在笔者上一篇文章 《驱动开发：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。



DPC定时器的作用是什么？

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 ffffbe81`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。

: \_ULARGE\_INTEGER 0x00000001`a8030353

: 0

: \_LIST\_ENTRY [ 0xffffa707`a0d3e1a0 -

+0x000 Lock

+0x008 Entry 0xffffa707`a0d3e1a0 ]

+0x018 Time

lyshark.com 0: kd> dt \_KTIMER\_TABLE\_ENTRY 0xfffff807`6f77c180 + 0x3680 + 0x200

ntdll!\_KTIMER\_TABLE\_ENTRY

: \_ULARGE\_INTEGER 0x0

: 0

: \_LIST\_ENTRY [ 0x00000000`00000000 -

+0x000 Lock

+0x008 Entry 0x00000000`00000000 ]

+0x018 Time

lyshark.com 0: kd> dt \_KTIMER\_TABLE\_ENTRY 0xfffff807`6f77c180 + 0x3680

ntdll!\_KTIMER\_TABLE\_ENTRY

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:

0: kd> u nt!KiSetTimerEx l50

nt!KiSetTimerEx:

fffff803`0fc63a78 4883ec50 fffff803`0fc63a7c 488b057d0c5100 (fffff803`10174700)] fffff803`0fc63a83 488bf9 fffff803`0fc63a86 488b35630e5100 (fffff803`101748f0)]

fffff803`0fc63a8d 410fb6e9

sub

mov

rsp,50h

rax,qword ptr [nt!KiWaitNever

mov

mov

rdi,rcx

rsi,qword ptr [nt!KiWaitAlways

movzx

fffff803`0fc63a91 4c8bac24a0000000 mov

fffff803`0fc63a99 458bf8 mov

fffff803`0fc63a9c 4933f5 xor

fffff803`0fc63a9f 488bda mov

fffff803`0fc63aa2 480fce bswap

fffff803`0fc63aa5 4833f1 xor

ebp,r9b

r13,qword ptr [rsp+0A0h] r15d,r8d

rsi,r13

rbx,rdx

rsi rsi,rcx

|  |  |  |  |
| --- | --- | --- | --- |
| 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  fffff803`0fc63a58 | 4883c438  c3 | add  ret | rsp,38h |
| fffff803`0fc63a59 | cc | int | 3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| fffff803`0fc63a60 fffff803`0fc63a65  fffff803`0fc63a6a | 48895c2408  48896c2410  4889742418 | mov mov  mov | qword qword  qword | ptr ptr  ptr | [rsp+8],rbx [rsp+10h],rbp  [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`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,al

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],bpl 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 bpl,0FCh fffff803`0fc63b25 0f85a3000000 jne nt!KiSetTimerEx+0x16e (fffff803`0fc63bce)

fffff803`0fc63b2b 48894c2420 mov qword ptr [rsp+20h],rcx fffff803`0fc63b30 48b80800000080f7ffff mov rax,0FFFFF78000000008h 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,al 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 e8209ffdff call nt!KiInsertTimerTable (fffff803`0fc3da90)

fffff803`0fc63b70 84c0 test al,al

fffff803`0fc63b72 0f8495000000

(fffff803`0fc63c0d)

je

nt!KiSetTimerEx+0x1ad

fffff803`0fc63b78 f7058608510000000200 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

fffff803`0fc63b97 4533c9 fffff803`0fc63b9a 33d2 fffff803`0fc63b9c 88442420 fffff803`0fc63ba0 498bce fffff803`0fc63ba3 458d4101 fffff803`0fc63ba7 e8044efeff

(fffff803`0fc489b0)

