

在上一篇文章《内核枚举DpcTimer定时器》中我们通过枚举特征码的方式找到了 DPC 定时器基址并输出了内核中存在的定时器列表，本章将学习如何通过特征码定位的方式寻找 Windows 10 系统下面的 PspCidTable 内核句柄表地址。

首先引入一段基础概念：

- 1. 在 windows 下所有的资源都是用对象的方式进行管理的（文件、进程、设备等都是对象），当要访问一个对象时，如打开一个文件，系统就会创建一个对象句柄，通过这个句柄可以对这个文件进行各种操作。
- 2. 句柄和对象的联系是通过句柄表来进行的，准确来说一个句柄就是它所对应的对象在句柄表中的索引。
- 3. 通过句柄可以在句柄表中找到对象的指针，通过指针就可以对，对象进行操作。

PspCidTable 就是这样的一种表（内核句柄表），表的内部存放的是 进程 EPROCESS 和 线程ETHREAD 的内核对象，并通过 进程 PID 和 线程 TID 进行索引，ID号以4递增，内核句柄表不属于任何进程，也不连接在系统的句柄表上，通过它可以返回系统的任何对象。

内核句柄表与普通句柄表完全一样，但它与每个进程私有的句柄表有以下不同：

- 1.PspCidTable 中存放的对象是系统中所有的进程线程对象，其索引就是 PID 和 TID 。
- 2.PspCidTable 中存放的直接是对象体 EPROCESS 和ETHREAD，而每个进程私有的句柄表则存放的是对象头 OBJECT\_HEADER 。
- 3.PspCidTable 是一个独立的句柄表，而每个进程私有的句柄表以一个双链连接起来。
- 4.PspCidTable 访问对象时要掩掉低三位，每个进程私有的句柄表是双链连接起来的。

那么在 windows10 系统中该如何枚举句柄表；

- 1.首先找到 PsLookupProcessByProcessId 函数地址，该函数是被导出的可以动态拿到。
- 2.其次在 PsLookupProcessByProcessId 地址中搜索 PspReferenceCidTableEntry 函数。
- 3.最后在 PspReferenceCidTableEntry 地址中找到 PspCidTable 函数。

首先第一步先要得到 PspCidTable 函数内存地址，输入 dp PspCidTable 即可得到，如果在程序中则是调用 MmGetSystemRoutineAddress 取到。

#### Command

```
0: kd> dp PspCidTable
fffff802`08374530  fffffdc88`79605dc0  fffffb281`de2b07a0
fffff802`08374540  00000000`00000000  00010000`00000000
fffff802`08374550  0001017c`00001000  00000000`00009e0a
fffff802`08374560  00000000`00000000  00000000`00000000
fffff802`08374570  00000000`00000000  00000000`00000000
fffff802`08374580  0000002e`00000000  00000000`00000000
fffff802`08374590  fffffb281`de2b1c40  00000000`00000000
fffff802`083745a0  00000000`00000000  00000000`00000000
0: kd> uf PspCidTable
Flow analysis was incomplete, some code may be missing
nt!PspCidTable:
fffff802`08374530 c05d6079          rcr      byte ptr [rbp+60h],79h
fffff802`08374534 88dc            mov      ah,bl
```

PspCidTable是一个 HANDLE\_TABLE 结构，当新建一个进程时，对应的会在 PspCidTable 存在一个该进程和线程对应的 HANDLE\_TABLE\_ENTRY 项。在 windows10 中依然采用 动态扩展 的方法，当句柄数少的时候就采用下层表，多的时候才启用中层表或上层表。

接着我们解析 fffffdc88-79605dc0 这个内存地址，执行 dt \_HANDLE\_TABLE 0xfffffdc8879605dc0 得到规范化结构体。

```

Command
0: kd> dt _HANDLE_TABLE 0xfffffdc8879605dc0
ntdll!_HANDLE_TABLE
+0x000 NextHandleNeedingPool : 0x1c00
+0x004 ExtraInfoPages : 0n0
+0x008 TableCode : 0xfffffdc88`7d09b001
+0x010 QuotaProcess : (null)
+0x018 HandleTableList : _LIST_ENTRY [ 0xfffffdc88`79605dd8 - 0xfffffdc88`79605dd8 ]
+0x028 UniqueProcessId : 0
+0x02c Flags : 1
+0x02c StrictFIFO : 0y1
+0x02c EnableHandleExceptions : 0y0
+0x02c Rundown : 0y0
+0x02c Duplicated : 0y0
+0x02c RaiseUMExceptionOnInvalidHandleClose : 0y0
+0x030 HandleContentionEvent : _EX_PUSH_LOCK
+0x038 HandleTableLock : _EX_PUSH_LOCK
+0x040 FreeLists : [1] _HANDLE_TABLE_FREE_LIST
+0x040 ActualEntry : [32] ""
+0x060 DebugInfo : (null)

