在内核里操作进程，相信是很多对 WINDOWS 内核编程感兴趣的朋友第一个学习的知识点。但在这里，我要让大家失望了，在内核里操作进程没什么特别的，就标准方法而言，还是调用那几个和进程相关的 NATIVE API 而已（当然了，本文所说的进程操作，还包括对线程和 DLL 模块的操作）。本文包括 10 个部分：分别是：枚举进程、暂停进程、恢复进程、结束进程、枚举线程、暂停线程、恢复线程、结束线程、枚举 DLL 模块、卸载 DLL 模块。

1.枚举进程。进程就是活动起来的程序。每一个进程在内核里，都有一个名为 EPROCESS 的巨大结构体记录它的详细信息，包括它的名字，编号（PID），出生地点（进程路径），老爹是谁（PPID 或父进程 ID）等。在 RING3 枚举进程，通常只要列出所有进程的编号即可。不过在 RING0 里，我们还要把它的身份证（EPROCESS）地址给列举出来。顺带说一句，**现实中男人最怕的事情就是“喜当爹”，这种事情在内核里更加容易发生。因为 EPROCESS 里有且只有一个成员是记录父进程 ID 的，稍微改一下，就可以认任意进程为爹了**。枚举进程的方法很多，标准方法是使用 ZwQuerySystemInformation 的 SystemProcessInformation 功能号，不过如果在内核里也这么用的话，那就真是脱了裤子放屁——多此一举。因为在内核里使用这个函数照样是得不到进程的 EPROCESS 地址，而且一旦内存出错，还会蓝屏，更加逃不过任何隐藏进程的手法。**所以在内核里稳定又不失强度的枚举进程方法是枚举 PspCidTable，它能最大的好处是能得到进程的 EPROCESS 地址，而且能检查出使用“断链”这种低级手法的隐藏进程**。不过话也说回来，枚举 PspCidTable 并不是一件很爽的事情，因为 **PspCidTable 是一个不公开的变量，要获得它地址的话，必然要使用硬编码或者符号。所以我的方法是：变相枚举 PspCidTable**。内核里有个函数叫做 PsLookupProcessByProcessId，它能通过进程 PID 查到进程的 EPROCESS，它的内部实现正是枚举了 PspCidTable。PID 的范围是从 4 开始，到 MAX_INT（2^31-1）结束，步进为 4。但实际上，大家见到的 PID 基本都是小于 10000 的，而上 10000 的 PID 相信很多人都没有见过。所以我们实际的枚举范围是 4～2^18，如果 PsLookupProcessByProcessId 返回失败，则证明此进程不存在，如果返回成功，则把 EPROCESS、PID、PPID、进程名打印出来。

```
//声明 API
NTKERNELAPI UCHAR* PsGetProcessImageFileName( IN PEPROCESS Process );
NTKERNELAPI HANDLE PsGetProcessInheritedFromUniqueProcessId( IN PEPROCESS Process );

//根据进程 ID 返回进程 EPROCESS，失败返回 NULL
PEPROCESS LookupProcess(HANDLE Pid)
{
	PEPROCESS eprocess=NULL;
	if( NT_SUCCESS(PsLookupProcessByProcessId(Pid, &eprocess)) )
		return eprocess;
	else
		return NULL;
}
//枚举进程
VOID EnumProcess()
{
	ULONG i=0;
	PEPROCESS eproc=NULL;
	for(i=4; i<262144; i=i+4)
```

```
    {
        eproc=LookupProcess((HANDLE)i);
        if(eproc!=NULL)
        {
            DbgPrint("EPROCESS=%p, PID=%ld, PPID=%ld, Name=%s",
                    eproc,
                    (DWORD)PsGetProcessId(eproc),
                    (DWORD)PsGetProcessInheritedFromUniqueProcessId(eproc),
                    PsGetProcessImageFileName(eproc));
            ObDereferenceObject(eproc);
        }
    }
}
```

2.暂停进程。暂停进程就是暂停进程的活动，但是不将其杀死。暂停进程在 VISTA 之后有导出的函数：PsSuspendProcess。它的函数原型很简单：

```
NTKERNELAPI //声明要使用此函数
NTSTATUS //返回类型
PsSuspendProcess(PEPROCESS Process); //唯一的参数是 EPROCESS
```

3.恢复进程。恢复进程就是让被暂停进程的恢复活动，是上一个操作的反操作。恢复进程在 VISTA 之后有导出的函数：PsResumeProcess。它的函数原型很简单：

```
NTKERNELAPI //声明要使用此函数
NTSTATUS //返回类型
PsResumeProcess(PEPROCESS Process); //唯一的参数是 EPROCESS
```

