内核进程线程和模块是操作系统内核中非常重要的概念。它们是操作系统的核心部分，用于管理系统资源和处理系统请求。在驱动安全开发中，理解内核进程线程和模块的概念对于编写安全的内核驱动程序至关重要。

内核进程是在操作系统内核中运行的程序。每个进程都有一个唯一的进程标识符（PID），它用于在系统中唯一地标识该进程。在内核中，进程被表示为一个进程控制块（PCB），它包含有关进程的信息，如进程状态、优先级、内存使用情况等。枚举进程可以让我们获取当前系统中所有正在运行的进程的PID和其他有用的信息，以便我们可以监视和管理系统中的进程。

线程是在进程内部执行的轻量级执行单元。与进程不同，线程不拥有自己的地址空间和系统资源，它们共享它们所属进程的资源。在内核中，线程被表示为线程控制块（TCB），它包含有关线程的信息，如线程状态、调度信息、执行上下文等。枚举线程可以让我们获取当前系统中所有正在运行的线程的PID、线程ID和其他有用的信息，以便我们可以监视和管理系统中的线程。

内核模块是一种可加载的内核组件，它可以动态地添加到内核中。内核模块通常用于向内核添加新的设备驱动程序或系统功能。在驱动安全开发中，理解内核模块的概念对于编写安全的内核驱动程序非常重要。枚举内核模块可以让我们获取当前系统中加载的所有内核模块的名称、版本号和其他有用的信息，以便我们可以分析和调试内核模块。

在总体上，内核进程线程和模块是操作系统内核中非常重要的概念。通过了解这些概念，我们可以更好地理解操作系统内部的工作原理，从而编写更安全的内核驱动程序。

内核中实现枚举进程

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` 的结构记录它的详细信息，其中就包括进程名，PID，PPID，进程路径等，通常在应用层枚举进程只列出所有进程的编号即可，不过在内核层需要把它的 EPROCESS 地址给列举出来。

在内核中枚举进程我们可通过循环语句遍历进程句柄 `0-100000` 以内的值，每次通过 `PsLookupProcessByProcessId` 打开一个进程并得到进程 `EPROCESS` 结构，当获取到该结构体时只需要调用不同的三个内核函数即可获取到当前句柄所对应的进程相关信息。

当我们需要通过 `EPROCESS` 得到 `进程名` 时可使用 `PsGetProcessImageFileName()` 这个内核函数，该函数的具体定义规范如下所示；

```
PCHAR PsGetProcessImageFileName(
  PEPROCESS Process
);
```

其中，参数 `Process` 是一个 `PEPROCESS` 类型的指针，表示要获取映像文件名的进程的 `EPROCESS` 结构体指针；返回值是一个 `PCHAR` 类型的指针，指向包含指定进程映像文件名的空字符结尾字符串。

与之功能类似，当我们需要通过 `EPROCESS` 获取进程 `PID` 时，则可以调用 `PsGetProcessId()` 来获取到，该函数的具体定义规范如下所示；

```
HANDLE PsGetProcessId(
  PEPROCESS Process
);
```

其中，参数 `Process` 是一个 `PEPROCESS` 类型的指针，表示要获取进程 `ID` 的进程的 `EPROCESS` 结构体指针；返回值是一个 `HANDLE` 类型的进程 `ID` 值。

而如果当我们想要获取到进程的父进程时，则同样可使用 `PsGetProcessInheritedFromUniqueProcessId()` 来获取，该函数的具体定义规范如下所示；

```
HANDLE PsGetProcessInheritedFromUniqueProcessId(
  PEPROCESS Process
);
```

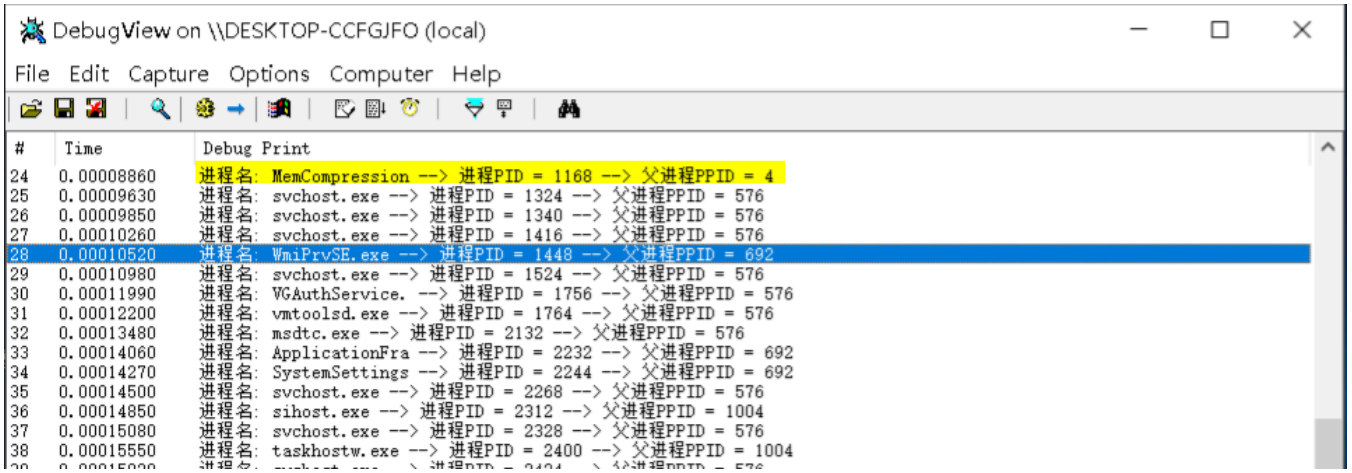
其中，参数`Process`是一个`PEPROCESS`类型的指针，表示要获取父进程ID的进程的`EPROCESS`结构体指针；返回值是一个`HANDLE`类型的父进程ID值。

