... \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com");

    PKTIMER ptrTimer = NULL;

    DPC_Print(ptrTimer);

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

接着将这些功能通过代码实现，首先得到我们需要的函数地址，这些地址包括。

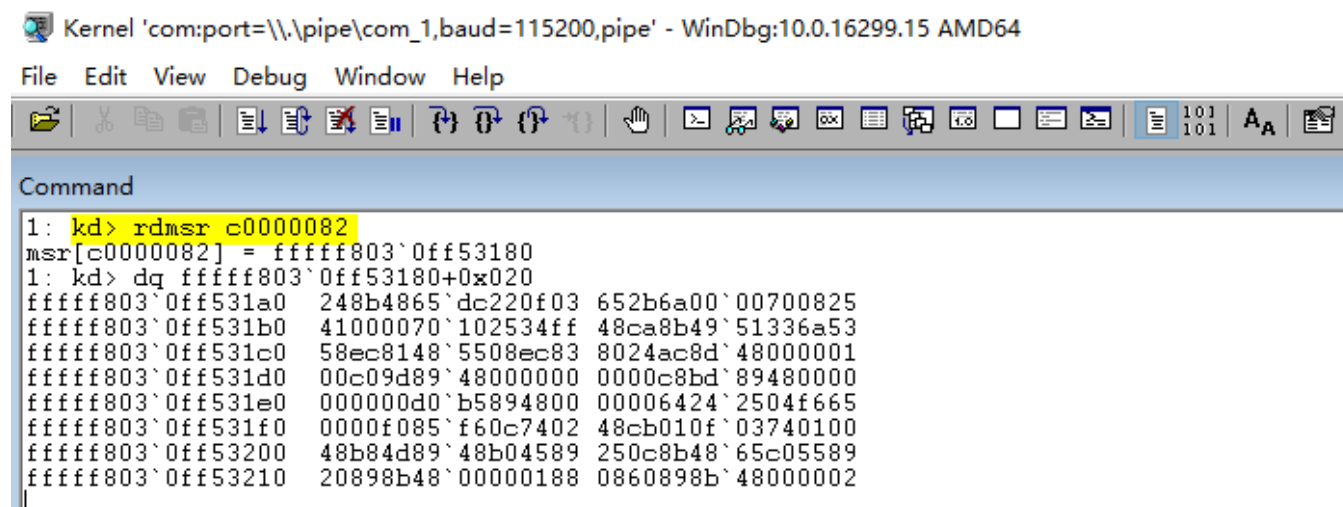
```
ULONG_PTR ptrKiProcessorBlock = 0xfffff80770a32cc0;
ULONG_PTR ptrOffsetKTimerTable = 0x3680;
ULONG_PTR ptrKiwaitNever = 0xfffff80770a316f8;
ULONG_PTR ptrKiwaitAlways = 0xfffff80770a318e8;
```

此处我把它分为三步走，第一步找到 `KiProcessorBlock` 函数地址，第二步找到 `KeSetTimer` 并从里面寻找 `KeSetTimerEx`，第三步根据 `KiSetTimerEx` 地址，搜索到 `KiwaitNever()`, `KiwaitAlways()` 这两个函数内存地址，最终循环链表并解密DPC队列。

## 寻找KiProcessorBlock函数

找到 `KiProcessorBlock` 函数地址，该地址可通过 `__readmsr()` 寄存器相加偏移得到。

在WinDBG中可以输入 `rdmsr c0000082` 得到MSR地址。



```
Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:10.0.16299.15 AMD64
File Edit View Debug Window Help
[Icons]
Command
1: kd> rdmsr c0000082
msr[c0000082] = fffff803`0ff53180
1: kd> dq fffff803`0ff53180+0x020
fffff803`0ff531a0 248b4865`dc220f03 652b6a00`00700825
fffff803`0ff531b0 41000070`102534ff 48ca8b49`51336a53
fffff803`0ff531c0 58ec8148`5508ec83 8024ac8d`48000001
fffff803`0ff531d0 00c09d89`48000000 0000c8bd`89480000
fffff803`0ff531e0 000000d0`b5894800 00006424`2504f665
fffff803`0ff531f0 0000f085`f60c7402 48cb010f`03740100
fffff803`0ff53200 48b84d89`48b04589 250c8b48`65c05589
fffff803`0ff53210 20898b48`00000188 0860898b`48000002
```