4.结束进程。结束进程的标准方法就是使用 ZwOpenProcess 打开进程获得句柄，然后使用 ZwTerminateProcess 结束，最后使用 ZwClose 关闭句柄。除了这种方法之外，还能用使用内存清零的方式结束进程，后者使用有一定的危险性，可能在特殊情况下发生蓝屏，但强度比前者大得多。在 WIN64 不可以搞内核 HOOK 的大前提下，后者可以结束任何被保护的进程。

```
//正规方法结束进程
void ZwKillProcess()
{
    HANDLE hProcess = NULL;
    CLIENT_ID ClientId;
    OBJECT_ATTRIBUTES oa;
    //填充 CID
    ClientId.UniqueProcess = (HANDLE)2732; //这里修改为你要的 PID
    ClientId.UniqueThread = 0;
    //填充 OA
    oa.Length = sizeof(oa);
    oa.RootDirectory = 0;
    oa.ObjectName = 0;
    oa.Attributes = 0;
```

```
	oa.SecurityDescriptor = 0;
	oa.SecurityQualityOfService = 0;
	//打开进程，如果句柄有效，则结束进程
	ZwOpenProcess(&hProcess,1,&oa,&ClientId);
	if(hProcess)
	{
		ZwTerminateProcess(hProcess,0);
		ZwClose(hProcess);
	};
}
```

```
NTKERNELAPI VOID NTAPI KeAttachProcess(PEPROCESS Process);
NTKERNELAPI VOID NTAPI KeDetachProcess();
//内存清零法结束进程
void PVASE()
{
	SIZE_T i=0;
	//依附进程
	KeAttachProcess((PEPROCESS)0xFFFFFA8003ABDB30); //这里改为指定进程的 EPROCESS
	for(i=0x10000;i<0x20000000;i+=PAGE_SIZE)
	{
		__try
		{
			memset((PVOID)i,0,PAGE_SIZE); //把进程内存全部置零
		}
		_except(1)
		{
			;
		}
	}
	//退出依附进程
	KeDetachProcess();
}
```

5.枚举线程。线程跟进程类似，也有一个身份证一样的结构体 ETHREAD 存放在内核里，而它所有的 ETHREAD 也是放在 PspCidTable 里的。于是有了类似枚举进程的代码：

```
//根据线程 ID 返回线程 ETHREAD，失败返回 NULL
PETHREAD LookupThread(HANDLE Tid)
{
	PETHREAD ethread;
	if( NT_SUCCESS(PsLookupThreadByThreadId(Tid, &ethread)) )
		return ethread;
	else
		return NULL;
}
```

```
//枚举指定进程的线程
VOID EnumThread(PEPROCESS Process)
{
	ULONG i=0,c=0;
	PETHREAD ethrd=NULL;
	PEPROCESS eproc=NULL;
	for(i=4; i<262144; i=i+4)
	{
		ethrd=LookupThread((HANDLE)i);
		if(ethrd!=NULL)
		{
			//获得线程所属进程
			eproc=IoThreadToProcess(ethrd);
			if(eproc==Process)
			{
				//打印出 ETHREAD 和 TID
				DbgPrint("ETHREAD=%p, TID=%ld\n",
					ethrd,
					(ULONG)PsGetThreadId(ethrd));
			}
			ObDereferenceObject(ethrd);
		}
	}
}
```

6.挂起线程。类似于“挂起进程”，唯一的差别是没有导出函数可用了。可以自行定位 PsSuspendThread，它的原型如下：

```
NTSTATUS PsSuspendThread
(IN PETHREAD Thread, //线程 ETHREAD
OUT PULONG PreviousSuspendCount OPTIONAL) //挂起的次数，每挂起一次此值增 1
```

7.恢复线程。类似于“恢复进程”， 唯一的差别是没有导出函数可用了。可以自行定位 PsResumeThread，它的原型如下：

```
NTSTATUS PsResumeThread
(PETHREAD Thread, //线程 ETHREAD
OUT PULONG PreviousCount); //恢复的次数，每恢复一次此值减 1，为 0 时线程才正常
```

8.结束线程。结束线程的标准方法是 ZwOpenThread+ZwTerminateThread+ZwClose，暴力方法是直接调用 PspTerminateThreadByPointer。暴力方法在后面的课程里讲，这里先讲标准方法。由于 ZwTerminateThread 没有导出，所以只能先硬编码了（在 WINDBG 里使用 x 命令获得地址：x nt!ZwTerminateThread）：

```
typedef NTSTATUS (__fastcall *ZWTERMINATETHREAD)(HANDLE hThread, ULONG uExitCode);
ZWTERMINATETHREAD ZwTerminateThread=0Xfffff80012345678; //要修改这个值
//正规方法结束线程
```

```
void ZwKillThread()
{
	HANDLE hThread = NULL;
	CLIENT_ID ClientId;
	OBJECT_ATTRIBUTES oa;
	//填充 CID
	ClientId.UniqueProcess =0;
	ClientId.UniqueThread = (HANDLE)1234; //这里修改为你要的 TID
	//填充 OA
	oa.Length = sizeof(oa);
	oa.RootDirectory = 0;
	oa.ObjectName = 0;
	oa.Attributes = 0;
	oa.SecurityDescriptor = 0;
	oa.SecurityQualityOfService = 0;
	//打开进程，如果句柄有效，则结束进程
	ZwOpenProcess(&hThread,1,&oa,&ClientId);
	if(hThread)
	{
		ZwTerminateThread(hThread,0);
		ZwClose(hThread);
	};
}
```