```

内核句柄表分为三层如下：

- 下层表：是一个 `HANDLE_TABLE_ENTRY` 项的索引，整个表共有 256 个元素，每个元素是一个 8 个字节长的 `HANDLE_TABLE_ENTRY` 项及索引，`HANDLE_TABLE_ENTRY` 项中保存着指向对象的指针，下层表可以看成是进程和线程的稠密索引。
- 中层表：共有 256 个元素，每个元素是 4 个字节长的指向下层表的入口指针及索引，中层表可以看成是进程和线程的稀疏索引。
- 上层表：共有 256 个元素，每个元素是 4 个字节长的指向中层表的入口指针及索引，上层表可以看成是中层表的稀疏索引。

总结起来一个句柄表有一个上层表，一个上层表最多可以有 256 个中层表的入口指针，每个中层表最多可以有 256 个下层表的入口指针，每个下层表最多可以有 256 个进程和线程对象的指针。`PspCidTable` 表可以看成是 `HANDLE_TABLE_ENTRY` 项的多级索引。

如上图所示 `TableCode` 是指向句柄表的指针，低二位（二进制）记录句柄表的等级：0 (00) 表示一级表，1 (01) 表示二级表，2 (10) 表示三级表。这里的 `0xfffffdc88-7d09b001` 就说明它是一个二级表。

### Command

```
0: kd> dp PspCidTable
fffff802`08374530  fffffdc88`79605dc0  fffffb281`de2b07a0
fffff802`08374540  00000000`00000000  00010000`00000000
fffff802`08374550  0001017c`00001000  00000000`00009e0a
fffff802`08374560  00000000`00000000  00000000`00000000
fffff802`08374570  00000000`00000000  00000000`00000000
fffff802`08374580  00000002e`00000000  00000000`00000000
fffff802`08374590  ffffdb281`de2b1c40  00000000`00000000
fffff802`083745a0  00000000`00000000  00000000`00000000
0: kd> dt _HANDLE_TABLE fffffdc88`79605dc0
ntdll!_HANDLE_TABLE
+0x000 NextHandleNeedingPool : 0x1c00
+0x004 ExtraInfoPages : 0n0
+0x008 TableCode : 0xfffffd88`7d09b001
+0x010 QuotaProcess : (null)
+0x018 HandleTableList : _LIST_ENTRY [ 0xfffffd88`79605dd8 - 0xfffffd88`79605dd8 ]
+0x028 UniqueProcessId : 0
+0x02c Flags : 1
+0x02c StrictFIFO : 0y1
+0x02c EnableHandleExceptions : 0y0
+0x02c Rundown : 0y0
+0x02c Duplicated : 0y0
+0x02c RaiseUMExceptionOnInvalidHandleClose : 0y0
+0x030 HandleContentionEvent : _EX_PUSH_LOCK
+0x038 HandleTableLock : _EX_PUSH_LOCK
+0x040 FreeLists : [1]_HANDLE_TABLE_FREE_LIST
+0x040 ActualEntry : [32] ""
+0x060 DebugInfo : (null)
```

一级表里存放的就是进程和线程对象（加密过的，需要一些计算来解密），二级表里存放的是指向某个一级表的指针，同理三级表存放的是指向二级表的指针。

x64 系统中，每张表的大小是 0x1000 (4096)，一级表中存放的是 \_handle\_table\_entry 结构（大小 = 16），二级表和三级表存放的是指针（大小 = 8）。

我们对 0xfffffd88-7d09b001 抹去低二位，输入 dp 0xfffffd887d09b000 输出的结果就是一张二级表，里面存储的就是一级表指针。