MSR寄存器 使用 代码获取 也是很容易，只要找到MSR地址在加上 `0x20` 即可得到 `KiProcessorBlock` 的地址了。

```
/*
lyshark.com 0: kd> dp !KiProcessorBlock
fffff807`70a32cc0 fffff807`6f77c180 fffffbe81`3cee0180
fffff807`70a32cd0 00000000`00000000 00000000`00000000
fffff807`70a32ce0 00000000`00000000 00000000`00000000
fffff807`70a32cf0 00000000`00000000 00000000`00000000
fffff807`70a32d00 00000000`00000000 00000000`00000000
fffff807`70a32d10 00000000`00000000 00000000`00000000
fffff807`70a32d20 00000000`00000000 00000000`00000000
fffff807`70a32d30 00000000`00000000 00000000`00000000
*/

#include <ntddk.h>
#include <ntstrsafe.h>

// 得到KiProcessorBlock地址
ULONG64 GetKiProcessorBlock()
{
    ULONG64 PrcbAddress = 0;
    PrcbAddress = (ULONG64)__readmsr(0xc0000101) + 0x20;
```

```

if (PrcbAddress != 0)
{
    // PrcbAddress 是一个地址 这个地址存放了某个 CPU 的 _KPRCB 的地址
    return *(ULONG_PTR*)PrcbAddress;
}
return 0;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("卸载完成... \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    ULONG64 address = GetKiProcessorBlock();
    if (address != 0)
    {
        DbgPrint("KiProcessorBlock = %p \n", address);
    }

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

运行后即可得到输出效果如下：

#	Time	Debug Print
51	28.22346497	hello lyshark.com
52	28.22807693	KiProcessorBlock = FFFFB301976A5180
53	28.42008972	2779375000 - STORMINI: StorNVMe - POWER: IDLE
54	28.99872971	卸载完成...

## 寻找KeSetTimerEx函数

找到 KeSetTimer 从里面搜索特征得到 call KeSetTimerEx 函数地址，还记得《内核枚举IoTimer定时器》中我们采用的特征码定位方式吗，没错本次还要使用这个方法，我们此处需要搜索到 e80c000000 这段特征。

```

/*
lyshark.com 0: kd> uf KeSetTimer
nt!KeSetTimer:
fffff807`70520a30 4883ec38      sub     rsp,38h
fffff807`70520a34 4c89442420    mov     qword ptr [rsp+20h],r8
fffff807`70520a39 4533c9       xor     r9d,r9d
fffff807`70520a3c 4533c0       xor     r8d,r8d
fffff807`70520a3f e80c000000    call   nt!KiSetTimerEx (fffff807`70520a50)
fffff807`70520a44 4883c438      add     rsp,38h

```

```

fffff807`70520a48 c3          ret
*/

#include <ntddk.h>
#include <ntstrsafe.h>

// 得到KiProcessorBlock地址
ULONG64 GetKeSetTimerEx()
{
    // 获取 KeSetTimer 地址
    ULONG64 ul_KeSetTimer = 0;
    UNICODE_STRING uc_KeSetTimer = { 0 };
    RtlInitUnicodeString(&uc_KeSetTimer, L"KeSetTimer");
    ul_KeSetTimer = (ULONG64)MmGetSystemRoutineAddress(&uc_KeSetTimer);
    if (ul_KeSetTimer == 0)
    {
        return 0;
    }

    // 前 30 字节找 call 指令
    BOOLEAN b_e8 = FALSE;
    ULONG64 ul_e8Addr = 0;

    for (INT i = 0; i < 30; i++)
    {
        // 验证地址是否可读写
        if (!MmIsAddressValid((PVOID64)ul_KeSetTimer))
        {
            continue;
        }

        // e8 0c 00 00 00 call nt!KiSetTimerEx (fffff807`70520a50)
        if (*(PCHAR)(ul_KeSetTimer + i) == 0xe8)
        {
            b_e8 = TRUE;
            ul_e8Addr = ul_KeSetTimer + i;
            break;
        }
    }

    // 找到 call 则解析目的地址
    if (b_e8 == TRUE)
    {
        if (!MmIsAddressValid((PVOID64)ul_e8Addr))
        {
            return 0;
        }

        INT ul_callCode = *(INT*)(ul_e8Addr + 1);
        ULONG64 ul_KiSetTimerEx = ul_e8Addr + ul_callCode + 5;
        return ul_KiSetTimerEx;
    }
    return 0;
}

```

