对象回调是目前绝大多数游戏保护用于保护游戏进程用的回调。比如著名的 CF，在 WIN64 系统上，TP 只有对象回调保护 CF 进程不被外挂修改其进程内容（这段话是 2013 年 12 月中旬研究 TP 得出的结论，不保证以后会不会变化），如果使用 WIN64AST 摘除 TP 的两个对象回调，TP 对 CF 进程的保护作用就会消失。

对象回调存储在对应对象结构体里，简单来说，就是存储在 ObjectType. CallbackList 这个双向链表里。但对象结构体在每个系统上都不一定相同。比如 WIN7X64 的结构体如下：

```
ntdll!_OBJECT_TYPE
    +0x000 TypeList             : _LIST_ENTRY
    +0x010 Name                 : _UNICODE_STRING
    +0x020 DefaultObject        : Ptr64 Void
    +0x028 Index                : UChar
    +0x02c TotalNumberOfObjects : Uint4B
    +0x030 TotalNumberOfHandles : Uint4B
    +0x034 HighWaterNumberOfObjects : Uint4B
    +0x038 HighWaterNumberOfHandles : Uint4B
    +0x040 TypeInfo             : _OBJECT_TYPE_INITIALIZER
    +0x0b0 TypeLock             : _EX_PUSH_LOCK
    +0x0b8 Key                  : Uint4B
    +0x0c0 CallbackList         : _LIST_ENTRY
```

按理来说，知道了回调存储的地址，枚举应该就很简单了。但是只有一个链表能知道什么？对象回调至少有三个关键信息：PreCall 函数地址，PostCall 函数地址，回调句柄。这些信息藏在哪里呢？当年我对此感到百思不得其解。后来经过研究，发现秘密就藏在 CallbackList 的第二项以及之后。换句话说，在 ListHead->Flink 以及之后大有乾坤。Object.CallbackList->FLink 指向的地址，是一个结构体链表，它的定义如下：

```
typedef struct    _OB_CALLBACK
{
    LIST_ENTRY    ListEntry;
    ULONG64       Unknown;
    ULONG64       ObHandle;
    ULONG64       ObjTypeAddr;
    ULONG64       PreCall;
    ULONG64       PostCall;
} OB_CALLBACK, *POB_CALLBACK;
```

微软没有公开这个结构体的定义，这个结构体是我逆向出来的。但是至少在 WIN7、WIN8 和 WIN8.1 上通用。知道了结构体的定义，枚举就方便了（WINDOWS 目前仅有进程对象回调和线程对象回调，但就算以后有了其它回调，也是通用的）：

```
ULONG EnumObCallbacks()
{
    ULONG c=0;
    PLIST_ENTRY CurrEntry=NULL;
    POB_CALLBACK pObCallback;
```

```
	BOOLEAN IsTxCallback;
	ULONG64 ObProcessCallbackListHead = *(ULONG64*)PsProcessType + ObjectCallbackListOffset;
	ULONG64 ObThreadCallbackListHead = *(ULONG64*)PsThreadType + ObjectCallbackListOffset;
	//
	dprintf("ObProcessCallbackListHead: %p\n",ObProcessCallbackListHead);
	CurrEntry=((PLIST_ENTRY)ObProcessCallbackListHead)->Flink;  //list_head 的数据是垃圾数据，忽略
	do
	{
		pObCallback=(POB_CALLBACK)CurrEntry;
		if(pObCallback->ObHandle!=0)
		{
			dprintf("ObHandle: %p\n",pObCallback->ObHandle);
			dprintf("PreCall: %p\n",pObCallback->PreCall);
			dprintf("PostCall: %p\n",pObCallback->PostCall);
			c++;
		}
		CurrEntry = CurrEntry->Flink;
	}
	while(CurrEntry != (PLIST_ENTRY)ObProcessCallbackListHead);
	//
	dprintf("ObThreadCallbackListHead: %p\n",ObThreadCallbackListHead);
	CurrEntry=((PLIST_ENTRY)ObThreadCallbackListHead)->Flink;  //list_head 的数据是垃圾数据，忽略
	do
	{
		pObCallback=(POB_CALLBACK)CurrEntry;
		if(pObCallback->ObHandle!=0)
		{
			dprintf("ObHandle: %p\n",pObCallback->ObHandle);
			dprintf("PreCall: %p\n",pObCallback->PreCall);
			dprintf("PostCall: %p\n",pObCallback->PostCall);
			c++;
		}
		CurrEntry = CurrEntry->Flink;
	}
	while(CurrEntry != (PLIST_ENTRY)ObThreadCallbackListHead);
	dprintf("ObCallback count: %ld\n",c);
	return c;
}
```

代码运行的效果如下（干净的 WIN7X64 系统上是没有对象回调的，为了体现枚举效果，

可以先运行 WIN64AST）：

```
16.79011917   NtBuildNumber: 7601
16.79012108   ObProcessCallbackListHead: FFFFFA8018D41B20
16.79012299   ObHandle: FFFFF8A0019FA510
16.79012489   PreCall: FFFFF88005E2D028
16.79012680   PostCall: 0000000000000000
16.79012871   ObThreadCallbackListHead: FFFFFA8018D419D0
16.79012871   ObHandle: FFFFF8A00115CC60
16.79013062   PreCall: FFFFF88005E2D1D8
16.79013252   PostCall: 0000000000000000
16.79013443   ObCallback count: 2
```

对付对象回调，方法还是老三套：1.用 ObUnRegisterCallbacks 传入 ObHandle 注销回调；2.把记录的回调函数地址改为自己的设置的空回调；3.给对方设置的回调函数地址写入 RET。不过这次使用第三种方法要注意，必须先禁掉 PostCall，再禁用 PreCall，否则容易蓝屏，不信自己试试。