1 句柄表

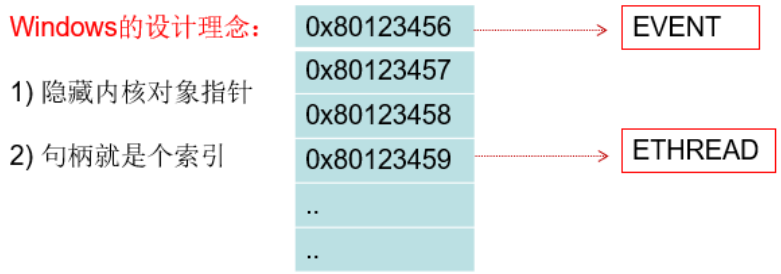
1.1 了解句柄表

1.1.1 基础知识

当一个进程创建或者打开一个内核对象，都会获得一个句柄，通过句柄我们可以访问到内核对象，**也就表示我们这里学习的句柄是对应一个内核对象的**，并不是我们之前所了解的窗口之类的3环句柄。

句柄的存在是为了避免在应用层直接修改内核对象，如果你创建一个线程，随之返回给你内核对象的地址，你就可以通过这个地址去修改内核对象，**一旦你修改内核对象地址指向了一个无效的内核内存地址，在0环处理时就会导致操作系统崩溃（蓝屏），而在三环指向一个无效的内存地址最多就程序程序退出**。

为了规避以上所述的情况，微软将内核对象的地址隐藏起来，但为了让3环可以使用，又设计了一张表（即句柄表），在这个表中存储着内核对象的地址，**句柄则为这张表中的索引**，这样在3环中就可以通过句柄来使用内核对象。



在每个进程的内存中都有一张自己的句柄表，表中记录着你创建、打开的内核对象地址，**需要注意的是，当你在当前进程中打开其他进程的线程、事件，操作系统并不会再次创建一份内核对象，而是返回这个内核对象的地址存到句柄表中，再返回句柄给你**。

1.1.2 句柄表

EPROCESS（进程结构体）中的0xC4偏移位成员ObjectTable（_HANDLE_TABLE）中的0x0偏移位成员TableCode就是句柄表。

```
0: kd> dt _EPROCESS
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   ...
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
0: kd> dt _HANDLE_TABLE
nt!_HANDLE_TABLE
   +0x000 TableCode        : Uint4B
```

我们可以在虚拟机中打开一个计算器，然后循环10次打开计算器的进程，这样我们就可以在句柄表中找到着10个被我们使用的内核对象信息，实验可以通过如下代码进行：

```
1  #include <windows.h>
2  #include <stdio.h>
```

```
3
4   int main(int argc, char* argv[])
5   {
6       DWORD pid;
7       HANDLE hProcess;
8
9       HWND hwnd = FindWindow(NULL, "计算器");
10      GetWindowThreadProcessId(hwnd, &pid);
11
12      for (int i = 0; i < 10; i++)
13      {
14          hProcess = OpenProcess(PROCESS_ALL_ACCESS, TRUE, pid);
15          printf("句柄：%x\n", hProcess);
16      }
17
18      getchar();
19      return 0;
20  }
```

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

int main(int argc, char* argv[])
{
    DWORD pid;
    HANDLE hProcess;

    HWND hwnd = FindWindow(NULL, "计算器");
    GetWindowThreadProcessId(hwnd, &pid);

    for (int i = 0; i < 10; i++)
    {
        hProcess = OpenProcess(PROCESS_ALL_ACCESS, TRUE, pid);
        printf("句柄：%x\n", hProcess);
    }

    getchar();
    return 0;
}
```

接着我们在Windbg中使用!process 0 0来找到Test.exe的进程结构体地址，如下图所示地址为890dd778：

```
0: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
Failed to get VadRoot
PROCESS 89a32830  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 0a3c0020  ObjectTable: e1000e08  HandleCount: 263.
    Image: System
 ...
Failed to get VadRoot
PROCESS 890dd778  SessionId: 0  Cid: 0e8c    Peb: 7ffd9000  ParentCid: 07e4
    DirBase: 0a3c03e0  ObjectTable: e10c7368  HandleCount:  27.
    Image: Test.exe
```

接着使用dt _EPROCESS 890dd778指令以结构体形式展示这块内存，找到成员ObjectTable：