```

}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("卸载完成... \n");
}

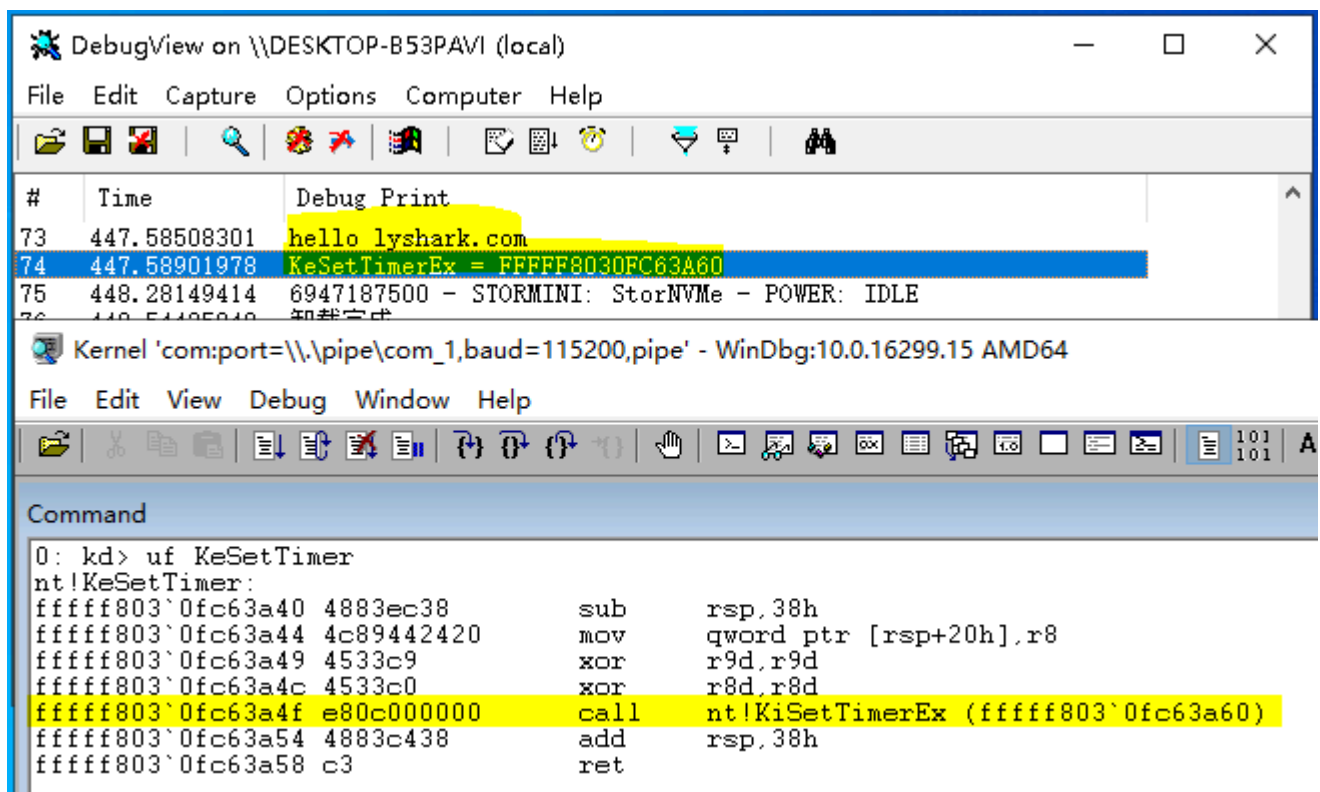
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    ULONG64 address = GetKeSetTimerEx();
    if (address != 0)
    {
        DbgPrint("KeSetTimerEx = %p \n", address);
    }

