在上一篇文章 《驱动开发：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。

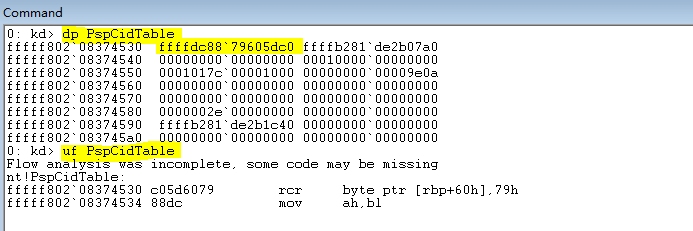
那么在 Windows10 系统中该如何枚举句柄表；

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。

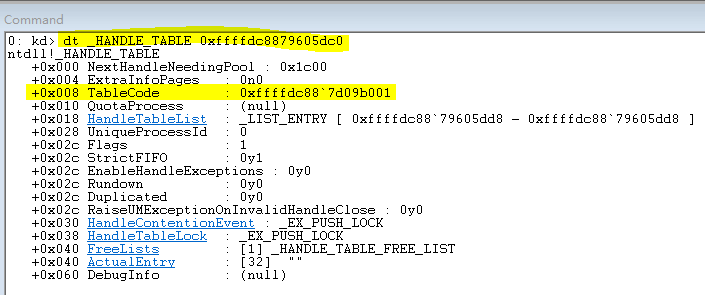
3.最后在 PspReferenceCidTableEntry 地址中找到 PspCidTable 函数。

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。



PspCidTable是一个 HANDLE\_TALBE 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 ffffdc88-79605dc0 这个内存地址，执行 dt \_HANDLE\_TABLE 0xffffdc8879605dc0 得到规范化结构体。



# 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\_TBALE\_ENTRY 项的多级索引。

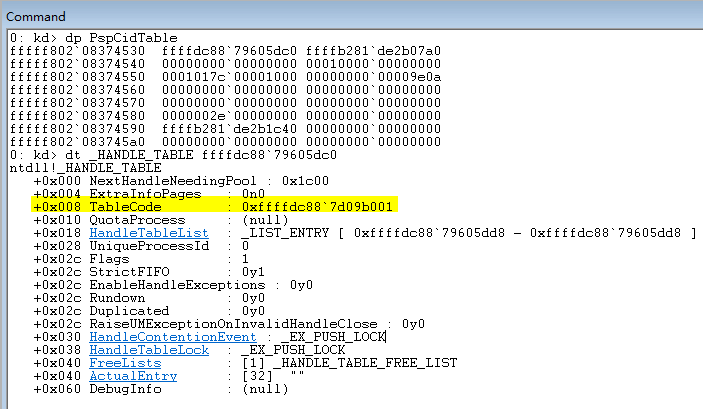
# 如上图所示 TableCode 是指向句柄表的指针，低二位（二进制）记录句柄表的等级：0（00）表示一级

0xffffdc88-7d09b001

表，1（01）表示二级表，2（10）表示三级表。这里的

表。

就说名它是一个二级



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 结构 （大

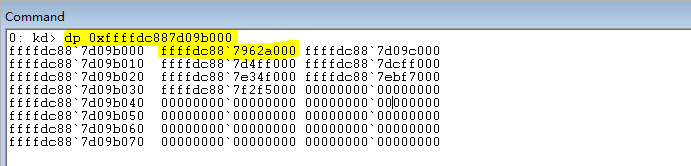
\_handle\_table\_entry

小 = 16） ，二级表和三级表存放的是指针 （大小 = 8） 。

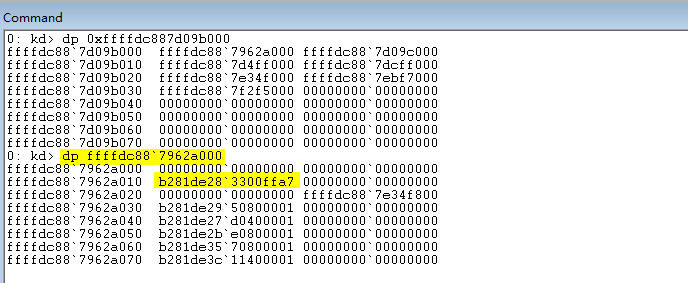
我们对 抹去低二位，输入 dp 0xffffdc887d09b000 输出的结果就是一张二级