### Command

```
0: kd> dp 0xfffffd887d09b000
fffffdc88`7d09b000  fffffdc88`7962a000  fffffdc88`7d09c000
fffffdc88`7d09b010  fffffdc88`7d4ff000  fffffdc88`7dcff000
fffffdc88`7d09b020  fffffdc88`7e34f000  fffffdc88`7ebf7000
fffffdc88`7d09b030  fffffdc88`7f2f5000  00000000`00000000
fffffdc88`7d09b040  00000000`00000000  00000000`00000000
fffffdc88`7d09b050  00000000`00000000  00000000`00000000
fffffdc88`7d09b060  00000000`00000000  00000000`00000000
fffffdc88`7d09b070  00000000`00000000  00000000`00000000
```

继续查看第一张一级表，输入 dp 0xfffffd887962a000 命令，我们知道一级句柄表是根据进程或线程ID来索引的，且以4累加，所以第一行对应 id = 0，第二行对应 id = 4。根据尝试， PID = 4 的进程是 System。

### Command

```
0: kd> dp 0xfffffdc88`7d09b000
fffffdc88`7d09b000 fffffdc88`7962a000 fffffdc88`7d09c000
fffffdc88`7d09b010 fffffdc88`7d4ff000 fffffdc88`7dcff000
fffffdc88`7d09b020 fffffdc88`7e34f000 fffffdc88`7ebf7000
fffffdc88`7d09b030 fffffdc88`7f2f5000 00000000`00000000
fffffdc88`7d09b040 00000000`00000000 00000000`00000000
fffffdc88`7d09b050 00000000`00000000 00000000`00000000
fffffdc88`7d09b060 00000000`00000000 00000000`00000000
fffffdc88`7d09b070 00000000`00000000 00000000`00000000
0: kd> dp fffffdc88`7962a000
fffffdc88`7962a000 00000000`00000000 00000000`00000000
fffffdc88`7962a010 b281de28`3300ffa7 00000000`00000000
fffffdc88`7962a020 00000000`00000000 fffffdc88`7e34f800
fffffdc88`7962a030 b281de29`50800001 00000000`00000000
fffffdc88`7962a040 b281de27`d0400001 00000000`00000000
fffffdc88`7962a050 b281de2b`e0800001 00000000`00000000
fffffdc88`7962a060 b281de35`70800001 00000000`00000000
fffffdc88`7962a070 b281de3c`11400001 00000000`00000000
```

所以此处的第二行 `0xb281de28-3300ffa7` 就是加密后的 system 进程的 EPROCESS 结构，对于 Win10 系统来说解密算法 (`value >> 0x10` & `0xfffffffffffffff0`) 是这样的，我们通过代码计算出来。

```
#include <windows.h>
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{
    std::cout << "hello lyshark.com" << std::endl;

    ULONG64 ul_recode = 0xb281de283300ffa7;

    ULONG64 ul_decode = (ULONG64)ul_recode >> 0x10;
    ul_decode &= 0xfffffffffffffff0;

    std::cout << "解密后地址：" << std::hex << ul_decode << std::endl;
    getchar();

    return 0;
}
```

运行程序得到如下输出，即可知道 system 系统进程解密后的 EPROCESS 结构地址是 `0xfffffb281de283300`

```
c:\users\admin\documents\visual studio 2013\Projects\ConsoleApplication1\x64\Release\ConsoleApplication1.exe
hello lyshark.com
解密后地址：fffffb281de283300
```

回到 WinDBG 调试器，输入命令 `dt _EPROCESS 0xfffffb281de283300` 解析以下这个结构，输出结果是 System 进程。

## Command

```
+0x420 DebugPort      : (null)
+0x428 WoW64Process   : (null)
+0x430 DeviceMap       : 0xfffffd88`79613540 Void
+0x438 EtwDataSource    : (null)
+0x440 PageDirectoryPte : 0
+0x448 ImageFilePointer : (null)
+0x450 ImageFileName    : [15] "System"
+0x45f PriorityClass    : 0x2 ''
+0x460 SecurityPort     : (null)
+0x468 SeAuditProcessCreateInfo : _SE_AUDIT_PROCESS_CREATION_INFO
+0x470 JobLinks        : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`00000000 ]
+0x480 HighestUserAddress : 0x00007fff`ffff0000 Void
+0x488 ThreadListHead   : _LIST_ENTRY [ 0xfffffb281`de295738 - 0xfffffb281`e169e738 ]
+0x498 ActiveThreads    : 0x8b
+0x49c ImagePathHash     : 0
+0x4a0 DefaultHardErrorProcessing : 5
+0x4a4 LastThreadExitStatus : 0n0
+0x4a8 PrefetchTrace   : _EX_FAST_REF
+0x4b0 LockedPagesList   : (null)
+0x4b8 ReadOperationCount : _LARGE_INTEGER 0x14
+0x4c0 WriteOperationCount : _LARGE_INTEGER 0x17
+0x4c8 OtherOperationCount : _LARGE_INTEGER 0x294
+0x4d0 ReadTransferCount : _LARGE_INTEGER 0x13a0
+0x4d8 WriteTransferCount : _LARGE_INTEGER 0xd304cb
+0x4e0 OtherTransferCount : _LARGE_INTEGER 0x37b2
```