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

输出寻找CALL地址效果图如下：



## 寻找KiWaitNever函数

这一步也是最重要的，在KiSetTimerEx里面，搜索特征，拿到里面的KiwaitNever(),KiwaitAlways()这两个函数地址。

- 488b05850c5100 KiWaitNever
- 488b356b0e5100 KiWaitAlways

这个过程需要重复搜索，所以要把第一步和第二部过程归纳起来，具体代码如下所示。

```
/*
0: kd> uf KiSetTimerEx
nt!KiSetTimerEx:
fffff807`70520a50 48895c2408      mov     qword ptr [rsp+8],rbx
fffff807`70520a55 48896c2410      mov     qword ptr [rsp+10h],rbp
fffff807`70520a5a 4889742418      mov     qword ptr [rsp+18h],rsi
fffff807`70520a5f 57             push    rdi
fffff807`70520a60 4154           push    r12
fffff807`70520a62 4155           push    r13
fffff807`70520a64 4156           push    r14
fffff807`70520a66 4157           push    r15
fffff807`70520a68 4883ec50       sub     rsp,50h
fffff807`70520a6c 488b05850c5100  mov     rax,qword ptr [nt!KiWaitNever
(fffff807`70a316f8)]
fffff807`70520a73 488bf9         mov     rdi,rcx
fffff807`70520a76 488b356b0e5100  mov     rsi,qword ptr [nt!KiWaitAlways
(fffff807`70a318e8)]
fffff807`70520a7d 410fb6e9       movzx   ebp,r9b
*/

#include <ntddk.h>
#include <ntstrsafe.h>

// 得到KiProcessorBlock地址
ULONG64 GetKeSetTimerEx()
{
    // 获取 KeSetTimer 地址
    ULONG64 ul_KeSetTimer = 0;
    UNICODE_STRING uc_KeSetTimer = { 0 };
    RtlInitUnicodeString(&uc_KeSetTimer, L"KeSetTimer");
    ul_KeSetTimer = (ULONG64)MmGetSystemRoutineAddress(&uc_KeSetTimer);
    if (ul_KeSetTimer == 0)
    {
        return 0;
    }

    // 前 30 字节找 call 指令
    BOOLEAN b_e8 = FALSE;
    ULONG64 ul_e8Addr = 0;

    for (INT i = 0; i < 30; i++)
    {
        // 验证地址是否可读写
        if (!MmIsAddressValid((PVOID64)ul_KeSetTimer))
        {
            continue;
        }

        // e8 0c 00 00 00 call nt!KiSetTimerEx (fffff807`70520a50)
        if (*(PCHAR)(ul_KeSetTimer + i) == 0xe8)
        {

```

```

        b_e8 = TRUE;
        ul_e8Addr = ul_KeSetTimer + i;
        break;
    }
}

// 找到 call 则解析目的地址
if (b_e8 == TRUE)
{
    if (!MmIsAddressValid((PVOID64)ul_e8Addr))
    {
        return 0;
    }

    INT ul_callCode = *(INT*)(ul_e8Addr + 1);
    ULONG64 ul_KiSetTimerEx = ul_e8Addr + ul_callCode + 5;
    return ul_KiSetTimerEx;
}
return 0;
}

// 得到KiWaitNever地址
ULONG64 GetKiWaitNever(ULONG64 address)
{
    // 验证地址是否可读写
    if (!MmIsAddressValid((PVOID64)address))
    {
        return 0;
    }

    // 前 100 字节找 找 KiWaitNever
    for (INT i = 0; i < 100; i++)
    {
        // 48 8b 05 85 0c 51 00 | mov rax, qword ptr[nt!KiWaitNever(fffff807`70a316f8)]
        if (*(PUCHAR)(address + i) == 0x48 && *(PUCHAR)(address + i + 1) == 0x8b && *(PUCHAR)(address + i + 2) == 0x05)
        {
            ULONG64 ul_movCode = *(UINT32*)(address + i + 3);
            ULONG64 ul_movAddr = address + i + ul_movCode + 7;
            // DbgPrint("找到KiWaitNever地址: %p \n", ul_movAddr);
            return ul_movAddr;
        }
    }
}
return 0;
}

// 得到KiWaitAlways地址
ULONG64 GetKiWaitAlways(ULONG64 address)
{
    // 验证地址是否可读写
    if (!MmIsAddressValid((PVOID64)address))
    {
        return 0;
    }

```

```

}

// 前 100 字节找 找 KiWaitNever
for (INT i = 0; i < 100; i++)
{
    // 48 8b 35 6b 0e 51 00 | mov rsi,qword ptr [nt!KiWaitAlways (fffff807`70a318e8)]
    if (*(PUCHAR)(address + i) == 0x48 && *(PUCHAR)(address + i + 1) == 0x8b && *(PUCHAR)(address + i + 2) == 0x35)
    {
        ULONG64 ul_movCode = *(UINT32*)(address + i + 3);
        ULONG64 ul_movAddr = address + i + ul_movCode + 7;
        return ul_movAddr;
    }
}
return 0;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("卸载完成... \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    ULONG64 address = GetKeSetTimerEx();
    if (address != 0)
    {
        ULONG64 KiWaitNeverAddress = GetKiWaitNever(address);
        DbgPrint("KiWaitNeverAddress = %p \n", KiWaitNeverAddress);

        ULONG64 KiWaitAlwaysAddress = GetKiWaitAlways(address);
        DbgPrint("KiWaitAlwaysAddress = %p \n", KiWaitAlwaysAddress);
    }

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

运行这个程序，我们看下寻找到的地址是否与WinDBG中找到的地址一致。

The image shows two windows from a Windows debugging environment. The top window is 'DebugView on \\DESKTOP-B53PAVI (local)' showing a list of debug prints. The bottom window is 'Kernel \'com:port=\\.\pipe\com\_1,baud=115200,pipe\' - WinDbg:10.0.16299.15 AMD64' showing assembly code for the 'KiSetTimerEx' function.

#	Time	Debug Print
138	828.93640137	7674062500 - STORMINI: StorNVMe - POWER: ACTIVE
139	829.67199707	hello lyshark.com
140	829.67639160	KiWaitNeverAddress = FFFFF80310174700
141	829.68182373	KiWaitAlwaysAddress = FFFFF803101748F0
142	830.06951904	7684062500 - STORMINI: StorNVMe - POWER: IDLE