```
0: kd> dt _EPROCESS 890dd778
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER 0x01d935dc`47a70a45
   +0x078 ExitTime         : _LARGE_INTEGER 0x0
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : 0x00000e8c Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY [ 0x805637b8 - 0x8962b998 ]
   +0x090 QuotaUsage       : [3] 0x5c8
   +0x09c QuotaPeak        : [3] 0x5c8
   +0x0a8 CommitCharge     : 0x73
   +0x0ac PeakVirtualSize  : 0xf02000
   +0x0b0 VirtualSize      : 0xf02000
   +0x0b4 SessionProcessLinks : _LIST_ENTRY [ 0xbadd8014 - 0x8962b9c4 ]
   +0x0bc DebugPort        : 0x89926590 Void
   +0x0c0 ExceptionPort    : 0xe15a0b18 Void
   +0x0c4 ObjectTable      : 0xe10c7368 _HANDLE_TABLE
```

再跟进这个成员就找到了句柄表的位置，即如下图所示的0xe26fa000：

```
0: kd> dt _HANDLE_TABLE 0xe10c7368
nt!_HANDLE_TABLE
   +0x000 TableCode         : 0xe26fa000
   +0x004 QuotaProcess      : 0x890dd778 _EPROCESS
   +0x008 UniqueProcessId   : 0x00000e8c Void
   +0x00c HandleTableLock   : [4] _EX_PUSH_LOCK
   +0x01c HandleTableList   : _LIST_ENTRY [ 0x80564aa8 - 0xe1c6f724 ]
   +0x024 HandleContentionEvent : _EX_PUSH_LOCK
   +0x028 DebugInfo         : (null)
   +0x02c ExtraInfoPages    : 0n0
   +0x030 FirstFree         : 0x7a8
   +0x034 LastFree          : 0
   +0x038 NextHandleNeedingPool : 0x800
   +0x03c HandleCount       : 0n27
   +0x040 Flags             : 0
   +0x040 StrictFIFO        : 0y0
```

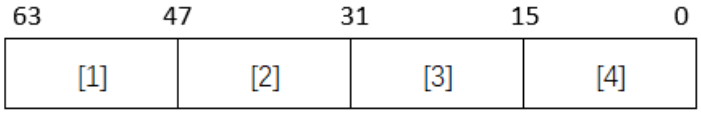
在查看整个句柄表之前我们要知道Windows考虑到兼容性等原因，设计句柄表的每个成员的宽度为8字节，而不是4字节，而我们在3环所看见的句柄的宽度为4字节，**因此我们想要通过句柄在句柄表中找到对应的内核对象地址，就需要使用这个公式：句柄表地址 + 句柄 / 4 * 8**。

因此，我们取一个句柄7ac，来找到它对应的内核对象，可以使用dq 0xe26fa000 + 0x7ac / 4 * 8指令来找到，如下图所示我们可以看见有10个一样的句柄表成员，这也就是我们所使用的10个内核对象：

```
dq 0xe26fa000 + 0x7ac / 4 * 8
   001f0fff`8962b8fb 001f0fff`8962b8fb
   001f0fff`8962b8fb 001f0fff`8962b8fb
   001f0fff`8962b8fb 001f0fff`8962b8fb
   001f0fff`8962b8fb 001f0fff`8962b8fb
   001f0fff`8962b8fb 001f0fff`8962b8fb
   000f01ff`8983fdc9 000f037f`899282b9
   021f0003`89706229 000f037f`899282b9
   001f0003`89853e51 000f001f`e1718ab9
```

表项结构

句柄表表项数据宽度为8字节，主要分为四个部分：



第一部分：共计两个字节（48-63位），低字节保留（一直都是0），高位字节是给SetHandleInformation这个函数用的，例如当执行如下语句：

```
SetHandleInformation(Handle, HANDLE_FLAG_PROTECT_FROM_CLOSE,
HANDLE_FLAG_PROTECT_FROM_CLOSE);
```

那么高位字节就会被写入0x02，这是因为HANDLE_FLAG_PROTECT_FROM_CLOSE宏的值为0x00000002，取了其最低字节写入第一部分的高位字节中，因此第一部分最终的值就是0x0200；

第二部分：共计2个字节（32-47），表示访问掩码，是给OpenProcess函数使用的，其存储的值就是OpenProcess函数的第一个参数对应的值；

第三部分和第四部分：共计4字节（0-31），第0位表示调用者是否允许关闭该句柄默认值为1，第1位表示该句柄是否可继承（是否可以将句柄项拷贝到其他句柄表中），第2位表示关闭该对象时是否产生一个审计事件默认值为0，**第3位到第31位存放的是该内核对象在内核中的具体地址**。

内核对象

_OBJECT_HEADER

通过上文的学习，我们了解到句柄表项的低32位可以获取到内核对象的地址，但是这个地址指向的并不是结构体的开头，**内核对象在开头都会有一个0x18字节的_OBJECT_HEADER结构**，这是内核对象的头部，也就是说从0x18字节开始才是进程结构体第一个成员的位置。