理论知识总结已经结束了，接下来就是如何实现枚举进程线程了，枚举流程如下：

- 1.首先找到 `PspCidTable` 的地址。
- 2.然后找到 `HANDLE_TBALE` 的地址。
- 3.根据 `TableCode` 来判断层次结构。
- 4.遍历层次结构来获取对象地址。
- 5.判断对象类型是否为进程对象。
- 6.判断进程是否有效。

这里先来实现获取 `PspCidTable` 函数的动态地址，代码如下。

```
#include <ntifs.h>
#include <windef.h>

// 获取 PspCidTable
BOOLEAN get_PspCidTable(ULONG64* tableAddr)
{
    // 获取 PsLookupProcessByProcessId 地址
    UNICODE_STRING uc_funcName;
    RtlInitUnicodeString(&uc_funcName, L"PsLookupProcessByProcessId");
    ULONG64 ul_funcAddr = MmGetSystemRoutineAddress(&uc_funcName);
    if (ul_funcAddr == NULL)
    {
        return FALSE;
    }
    DbgPrint("PsLookupProcessByProcessId addr = %p \n", ul_funcAddr);

    // 前 40 字节有 call (PspReferenceCidTableEntry)
    /*
    0: kd> uf PsLookupProcessByProcessId
        nt!PsLookupProcessByProcessId:
        fffff802`0841cf0 48895c2418      mov     qword ptr [rsp+18h],rbx
```

```

fffff802`0841cfe5 56      push    rsi
fffff802`0841cfe6 4883ec20   sub     rsp,20h
fffff802`0841cfed 48897c2438   mov     qword ptr [rsp+38h],rdi
fffff802`0841cfef 488bf2      mov     rsi,rdx
fffff802`0841cff2 65488b3c2588010000   mov     rdi,qword ptr gs:[188h]
fffff802`0841cffb 66ff8fe6010000   dec     word ptr [rdi+1E6h]
fffff802`0841d002 b203      mov     dl,3
fffff802`0841d004 e887000000   call    nt!PspReferenceCidTableEntry
(fffff802`0841d090)
fffff802`0841d009 488bd8      mov     rbx,rax
fffff802`0841d00c 4885c0      test    rax,rax
fffff802`0841d00f 7435      je     nt!PsLookupProcessByProcessId+0x66
(fffff802`0841d046) Branch
*/
ULONG64 ul_entry = 0;
for (INT i = 0; i < 100; i++)
{
    // fffff802`0841d004 e8 87 00 00 00      call    nt!PspReferenceCidTableEntry
(fffff802`0841d090)
    if (*(PUCHAR)(ul_funcAddr + i) == 0xe8)
    {
        ul_entry = ul_funcAddr + i;
        break;
    }
}

if (ul_entry != 0)
{
    // 解析 call 地址
    INT i_callCode = *(INT*)(ul_entry + 1);
    DbgPrint("i_callCode = %p \n", i_callCode);
    ULONG64 ul_callJmp = ul_entry + i_callCode + 5;
    DbgPrint("ul_callJmp = %p \n", ul_callJmp);

    // 来到 call (PspReferenceCidTableEntry) 内找 PspCidTable
/*
0: kd> uf PspReferenceCidTableEntry
    nt!PspReferenceCidTableEntry+0x115:
    fffff802`0841d1a5 488b0d8473f5ff  mov     rcx,qword ptr [nt!PspCidTable
(fffff802`08374530)]
    fffff802`0841d1ac b801000000  mov     eax,1
    fffff802`0841d1b1 f0480fc107  lock   xadd qword ptr [rdi],rax
    fffff802`0841d1b6 4883c130  add    rcx,30h
    fffff802`0841d1ba f0830c2400  lock   or dword ptr [rsp],0
    fffff802`0841d1bf 48833900  cmp    qword ptr [rcx],0
    fffff802`0841d1c3 0f843fffffff  je     nt!PspReferenceCidTableEntry+0x78
(fffff802`0841d108) Branch
*/
for (INT i = 0; i < 0x120; i++)
{
    // fffff802`0841d1a5 48 8b 0d 84 73 f5 ff  mov     rcx,qword ptr [nt!PspCidTable
(fffff802`08374530)]