xor xor mov mov lea

call

rax,qword ptr [rsp+0A0h] r9d,r9d

edx,edx

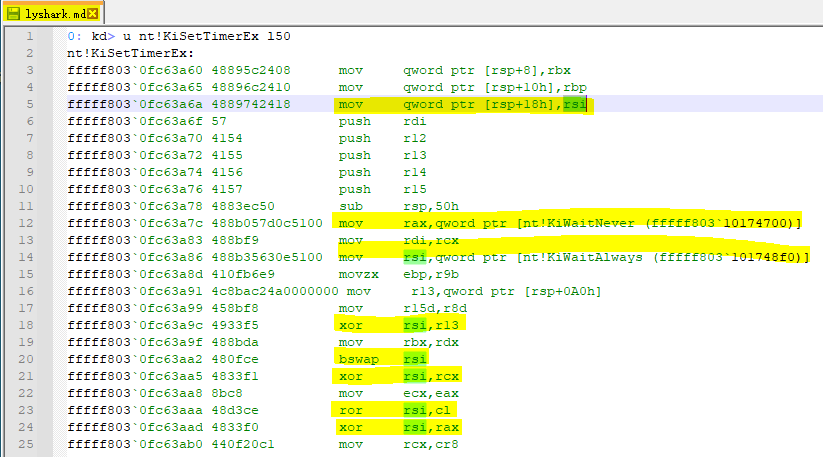
byte ptr [rsp+20h],al rcx,r14

r8d,[r9+1]

nt!KiExitDispatcher

# 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。



# 将解密流程通过代码的方式实现。



#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);

int64)Timer ^ \_RBX, KiWaitNever); ptrDpc ^= (ULONG\_PTR)ptrTimer; ptrDpc = \_byteswap\_uint64(ptrDpc); ptrDpc ^= p2dq(ptrKiWaitAlways);

KiWaitAlways;

// v18 = ROR8 ((unsigned

// asm { bswap

rbx }

// \_RBX = (unsigned int64)DPC ^

// 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。

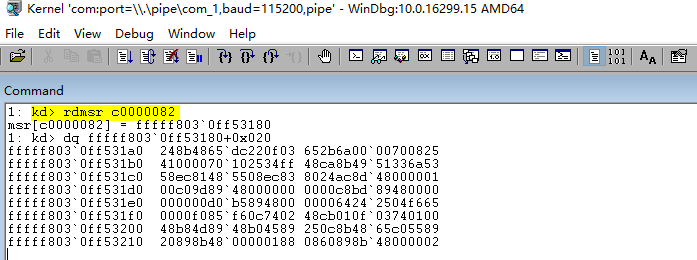
ULONG\_PTR ptrKiProcessorBlock = 0xfffff80770a32cc0; ULONG\_PTR ptrOffsetKTimerTable = 0x3680;

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。

**第一步：** 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。



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 ffffbe81`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)

{

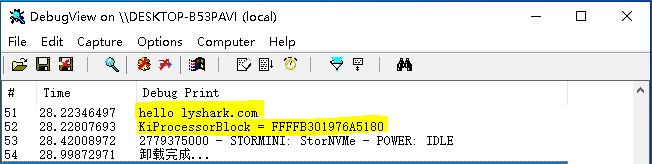
DbgPrint("KiProcessorBlock = %p \n", address);

}

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

# 运行后即可得到输出效果如下：



**第二步：** 找到 KeSetTimer 从里面搜索特征得到 call KeSetTimerEx 函数地址，还记得 《驱动开发：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

fffff807`70520a34 4c89442420

fffff807`70520a39 4533c9 fffff807`70520a3c 4533c0 fffff807`70520a3f e80c000000 fffff807`70520a44 4883c438 fffff807`70520a48 c3

/

sub mov xor xor call add

ret

rsp,38h

qword ptr [rsp+20h],r8 r9d,r9d

r8d,r8d

nt!KiSetTimerEx (fffff807`70520a50) rsp,38h

#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 ( (PUCHAR)(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)

{

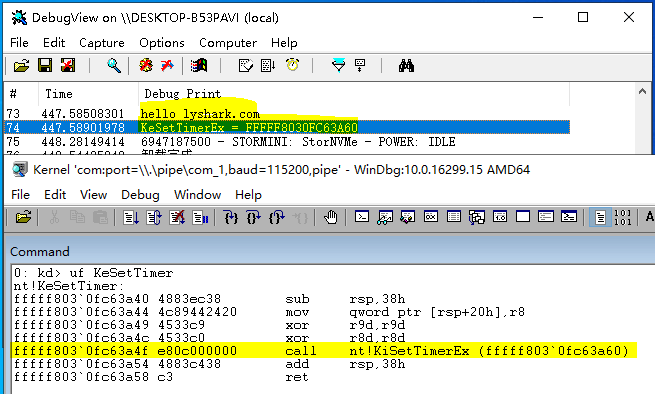
DbgPrint("KeSetTimerEx = %p \n", address);

}

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

# 输出寻找CALL地址效果图如下：



**第三步：** 也是最重要的一步，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`70520a68 4883ec50 fffff807`70520a6c 488b05850c5100