```
0: kd> dt _OBJECT_HEADER
nt!_OBJECT_HEADER
   +0x000 PointerCount     : Int4B
   +0x004 HandleCount      : Int4B
   +0x004 NextToFree       : Ptr32 Void
   +0x008 Type             : Ptr32 _OBJECT_TYPE
   +0x00c NameInfoOffset   : UChar
   +0x00d HandleInfoOffset : UChar
   +0x00e QuotaInfoOffset  : UChar
   +0x00f Flags            : UChar
   +0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : Ptr32 Void
   +0x014 SecurityDescriptor : Ptr32 Void
   +0x018 Body             : _QUAD
```

作用

我们可以通过遍历句柄表的方式来查看是否有程序加载自己，以此来达到反调试的目的；除此之外我们还可以通过句柄表来遍历进程列表。

1.2 全局句柄表

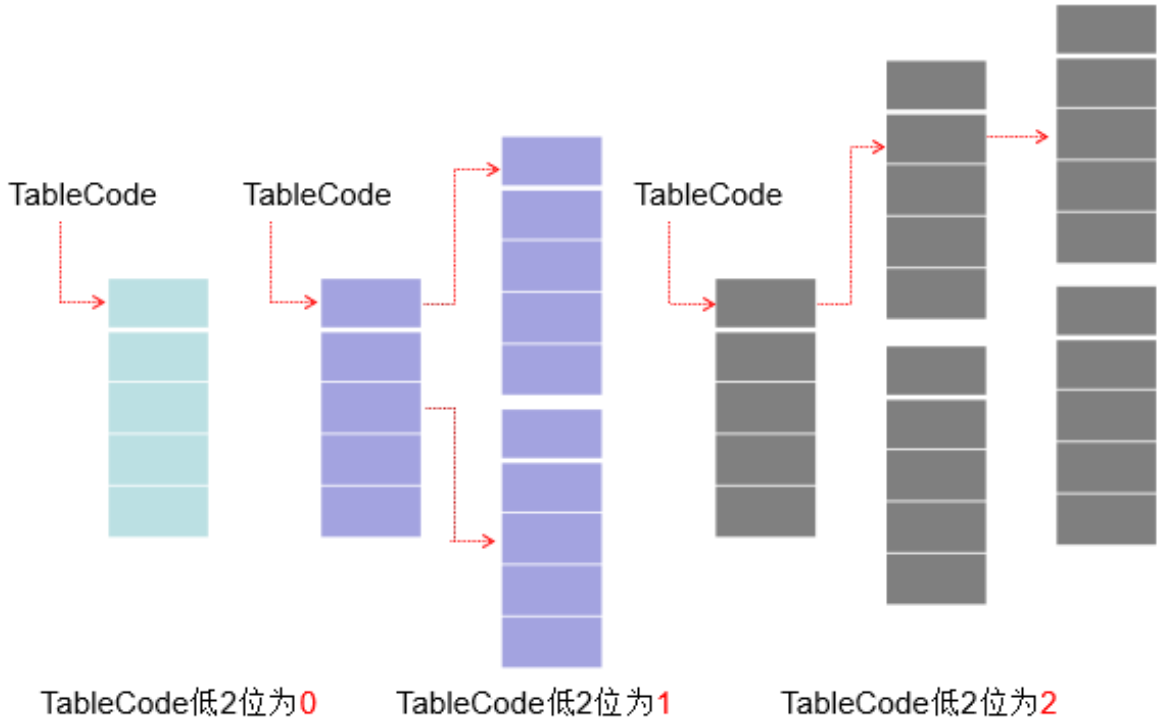
之前我们所了解的句柄表都是每个进程私有的句柄表，而实际上在Windows操作系统上有一个全局句柄表，所有进程、线程无论是否打开，都存放在这张表中。每个进程和线程都有一个唯一的编号，即PID、CID（我们在任务管理器中可以看到进程的PID），这两个值就是全局句柄表中的索引。

我们可以通过如下函数根据PID、CID来获得进程、线程的内核对象，它们所查询的内核对象就是来自全局句柄表（PsdCidTable）：