```
1: kd> uf KiSetTimerEx
nt!KiSetTimerEx:
fffff803`0fc63a60 48895c2408      mov     qword ptr [rsp+8],rbx
fffff803`0fc63a65 48896c2410      mov     qword ptr [rsp+10h],rbp
fffff803`0fc63a6a 4889742418      mov     qword ptr [rsp+18h],rsi
fffff803`0fc63a6f 57             push    rdi
fffff803`0fc63a70 4154           push    r12
fffff803`0fc63a72 4155           push    r13
fffff803`0fc63a74 4156           push    r14
fffff803`0fc63a76 4157           push    r15
fffff803`0fc63a78 4883ec50       sub     rsp,50h
fffff803`0fc63a7c 488b057d0c5100 mov     rax,qword ptr [nt!KiWaitNever (fffff803`10174700)]
fffff803`0fc63a83 488bf9         mov     rdi,rcx
fffff803`0fc63a86 488b35630e5100 mov     rsi,qword ptr [nt!KiWaitAlways (fffff803`101748f0)]
fffff803`0fc63a8d 410fb6e9       movzx   ebp,r9b
fffff803`0fc63a91 4c8bac24a0000000 mov     r13,qword ptr [rsp+0A0h]
fffff803`0fc63a99 458bf8         mov     r15d,r8d
```

## 实现枚举DPC定时器

最后将这些功能整合在一起，循环输出链表元素，并解密元素即可实现枚举当前系统DPC定时器。

代码核心API分析：

- KeNumberProcessors 得到CPU数量(内核常量)
- KeSetSystemAffinityThread 线程绑定到特定CPU上
- GetKiProcessorBlock 获得KPRCB的地址
- KeRevertToUserAffinityThread 取消绑定CPU

解密部分提取出 `KiwaitNever` 和 `KiwaitAlways` 用于解密计算，转换 `PKDPC` 对象结构，并输出即可。

```
#include <Fltkernel.h>
#include <ntddk.h>
#include <intrin.h>

#define IRP_TEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)

UNICODE_STRING name_device;      // 设备名
UNICODE_STRING name_symbol;      // 符号链接
PDEVICE_OBJECT deviceObj;        // 设备对象

typedef struct _KTIMER_TABLE_ENTRY
{
    ULONG_PTR    Lock;
    LIST_ENTRY   Entry;
```

```

    ULONG_PTR    Time;
}KTIMER_TABLE_ENTRY, *PKTIMER_TABLE_ENTRY;

typedef struct _KTIMER_TABLE
{
    ULONG_PTR      TimerExpiry[64];
    KTIMER_TABLE_ENTRY TimerEntries[256];
}KTIMER_TABLE, *PKTIMER_TABLE;

BOOLEAN get_kiwait(PULONG64 never, PULONG64 always)
{
    // 获取 KeSetTimer 地址
    ULONG64 ul_KeSetTimer = 0;
    UNICODE_STRING uc_KeSetTimer = { 0 };
    RtlInitUnicodeString(&uc_KeSetTimer, L"KeSetTimer");
    ul_KeSetTimer = (ULONG64)MmGetSystemRoutineAddress(&uc_KeSetTimer);
    if (ul_KeSetTimer == NULL)
    {
        return FALSE;
    }

    // 前 30 字节找 call 指令
    BOOLEAN b_e8 = FALSE;
    ULONG64 ul_e8Addr = 0;
    for (INT i = 0; i < 30; i++)
    {
        if (!MmIsAddressValid((PVOID64)ul_KeSetTimer))
        {
            continue;
        }