```

```

        if (*((PUCHAR)(ul_calljmp + i) == 0x48 && *((PUCHAR)(ul_calljmp + i + 1) == 0x8b
&& *(PUCHAR)(ul_calljmp + i + 2) == 0xd0)
    {
        // 解析 mov 地址
        INT i_movCode = *(INT*)(ul_calljmp + i + 3);
        DbgPrint("i_movCode = %p \n", i_movCode);
        ULONG64 ul_movJmp = ul_calljmp + i + i_movCode + 7;
        DbgPrint("ul_movJmp = %p \n", ul_movJmp);

        // 得到 PspCidTable
        *tableAddr = ul_movJmp;
        return TRUE;
    }
}

return FALSE;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint(("Uninstall Driver Is OK \n"));
}

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

    ULONG64 tableAddr = 0;

    get_PspCidTable(&tableAddr);

    DbgPrint("PspCidTable Address = %p \n", tableAddr);

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

运行后即可得到动态地址，我们可以验证一下是否一致：

#	Time	Debug Print
1	0.00000000	4071250000 - STORMINI: StorNVMe - POWER: ACTIVE
2	0.00753130	4071250000 - STORMINI: StorNVMe - POWER: IDLE
3	0.01759370	4071406250 - STORMINI: StorNVMe - POWER: ACTIVE
4	1.05251265	hello lshark
5	1.05732489	PsLookupProcessByProcessId addr = FFFFF8020841CFE0
6	1.06266582	i_callCode = 00000000000000087
7	1.06803918	ul_callJmp = FFFF8020841D090
8	1.07343614	i_movCode = 00000000FF57454
9	1.07873988	ul_movJmp = FFFF80208374530
10	1.08401048	PspCidTable Address = FFFF80208374530
11	1.17620623	4081406250 - STORMINI: StorNVMe - POWER: IDLE

Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:10.0.16299.15 AMD64	
File	Edit View Debug Window Help
Command	

```
0: kd> uf PspCidTable
Flow analysis was incomplete, some code may be missing
nt!PspCidTable:
fffff802`08374530 c05d6079      rcr      byte ptr [rbp+60h],79h
fffff802`08374534 88dc          mov      ah,bl
```

继续增加对与三级表的动态解析代码，最终代码如下所示：

```
#include <ntifs.h>
#include <windef.h>

// 获取 PspCidTable
BOOLEAN get_PspCidTable(ULONG64* tableAddr)
{
    // 获取 PsLookupProcessByProcessId 地址
    UNICODE_STRING uc_funcName;
    RtlInitUnicodeString(&uc_funcName, L"PsLookupProcessByProcessId");
    ULONG64 ul_funcAddr = MmGetSystemRoutineAddress(&uc_funcName);
    if (ul_funcAddr == NULL)
    {
        return FALSE;
    }
    DbgPrint("PsLookupProcessByProcessId addr = %p \n", ul_funcAddr);

    // 前 40 字节有 call (PspReferenceCidTableEntry)
    /*
    0: kd> uf PsLookupProcessByProcessId
    nt!PsLookupProcessByProcessId:
    fffff802`0841cf0 48895c2418      mov      qword ptr [rsp+18h],rbx
    fffff802`0841cf05 56              push     rsi
    fffff802`0841cf06 4883ec20      sub     rsp,20h
    fffff802`0841cf0a 48897c2438      mov      qword ptr [rsp+38h],rdi
    fffff802`0841cfef 488bf2          mov      rsi,rdx
    fffff802`0841cff2 65488b3c2588010000 mov     rdi,qword ptr gs:[188h]
    fffff802`0841cffb 66ff8fe6010000 dec     word ptr [rdi+1E6h]
    fffff802`0841d002 b203          mov     dl,3
```