```
PsLookupProcessThreadByCid
PsLookupProcessByProcessId
```

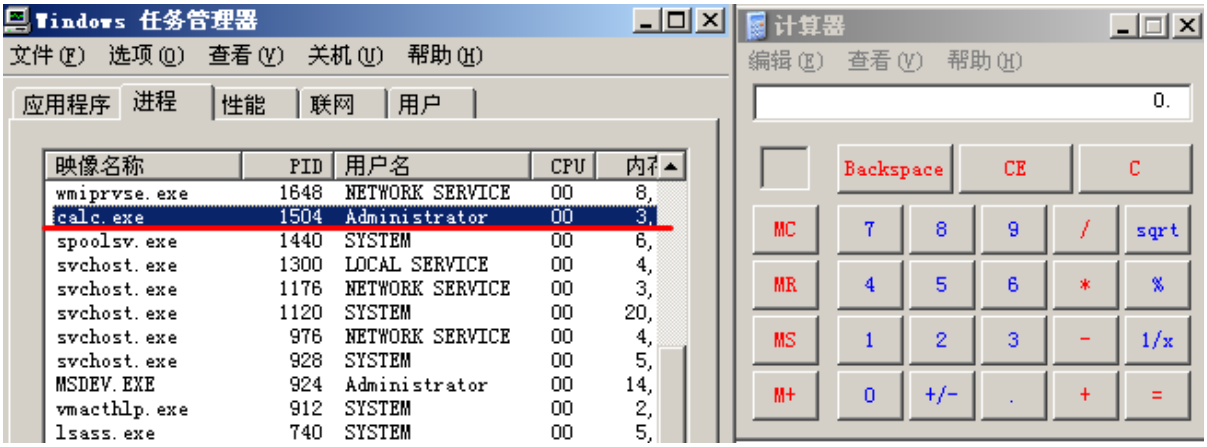
```
3    PsLookupThreadByThreadId
```

全局句柄表相对于进程私有的句柄表来说比较复杂，它的结构如下图所示，**全局句柄表的大小可以通过它的成员TableCode来查看**，当TableCode的低2位值为0时，则表示全局句柄表的大小只有一页也就是4KB（4096个字节），512个表项，当TableCode的低2位值为1时，它就是多级的，第一页存储的就是1024个地址，每个地址指向着具有实际内容（内核对象地址）的表，那也就表示有1024*512个表项，TableCode的其他值以此类推：



1.2.1 查找全局句柄表

我们可以通过Windbg实现查找全局句柄表中的进程内核对象，首先在Windows上打开一个计算器，在任务管理器中找到它的PID：



我们接着在Windbg中找到全局句柄表，通过第一个指令找到全局句柄表的地址，然后使用结构体形式展示全局句柄表，找到它的TableCode，低2位为1，则表示这是多级的全局句柄表：

```
0: kd> dd PspCidTable
805638c0  e1002c98 00000002 00000000 00000000
805638d0  00000000 00000000 00000000 00000000
805638e0  00000000 00000000 00000000 00000000
805638f0  00000000 00000000 00000000 00000000
80563900  00000000 00000000 00000000 00000000
80563910  00000000 00000000 00000000 00000000
80563920  00000000 00000000 00000000 00000000
80563930  00000000 00000000 00000000 00000000
0: kd> dt _HANDLE_TABLE e1002c98
nt!_HANDLE_TABLE
   +0x000 TableCode        : 0xe1044001
   +0x004 QuotaProcess     : (null)
```

在多级句柄表的情况下我们就要知道当前的索引是否超出512，使用PID/4得出376，则表示它是在第一张表内，我们清空TableCode的低2位跟进找到第一张表的地址，然后根据索引计算找到进程内核对象的地址：

```
0: kd> dd 0xe1044000
e1044000  e1005000 e1045000 00000000 00000000
e1044010  00000000 00000000 00000000 00000000
e1044020  00000000 00000000 00000000 00000000
e1044030  00000000 00000000 00000000 00000000
e1044040  00000000 00000000 00000000 00000000
e1044050  00000000 00000000 00000000 00000000
e1044060  00000000 00000000 00000000 00000000
e1044070  00000000 00000000 00000000 00000000
0: kd> dq e1005000 + 0x5e0 / 4 * 8
e1005bc0  00000000`8962b911 00000e90`00000000
e1005bd0  00000000`897aa1d9 00000000`8984a801
e1005be0  00000000`898f6021 00000370`00000000
e1005bf0  000000cc`00000000 00000000`89694659
e1005c00  000005ac`00000000 00000000`8980d601
e1005c10  00000000`894954c1 00000c18`00000000
e1005c20  00000428`00000000 00000848`00000000
e1005c30  00000f18`00000000 000006c4`00000000
```

值得注意的是全局句柄表与私有句柄表不同，前者指向的内核对象地址不包含0x18字节的_OBJECT_HEADER结构，因此我们只需要将低3位的属性清0即可直接访问到进程结构体，如下图所示，我们通过ImageFileName确定我们找到的是计算机进程对应的结构体：

```
0: kd> dt _EPROCESS 8962b910
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER 0x01d935dc`3e3fd776
   ...
   +0x174 ImageFileName    : [16]  "calc.exe"
```