9.枚举 DLL 模块。DLL 模块记录在 PEB 的 LDR 链表里，LDR 是一个双向链表，枚举它即可。另外，DLL 模块列表包含 EXE 的相关信息。换句话说，**枚举 DLL 模块即可实现枚举进程路径**。

```
//声明偏移
ULONG64 LdrInPebOffset=0x018;		//peb.ldr
ULONG64 ModListInPebOffset=0x010;	//peb.ldr.InLoadOrderModuleList

//声明 API
NTKERNELAPI PPEB PsGetProcessPeb(PEPROCESS Process);

//声明结构体
typedef struct _LDR_DATA_TABLE_ENTRY
{
	LIST_ENTRY64 InLoadOrderLinks;
	LIST_ENTRY64 InMemoryOrderLinks;
	LIST_ENTRY64 InInitializationOrderLinks;
	PVOID		DllBase;
	PVOID		EntryPoint;
	ULONG		SizeOfImage;
	UNICODE_STRING	FullDllName;
	UNICODE_STRING	BaseDllName;
```

```
	ULONG		Flags;
	USHORT		LoadCount;
	USHORT		TlsIndex;
	PVOID		SectionPointer;
	ULONG		CheckSum;
	PVOID		LoadedImports;
	PVOID		EntryPointActivationContext;
	PVOID		PatchInformation;
	LIST_ENTRY64 ForwarderLinks;
	LIST_ENTRY64 ServiceTagLinks;
	LIST_ENTRY64 StaticLinks;
	PVOID		ContextInformation;
	ULONG64		OriginalBase;
	LARGE_INTEGER	LoadTime;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;

//根据进程枚举模块
VOID EnumModule(PEPROCESS Process)
{
	ULONG64 Peb = 0;
	ULONG64 Ldr = 0;
	PLIST_ENTRY ModListHead = 0;
	PLIST_ENTRY Module = 0;
	ANSI_STRING AnsiString;
	KAPC_STATE ks;
	//EPROCESS 地址无效则退出
	if( !MmIsAddressValid(Process) )
		return;
	//获取 PEB 地址
	Peb = PsGetProcessPeb(Process);
	//PEB 地址无效则退出
	if (!Peb)
		return;
	//依附进程
	KeStackAttachProcess(Process, &ks);
	__try
	{
		//获得 LDR 地址
		Ldr = Peb + (ULONG64)LdrInPebOffset;
		//测试是否可读，不可读则抛出异常退出
		ProbeForRead((CONST PVOID)Ldr, 8, 8);
		//获得链表头
		ModListHead = (PLIST_ENTRY)(*(PULONG64)Ldr + ModListInPebOffset);
		//再次测试可读性
```

```
            ProbeForRead((CONST PVOID)ModListHead, 8, 8);
            //获得第一个模块的信息
            Module = ModListHead->Flink;
            while (ModListHead != Module)
            {
                //打印信息：基址、大小、DLL 路径
                DbgPrint("Base=%p, Size=%ld, Path=%wZ",
                        (PVOID)(((PLDR_DATA_TABLE_ENTRY)Module)->DllBase),
                        (ULONG)(((PLDR_DATA_TABLE_ENTRY)Module)->SizeOfImage),
                        &(((PLDR_DATA_TABLE_ENTRY)Module)->FullDllName));
                Module = Module->Flink;
                //测试下一个模块信息的可读性
                ProbeForRead((CONST PVOID)Module, 80, 8);
            }
        }
        __except (EXCEPTION_EXECUTE_HANDLER)
        {
            DbgPrint("[EnumModule]__except (EXCEPTION_EXECUTE_HANDLER)");
        }
        //取消依附进程
        KeUnstackDetachProcess(&ks);
}
```

10.卸载 DLL 模块。使用 MmUnmapViewOfSection 即可。MmUnmapViewOfSection 的原型如下。填写正确的 EPROCESS 和 DLL 模块基址就能把 DLL 卸载掉。如果卸载 NTDLL 等重要 DLL 将会导致进程崩溃。

```
NTSTATUS MmUnmapViewOfSection
(IN PEPROCESS Process, //进程的 EPROCESS
IN PVOID BaseAddress) //DLL 模块基址
```

方法总结：

枚举进程	循环调用 PsLookupProcessByProcessId
暂停进程	PsSuspendProcess
恢复进程	PsResumeProcess
结束进程	ZwTerminateProcess
枚举线程	循环调用 PsLookupThreadByThreadId
暂停线程	PsSuspendThread
恢复线程	PsResumeThread
结束线程	ZwTerminateThread
枚举 DLL 模块	枚举 PEB.LDR 的双向链表
卸载 DLL 模块	MmUnmapViewOfSection

课后作业：大家试一下把这些源码组装起来，弄成一个简易进程管理器。