```

fffff802`0841d004 e887000000      call    nt!PspReferenceCidTableEntry
(fffff802`0841d090)
fffff802`0841d009 488bd8          mov     rbx, rax
fffff802`0841d00c 4885c0          test   rax, rax
fffff802`0841d00f 7435            je     nt!PsLookupProcessByProcessId+0x66
(fffff802`0841d046)  Branch
/*
ULONG64 ul_entry = 0;
for (INT i = 0; i < 100; i++)
{
    // fffff802`0841d004 e8 87 00 00 00      call    nt!PspReferenceCidTableEntry
(fffff802`0841d090)
    if (*(PUCHAR)(ul_funcAddr + i) == 0xe8)
    {
        ul_entry = ul_funcAddr + i;
        break;
    }
}

if (ul_entry != 0)
{
    // 解析 call 地址
    INT i_callCode = *(INT*)(ul_entry + 1);
    DbgPrint("i_callCode = %p \n", i_callCode);
    ULONG64 ul_callJmp = ul_entry + i_callCode + 5;
    DbgPrint("ul_callJmp = %p \n", ul_callJmp);

    // 来到 call (PspReferenceCidTableEntry) 内找 PspCidTable
/*
0: kd> uf PspReferenceCidTableEntry
    nt!PspReferenceCidTableEntry+0x115:
    fffff802`0841d1a5 488b0d8473f5ff  mov     rcx,qword ptr [nt!PspCidTable
(fffff802`08374530)]
        fffff802`0841d1ac b801000000      mov     eax,1
        fffff802`0841d1b1 f0480fc107      lock   xadd qword ptr [rdi],rax
        fffff802`0841d1b6 4883c130      add    rcx,30h
        fffff802`0841d1ba f0830c2400      lock   or dword ptr [rsp],0
        fffff802`0841d1bf 48833900      cmp    qword ptr [rcx],0
        fffff802`0841d1c3 0f843ffffffff    je     nt!PspReferenceCidTableEntry+0x78
(fffff802`0841d108)  Branch
*/
    for (INT i = 0; i < 0x120; i++)
    {
        // fffff802`0841d1a5 48 8b 0d 84 73 f5 ff  mov     rcx,qword ptr [nt!PspCidTable
(fffff802`08374530)]
        if (*(PUCHAR)(ul_callJmp + i) == 0x48 && *(PUCHAR)(ul_callJmp + i + 1) == 0x8b
&& *(PUCHAR)(ul_callJmp + i + 2) == 0x0d)
        {
            // 解析 mov 地址
            INT i_movCode = *(INT*)(ul_callJmp + i + 3);
            DbgPrint("i_movCode = %p \n", i_movCode);
            ULONG64 ul_movJmp = ul_callJmp + i + i_movCode + 7;
            DbgPrint("ul_movJmp = %p \n", ul_movJmp);
        }
    }
}

```

```

        // 得到 PspCidTable
        *tableAddr = ul_movJmp;
        return TRUE;
    }
}
}

return FALSE;
}

/* 解析一级表
BaseAddr: 一级表的基地址
index1: 第几个一级表
index2: 第几个二级表
*/
VOID parse_table_1(ULONG64 BaseAddr, INT index1, INT index2)
{
    // 遍历一级表 (每个表项大小 16 ), 表大小 4k, 所以遍历 4096/16 = 526 次
    PEPROCESS p_eprocess = NULL;
    PETHREAD p_ethread = NULL;
    INT i_id = 0;
    for (INT i = 0; i < 256; i++)
    {
        if (!MmIsAddressValid((PVOID64)(BaseAddr + i * 16)))
        {
            DbgPrint("非法地址= %p \n", BaseAddr + i * 16);
            continue;
        }

        ULONG64 ul_recode = *(PULONG64)(BaseAddr + i * 16);

        // 解密
        ULONG64 ul_decode = (LONG64)ul_recode >> 0x10;
        ul_decode &= 0xfffffffffffffff0;