0xffffdc88-7d09b001

# 表，里面存储的就是一级表指针。



继续查看第一张一级表，输入 dp 0xffffdc887962a000 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 。

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 = (LONG64)ul\_recode >> 0x10; ul\_decode &= 0xfffffffffffffff0;

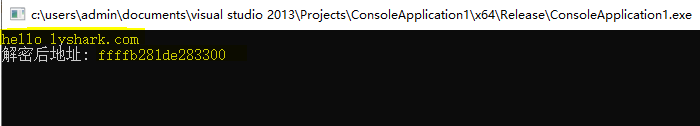
std::cout << "解密后地址: " << std::hex << ul\_decode << std::endl; getchar();

return 0;

}

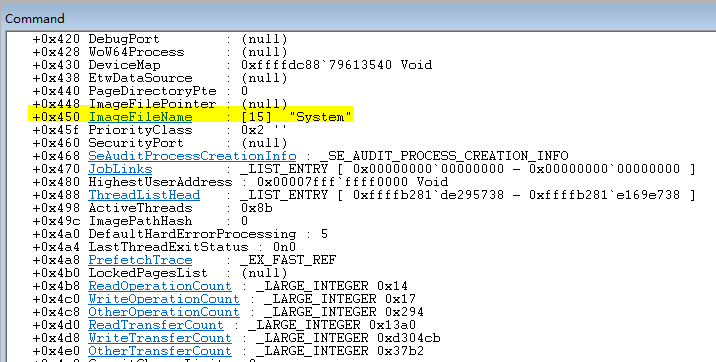
运行程序得到如下输出，即可知道 System 系统进程解密后的 EPROCESS 结构地址是

0xffffb281de283300



回到WinDBG调试器，输入命令 dt \_EPROCESS 0xffffb281de283300 解析以下这个结构，输出结果是

# System进程。



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。



// 署名

// PowerBy: LyShark

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

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

// 获取 PspCidTable

// By: LyShark.com

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`0841cfe0 | 48895c2418 | mov | qword ptr [rsp+18h],rbx |
| fffff802`0841cfe5 | 56 | push | rsi |
| fffff802`0841cfe6 | 4883ec20 | sub | rsp,20h |
| fffff802`0841cfea | 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) == 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;

}

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;

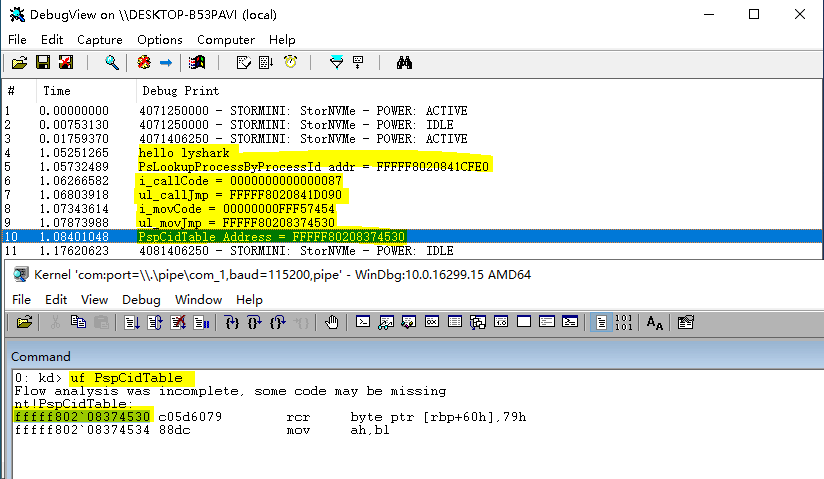
get\_PspCidTable(&tableAddr);