        /*
        0: kd> u nt!KeSetTimer
        nt!KeSetTimer:
        ffffffff803`0fc63a40 4883ec38      sub     rsp,38h
        ffffffff803`0fc63a44 4c89442420     mov     qword ptr [rsp+20h],r8
        ffffffff803`0fc63a49 4533c9        xor     r9d,r9d
        ffffffff803`0fc63a4c 4533c0        xor     r8d,r8d
        ffffffff803`0fc63a4f e80c000000     call   nt!KiSetTimerEx (ffffffff803`0fc63a60)
        ffffffff803`0fc63a54 4883c438      add     rsp,38h
        ffffffff803`0fc63a58 c3           ret
        ffffffff803`0fc63a59 cc           int     3
        */

        // ffffffff803`0fc63a4f e8 0c 00 00 00      call   nt!KiSetTimerEx (ffffffff803`0fc63a60)
        if (*(PUCHAR)(ul_KeSetTimer + i) == 0xe8)
        {
            b_e8 = TRUE;
            ul_e8Addr = ul_KeSetTimer + i;
            break;
        }
    }
}

```

```

// 找到 call 则解析目的地址
/*
0: kd> u nt!KiSetTimerEx l20
nt!KiSetTimerEx:
fffff803`0fc63a60 48895c2408      mov     qword ptr [rsp+8],rbx
fffff803`0fc63a65 48896c2410      mov     qword ptr [rsp+10h],rbp
fffff803`0fc63a6a 4889742418      mov     qword ptr [rsp+18h],rsi
fffff803`0fc63a6f 57              push    rdi
fffff803`0fc63a70 4154            push    r12
fffff803`0fc63a72 4155            push    r13
fffff803`0fc63a74 4156            push    r14
fffff803`0fc63a76 4157            push    r15
fffff803`0fc63a78 4883ec50        sub     rsp,50h
fffff803`0fc63a7c 488b057d0c5100  mov     rax,qword ptr [nt!KiWaitNever
(fffff803`10174700)]
fffff803`0fc63a83 488bf9          mov     rdi,rcx
*/

ULONG64 ul_KiSetTimerEx = 0;
if (b_e8 == TRUE)
{
    if (!MmIsAddressValid((PVOID64)ul_e8Addr))
    {
        return FALSE;
    }
    INT ul_callCode = *(INT*)(ul_e8Addr + 1);
    ULONG64 ul_callAddr = ul_e8Addr + ul_callCode + 5;
    ul_KiSetTimerEx = ul_callAddr;
}

// 没有 call 则直接在当前函数找
else
{
    ul_KiSetTimerEx = ul_KeSetTimer;
}

// 前 50 字节找 找 KiWaitNever 和 KiWaitAlways
/*
0: kd> u nt!KiSetTimerEx l20
nt!KiSetTimerEx:
fffff803`0fc63a60 48895c2408      mov     qword ptr [rsp+8],rbx
fffff803`0fc63a65 48896c2410      mov     qword ptr [rsp+10h],rbp
fffff803`0fc63a6a 4889742418      mov     qword ptr [rsp+18h],rsi
fffff803`0fc63a6f 57              push    rdi
fffff803`0fc63a70 4154            push    r12
fffff803`0fc63a72 4155            push    r13
fffff803`0fc63a74 4156            push    r14
fffff803`0fc63a76 4157            push    r15
fffff803`0fc63a78 4883ec50        sub     rsp,50h
fffff803`0fc63a7c 488b057d0c5100  mov     rax,qword ptr [nt!KiWaitNever
(fffff803`10174700)]
fffff803`0fc63a83 488bf9          mov     rdi,rcx

```

```

fffff803`0fc63a86 488b35630e5100 mov     rsi,qword ptr [nt!KiWaitAlways
(fffff803`101748f0)]
*/
if (!MmIsAddressValid((PVOID64)ul_KiSetTimerEx))
{
    return FALSE;
}

ULONG64 ul_arr_ret[2];          // 存放 KiWaitNever 和 KiWaitAlways 的地址
INT i_sub = 0;                  // 对应 ul_arr_ret 的下标
for (INT i = 0; i < 50; i++)
{
    // // fffff803`0fc63a7c 488b057d0c5100 mov     rax,qword ptr [nt!KiWaitNever
(fffff803`10174700)]
    if (*(PUCHAR)(ul_KiSetTimerEx + i) == 0x48 && *(PUCHAR)(ul_KiSetTimerEx + i + 1) == 0x8b
&& *(PUCHAR)(ul_KiSetTimerEx + i + 6) == 0x00)
    {
        ULONG64 ul_movCode = *(UINT32*)(ul_KiSetTimerEx + i + 3);
        ULONG64 ul_movAddr = ul_KiSetTimerEx + i + ul_movCode + 7;