(fffff807`70a316f8)] fffff807`70520a73 488bf9 fffff807`70520a76 488b356b0e5100

(fffff807`70a318e8)] fffff807`70520a7d 410fb6e9

/

sub

mov

rsp,50h

rax,qword ptr [nt!KiWaitNever

mov

mov

rdi,rcx

rsi,qword ptr [nt!KiWaitAlways

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;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 |  |  |

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 ( (PUCHAR)(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);

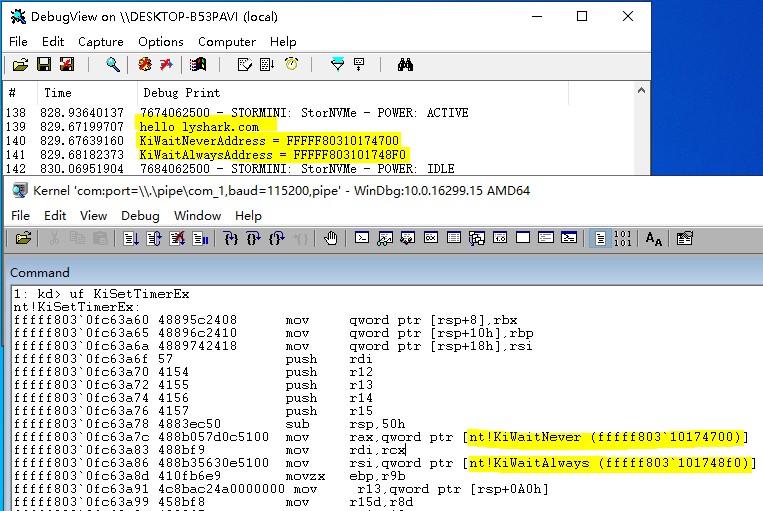
ULONG64 KiWaitAlwaysAddress = GetKiWaitAlways(address); DbgPrint("KiWaitAlwaysAddress = %p \n", KiWaitAlwaysAddress);

}

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。



**功能实现部分：** 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。

// 署名

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#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:

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

/

// fffff803`0fc63a4f e8 0c 00 00 00 call nt!KiSetTimerEx (fffff803`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 fffff803`0fc63a65  fffff803`0fc63a6a | 48895c2408  48896c2410  4889742418 | mov mov  mov | qword qword  qword | ptr ptr  ptr | [rsp+8],rbx [rsp+10h],rbp  [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 (!MmIsAddressValid((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 (!MmIsAddressValid((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 (!MmIsAddressValid((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 (!MmIsAddressValid((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;

}

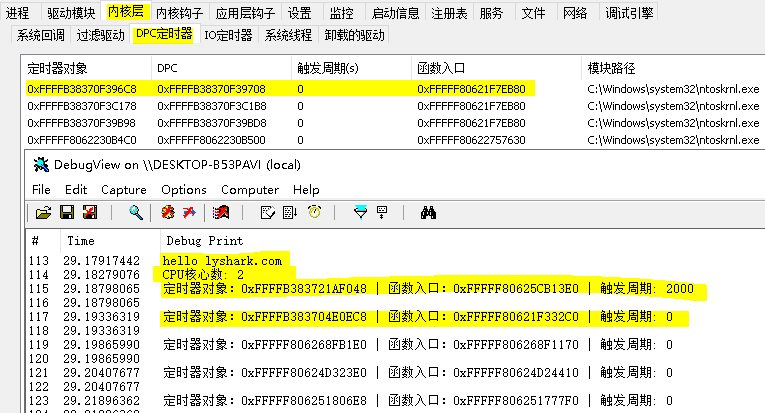
// 执行枚举

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。



**参考文献**

# h[ttps://www.cnblogs.com/kuangke/p/9397511.htm](http://www.cnblogs.com/kuangke/p/9397511.html)l

作者： 王瑞 (LyShark)

作者邮箱： m [e@lyshark.com](mailto:e@lyshark.com)

版权声明：本博客文章与代码均为学习时整理的笔记，文章 [均为原创] 作品，转载文章请遵守

《中华人民共和国著作权法》相关法律规定或遵守《署名CC BY-ND 4.0国际》规范，合理合规携带原创出处转载，如果不携带文章出处，并恶意转载多篇原创文章被本人发现，本人保留起诉权！