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：



继续增加对与三级表的动态解析代码，最终代码如下所示：



// 署名

// PowerBy: LyShark

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

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

// 获取 PspCidTable

// By: LyShark.com

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`0841cff2 65488b3c2588010000 mov rdi,qword ptr gs:[188h] fffff802`0841cffb 66ff8fe6010000 dec word ptr [rdi+1E6h]

fffff802`0841d002 b203 mov dl,3

|  |  |  |  |
| --- | --- | --- | --- |
| fffff802`0841cfe0 | 48895c2418 | mov | qword ptr [rsp+18h],rbx |
| fffff802`0841cfe5 | 56 | push | rsi |
| fffff802`0841cfe6 | 4883ec20 | sub | rsp,20h |
| fffff802`0841cfea | 48897c2438 | mov | qword ptr [rsp+38h],rdi |
| fffff802`0841cfef | 488bf2 | mov | rsi,rdx |

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) == 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

// By: LyShark.com 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

// By: LyShark.com

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

// By: LyShark.com

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

{

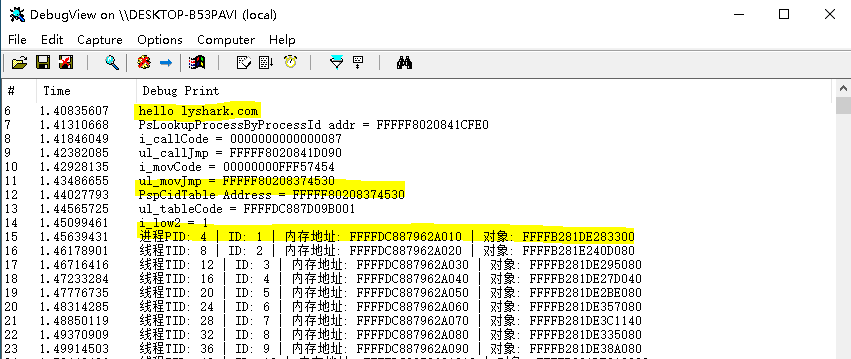
DbgPrint("LyShark提示: 错误,非法! "); return FALSE;

}

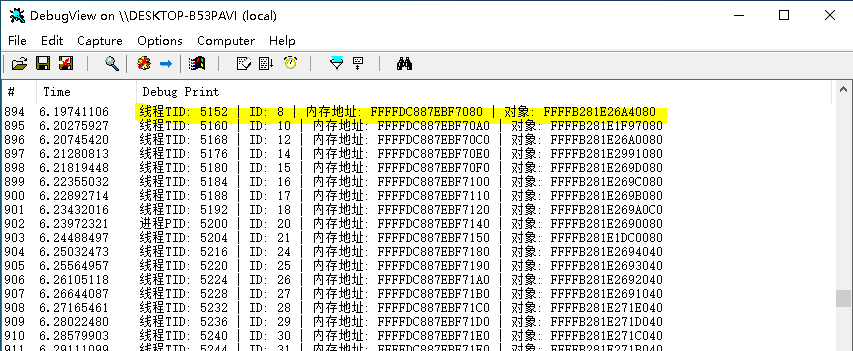
Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

# 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：



线程信息在进程信息的下面，枚举效果如下：



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。

**参考文献**

# h[ttp://www.blogfshare.com/details-in-pspcidtbale.htm](http://www.blogfshare.com/details-in-pspcidtbale.html)l h ttps://blog.csdn.net/whatday/article/details/17189093 h [ttps://www.cnblogs.com/kuangke/p/5761615.htm](http://www.cnblogs.com/kuangke/p/5761615.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！