        // 只拿符合条件的前两个值
        if (i_sub < 2)
        {
            ul_arr_ret[i_sub++] = ul_movAddr;
        }
    }
}
*never = ul_arr_ret[0];
*always = ul_arr_ret[1];

return TRUE;
}

BOOLEAN EnumDpc()
{
    DbgPrint("hello lyshark.com \n");

    // 获取 CPU 核心数
    INT i_cpuNum = KeNumberProcessors;
    DbgPrint("CPU核心数: %d \n", i_cpuNum);

    for (KAFFINITY i = 0; i < i_cpuNum; i++)
    {
        // 线程绑定特定 CPU
        KeSetSystemAffinityThread(i + 1);

        // 获得 KPRCB 的地址
        ULONG64 p_PRCB = (ULONG64)__readmsr(0xc0000101) + 0x20;
        if (!MmIsAddressValid((PVOID64)p_PRCB))
        {
            return FALSE;
        }
    }
}

```

```

// 取消绑定 CPU
KeRevertToUserAffinityThread();

// 计算 TimerTable 在 _KPRCB 结构中的偏移
PKTIMER_TABLE p_TimeTable = NULL;

// Windows 10 得到_KPRCB + 0x3680
p_TimeTable = (PKTIMER_TABLE)(*(PULONG64)p_PRCB + 0x3680);

// 遍历 TimerEntries[] 数组 (大小 256)
for (INT j = 0; j < 256; j++)
{
    // 获取 Entry 双向链表地址
    if (!MmIsValidAddress((PVOID64)p_TimeTable))
    {
        continue;
    }

    PLIST_ENTRY p_ListEntryHead = &(p_TimeTable->TimerEntries[j].Entry);

    // 遍历 Entry 双向链表
    for (PLIST_ENTRY p_ListEntry = p_ListEntryHead->Flink; p_ListEntry != p_ListEntryHead;
p_ListEntry = p_ListEntry->Flink)
    {
        // 根据 Entry 取 _KTIMER 对象地址
        if (!MmIsValidAddress((PVOID64)p_ListEntry))
        {
            continue;
        }

        PKTIMER p_Timer = CONTAINING_RECORD(p_ListEntry, KTIMER, TimerListEntry);

        // 硬编码取 KiWaitNever 和 KiWaitAlways
        ULONG64 never = 0, always = 0;
        if (get_KiWait(&never, &always) == FALSE)
        {
            return FALSE;
        }

        // 获取解密前的 Dpc 对象
        if (!MmIsValidAddress((PVOID64)p_Timer))
        {
            continue;
        }

        ULONG64 ul_Dpc = (ULONG64)p_Timer->Dpc;
        INT i_Shift = (*((PULONG64)never) & 0xFF);

        // 解密 Dpc 对象
        ul_Dpc ^= *((ULONG_PTR*)never); // 异或
        ul_Dpc = _rotl64(ul_Dpc, i_Shift); // 循环左移
        ul_Dpc ^= (ULONG_PTR)p_Timer; // 异或
        ul_Dpc = _byteswap_uint64(ul_Dpc); // 颠倒顺序
    }
}

```

```

        ul_Dpc ^= *((ULONG_PTR*)always);    // 异或

        // 对象类型转换
        PKDPC p_Dpc = (PKDPC)ul_Dpc;

        // 打印验证
        if (!MmIsValidAddress((PVOID64)p_Dpc))
        {
            continue;
        }

        DbgPrint("定时器对象: 0x%p | 函数入口: 0x%p | 触发周期: %d \n ", p_Timer, p_Dpc->DeferredRoutine, p_Timer->Period);
    }
}

return TRUE;
}

// 对应 IRP_MJ_DEVICE_CONTROL
NTSTATUS myIrpControl(IN PDEVICE_OBJECT pDevObj, IN PIRP pIRP)
{
    // 获取 IRP 对应的 I/O 堆栈指针
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIRP);

    // 得到输入缓冲区大小
    ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;

    // 得到输出缓冲区大小
    ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;

    // 得到 IOCTL 码
    ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;

    // 捕获 I/O 操作类型 (MajorFunction)
    switch (code)
    {
    {
        case IRP_TEST:
        {
            break;
        }
        default:
            break;
    }
    }

    // 完成 IO 请求
    IoCompleteRequest(pIRP, IO_NO_INCREMENT);

    return STATUS_SUCCESS;
}

// 对应 IRP_MJ_CREATE 、 IRP_MJ_CLOSE

```

```

NTSTATUS dpc_CAC(IN PDEVICE_OBJECT pDevObj, IN PIRP pIRP)
{
    // 将 IRP 返回给 I/O 管理器
    IoCompleteRequest(
        pIRP,          // IRP 指针
        IO_NO_INCREMENT // 线程优先级, IO_NO_INCREMENT : 不增加优先级
    );