        // 判断是进程还是线程
        i_id = i * 4 + 1024 * index1 + 512 * index2 * 1024;
        if (PsLookupProcessByProcessId(i_id, &p_eprocess) == STATUS_SUCCESS)
        {
            DbgPrint("进程PID: %d | ID: %d | 内存地址: %p | 对象: %p \n", i_id, i, BaseAddr +
i * 0x10, ul_decode);
        }
        else if (PsLookupThreadByThreadId(i_id, &p_ethread) == STATUS_SUCCESS)
        {
            DbgPrint("线程TID: %d | ID: %d | 内存地址: %p | 对象: %p \n", i_id, i, BaseAddr +
i * 0x10, ul_decode);
        }
    }
}

/* 解析二级表
BaseAddr: 二级表基地址
index2: 第几个二级表
*/

```

```

*/
VOID parse_table_2(ULONG64 BaseAddr, INT index2)
{
    // 遍历二级表（每个表项大小 8），表大小 4k，所以遍历 4096/8 = 512 次
    ULONG64 ul_baseAddr_1 = 0;
    for (INT i = 0; i < 512; i++)
    {
        if (!MmIsAddressValid((PVOID64)(BaseAddr + i * 8)))
        {
            DbgPrint("非法二级表指针 (1) :%p \n", BaseAddr + i * 8);
            continue;
        }
        if (!MmIsAddressValid((PVOID64)*(PULONG64)(BaseAddr + i * 8)))
        {
            DbgPrint("非法二级表指针 (2) :%p \n", BaseAddr + i * 8);
            continue;
        }
        ul_baseAddr_1 = *(PULONG64)(BaseAddr + i * 8);
        parse_table_1(ul_baseAddr_1, i, index2);
    }
}

/* 解析三级表
BaseAddr: 三级表基地址
*/
VOID parse_table_3(ULONG64 BaseAddr)
{
    // 遍历三级表（每个表项大小 8），表大小 4k，所以遍历 4096/8 = 512 次
    ULONG64 ul_baseAddr_2 = 0;
    for (INT i = 0; i < 512; i++)
    {
        if (!MmIsAddressValid((PVOID64)(BaseAddr + i * 8)))
        {
            continue;
        }
        if (!MmIsAddressValid((PVOID64)*(PULONG64)(BaseAddr + i * 8)))
        {
            continue;
        }
        ul_baseAddr_2 = *(PULONG64)(BaseAddr + i * 8);
        parse_table_2(ul_baseAddr_2, i);
    }
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("Uninstall Driver Is OK \n");
}

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

```

```

ULONG64 tableAddr = 0;

get_PspCidTable(&tableAddr);

DbgPrint("PspCidTable Address = %p \n", tableAddr);

// 获取 _HANDLE_TABLE 的 TableCode
ULONG64 ul_tableCode = *(PULONG64)((ULONG64)*(PULONG64)tableAddr + 8);
DbgPrint("ul_tableCode = %p \n", ul_tableCode);

// 取低 2位 (二级制11 = 3)
INT i_low2 = ul_tableCode & 3;
DbgPrint("i_low2 = %x \n", i_low2);

// 一级表
if (i_low2 == 0)
{
    // TableCode 低 2位抹零 (二级制11 = 3)
    parse_table_1(ul_tableCode & (~3), 0, 0);
}

// 二级表
else if (i_low2 == 1)
{
    // TableCode 低 2位抹零 (二级制11 = 3)
    parse_table_2(ul_tableCode & (~3), 0);
}

// 三级表
else if (i_low2 == 2)
{
    // TableCode 低 2位抹零 (二级制11 = 3)
    parse_table_3(ul_tableCode & (~3));
}