    // 设置 I/O 请求状态
    pIRP->IoStatus.Status = STATUS_SUCCESS;

    // 设置 I/O 请求传输的字节数
    pIRP->IoStatus.Information = 0;

    return STATUS_SUCCESS;
}

NTSTATUS CreateDevice(IN PDRIVER_OBJECT DriverObject)
{
    // 定义返回值
    NTSTATUS status;

    // 初始化设备名
    RtlInitUnicodeString(&name_device, L"\\Device\\LySharkDriver");

    // 创建设备
    status = IoCreateDevice(
        DriverObject,          // 指向驱动对象的指针
        0,                    // 设备扩展分配的字节数
        &name_device,          // 设备名
        FILE_DEVICE_UNKNOWN,   // 设备类型
        0,                    // 驱动设备附加信息
        TRUE,                 // 设备对象是否独占设备
        &deviceObj             // 设备对象指针
    );

    if (!NT_SUCCESS(status))
    {
        return status;
    }

    // 初始化符号链接名
    RtlInitUnicodeString(&name_symbol, L"\\??\\LySharkDriver");

    // 创建符号链接
    status = IoCreateSymbolicLink(&name_symbol, &name_device);
    if (!NT_SUCCESS(status))
    {
        return status;
    }

    return STATUS_SUCCESS;
}

```

```

NTSTATUS DriverUnload(IN PDRIVER_OBJECT DriverObject)
{
    // 定义返回值
    NTSTATUS status;

    // 删除符号链接
    status = IoDeleteSymbolicLink(&name_symbol);
    if (!NT_SUCCESS(status))
    {
        return status;
    }

    // 删除设备
    IoDeleteDevice(deviceObj);

    return STATUS_SUCCESS;
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)
{
    // 定义返回值
    NTSTATUS status;

    // 指定驱动卸载函数
    DriverObject->DriverUnload = (PDRIVER_UNLOAD)DriverUnload;

    // 指定派遣函数
    DriverObject->MajorFunction[IRP_MJ_CREATE] = dpc_CAC;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = dpc_CAC;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = myIrpControl;

    // 创建设备
    status = CreateDevice(DriverObject);
    if (!NT_SUCCESS(status))
    {
        return status;
    }

    // 执行枚举
    EnumDpc();

    return STATUS_SUCCESS;
}

```

最终运行枚举程序，你将会看到系统中所有的定时器，与ARK工具对比是一致的。

进程

驱动模块

内核层

内核钩子

应用层钩子

设置

监控

启动信息

注册表

服务

文件

网络

调试引擎

系统回调

过滤驱动

DPC定时器

IO定时器

系统线程

卸载的驱动

定时器对象	DPC	触发周期(s)	函数入口	模块路径
0xFFFFB38370F396C8	0xFFFFB38370F39708	0	0xFFFFF80621F7EB80	C:\Windows\system32\ntoskrnl.exe
0xFFFFB38370F3C178	0xFFFFB38370F3C1B8	0	0xFFFFF80621F7EB80	C:\Windows\system32\ntoskrnl.exe
0xFFFFB38370F39B98	0xFFFFB38370F39BD8	0	0xFFFFF80621F7EB80	C:\Windows\system32\ntoskrnl.exe
0xFFFFF8062230B4C0	0xFFFFF8062230B500	0	0xFFFFF80622757630	C:\Windows\system32\ntoskrnl.exe

DebugView on \\DESKTOP-B53PAVI (local)

File Edit Capture Options Computer Help

#	Time	Debug Print
113	29.17917442	hello lyshark.com
114	29.18279076	CPU核心数: 2
115	29.18798065	定时器对象: 0xFFFFB383721AF048   函数入口: 0xFFFFF80625CB13E0   触发周期: 2000
116	29.18798065	
117	29.19336319	定时器对象: 0xFFFFB383704E0EC8   函数入口: 0xFFFFF80621F332C0   触发周期: 0
118	29.19336319	
119	29.19865990	定时器对象: 0xFFFFF806268FB1E0   函数入口: 0xFFFFF806268F1170   触发周期: 0
120	29.19865990	
121	29.20407677	定时器对象: 0xFFFFF80624D323E0   函数入口: 0xFFFFF80624D24410   触发周期: 0
122	29.20407677	
123	29.21896362	定时器对象: 0xFFFFF806251806E8   函数入口: 0xFFFFF806251777F0   触发周期: 0
124	29.21896362	