else
{
    DbgPrint("LyShark提示：错误，非法！ ");
    return FALSE;
}

Driver->DriverUnload = UnDriver;
return STATUS_SUCCESS;
}

```

运行如上完整代码，我们可以在WinDBG中捕捉到枚举到的进程信息：

DebugView on \\DESKTOP-B53PAVI (local)

File Edit Capture Options Computer Help

#	Time	Debug Print
6	1.40835607	hello lyshark.com
7	1.41310668	PsLookupProcessByProcessId addr = FFFFF8020841CFE0
8	1.41846049	i_callCode = 000000000000000087
9	1.42382085	ul_callJmp = FFFFF8020841D090
10	1.42928135	i_movCode = 00000000FFF57454
11	1.43486655	ul_movJmp = FFFFF80208374530
12	1.44027793	PspCidTable Address = FFFFF80208374530
13	1.44565725	ul_tableCode = FFFFDC887D09B001
14	1.45099461	i_low2 = 1
15	1.45639431	进程PID: 4   ID: 1   内存地址: FFFFDC887962A010   对象: FFFFDB281DE283300
16	1.46178901	线程TID: 8   ID: 2   内存地址: FFFFDC887962A020   对象: FFFFDB281E240D080
17	1.46716416	线程TID: 12   ID: 3   内存地址: FFFFDC887962A030   对象: FFFFDB281DE295080
18	1.47233284	线程TID: 16   ID: 4   内存地址: FFFFDC887962A040   对象: FFFFDB281DE27D040
19	1.47776735	线程TID: 20   ID: 5   内存地址: FFFFDC887962A050   对象: FFFFDB281DE2BE080
20	1.48314285	线程TID: 24   ID: 6   内存地址: FFFFDC887962A060   对象: FFFFDB281DE357080
21	1.48850119	线程TID: 28   ID: 7   内存地址: FFFFDC887962A070   对象: FFFFDB281DE3C1140
22	1.49370909	线程TID: 32   ID: 8   内存地址: FFFFDC887962A080   对象: FFFFDB281DE335080
23	1.49914503	线程TID: 36   ID: 9   内存地址: FFFFDC887962A090   对象: FFFFDB281DE38A080

线程信息在进程信息的下面，枚举效果如下：

DebugView on \\DESKTOP-B53PAVI (local)

File Edit Capture Options Computer Help

#	Time	Debug Print
894	6.19741106	线程TID: 5152   ID: 8   内存地址: FFFFDC887EBF7080   对象: FFFFDB281E26A4080
895	6.20275927	线程TID: 5160   ID: 10   内存地址: FFFFDC887EBF70A0   对象: FFFFDB281E1F97080
896	6.20745420	线程TID: 5168   ID: 12   内存地址: FFFFDC887EBF70C0   对象: FFFFDB281E26A0080
897	6.21280813	线程TID: 5176   ID: 14   内存地址: FFFFDC887EBF70E0   对象: FFFFDB281E2991080
898	6.21819448	线程TID: 5180   ID: 15   内存地址: FFFFDC887EBF70F0   对象: FFFFDB281E269D080
899	6.22355032	线程TID: 5184   ID: 16   内存地址: FFFFDC887EBF7100   对象: FFFFDB281E269C080
900	6.22892714	线程TID: 5188   ID: 17   内存地址: FFFFDC887EBF7110   对象: FFFFDB281E269B080
901	6.23432016	线程TID: 5192   ID: 18   内存地址: FFFFDC887EBF7120   对象: FFFFDB281E269A0C0
902	6.23972321	进程PID: 5200   ID: 20   内存地址: FFFFDC887EBF7140   对象: FFFFDB281E2690080
903	6.24488497	线程TID: 5204   ID: 21   内存地址: FFFFDC887EBF7150   对象: FFFFDB281E1DC0080
904	6.25032473	线程TID: 5216   ID: 24   内存地址: FFFFDC887EBF7180   对象: FFFFDB281E2694040
905	6.25564957	线程TID: 5220   ID: 25   内存地址: FFFFDC887EBF7190   对象: FFFFDB281E2693040
906	6.26105118	线程TID: 5224   ID: 26   内存地址: FFFFDC887EBF71A0   对象: FFFFDB281E2692040
907	6.26644087	线程TID: 5228   ID: 27   内存地址: FFFFDC887EBF71B0   对象: FFFFDB281E2691040
908	6.27165461	线程TID: 5232   ID: 28   内存地址: FFFFDC887EBF71C0   对象: FFFFDB281E271E040
909	6.28022480	线程TID: 5236   ID: 29   内存地址: FFFFDC887EBF71D0   对象: FFFFDB281E271D040
910	6.28579903	线程TID: 5240   ID: 30   内存地址: FFFFDC887EBF71E0   对象: FFFFDB281E271C040
911	6.29111000	线程TID: 5244   ID: 31   内存地址: FFFFDC887EBF71F0   对象: FFFFDB281E271B040

至此文章就结束了，这里多说一句，实际上 ZwQuerySystemInformation 枚举系统句柄时就是走的这条双链，枚举系统进程如果使用的是这个API函数，那么不出意外它也是在这些内核表中做的解析。