有了这三个函数的支持，我们就可以实现遍历当前所有运行的进程啦，具体实现代码如下所示；

```
#include <ntifs.h>

// 未公开的进行导出即可
NTKERNELAPI UCHAR* PsGetProcessImageFileName(IN PEPROCESS Process);

// 未公开进行导出
NTKERNELAPI HANDLE PsGetProcessInheritedFromUniqueProcessId(IN PEPROCESS Process);

// 根据进程ID返回进程EPROCESS结构体,失败返回NULL
PEPROCESS LookupProcess(HANDLE Pid)
{
    PEPROCESS eprocess = NULL;
    NTSTATUS Status = STATUS_UNSUCCESSFUL;
    Status = PsLookupProcessByProcessId(Pid, &eprocess);
    if (NT_SUCCESS(Status))
        return eprocess;
    return NULL;
}

VOID EnumProcess()
{
    PEPROCESS eproc = NULL;
    for (int temp = 0; temp < 100000; temp += 4)
    {
        eproc = LookupProcess((HANDLE)temp);
        if (eproc != NULL)
        {
            DbgPrint("进程名: %s --> 进程PID = %d --> 父进程PPID =
%d\r\n",PsGetProcessImageFileName(eproc),PsGetProcessId(eproc),
                PsGetProcessInheritedFromUniqueProcessId(eproc));
            ObDereferenceObject(eproc);
        }
    }
}

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

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    EnumProcess();
```

```
    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

输出效果图如下所示：



内核中实现枚举线程

内核线程的枚举与枚举进程十分相似，内核线程中也存在一个`ETHREAD`结构，但在枚举线程之前需要先来枚举到指定进程的`eprocess`结构，然后再根据`eprocess`结构对指定线程进行枚举。

在内核中实现枚举线程需要遵循以下步骤：

- 枚举指定进程的`eprocess`结构：在内核中，每个进程都有一个唯一的`eprocess`结构表示，该结构包含了该进程的各种信息，包括其线程列表。首先，需要枚举到指定进程的`eprocess`结构。可以通过访问系统的进程链表，找到该进程的`eprocess`结构。
- 遍历线程列表：一旦枚举到了指定进程的`eprocess`结构，就可以通过该结构中的线程列表来枚举该进程的所有线程。线程列表中包含每个线程的ETHREAD结构。
- 枚举每个线程的`ETHREAD`结构：遍历线程列表，对于每个线程，可以通过其`ETHREAD`结构访问该线程的各种信息，包括其状态、优先级、CPU时间等等。
- 处理枚举结果：枚举过程中可以将每个线程的`ETHREAD`结构存储到一个缓冲区中，以便后续处理。

需要注意的是，在枚举线程的过程中，需要保证访问的安全性和正确性。例如，需要确保在访问每个线程的`ETHREAD`结构时，该线程不会被销毁或修改。同时，还需要考虑内核与用户空间的交互，以及多处理器系统中的并发访问等问题。

为了能写出完整的代码，这里我们还需要介绍三个未导出函数，`PsGetProcessImageFileName`，`PsLookupThreadByThreadId`，`IoThreadToProcess`这三个函数是实现枚举线程的关键，它们提供了枚举线程相关的关键功能；

PsGetProcessImageFileName 函数的作用是获取指定进程的可执行文件名。在枚举线程时，可以使用该函数获取线程所属进程的可执行文件名，从而可以更方便地识别线程。

```
NTKERNELAPI PCHAR PsGetProcessImageFileName(IN PEPROCESS Process)
```

其中，`PEPROCESS`是一个指向进程对象的指针，该函数将返回一个指向进程可执行文件名的指针。

PsLookupThreadByThreadId 函数的作用是根据线程ID查找线程对象。在枚举线程时，可以使用该函数根据线程ID获取线程对象的指针，进而获取线程的相关信息。

```
NTKERNELAPI PETHREAD PsLookupThreadByThreadId(IN HANDLE ThreadId)
```

其中，`HANDLE`是一个线程ID，该函数将返回一个指向线程对象的指针。

IoThreadToProcess 函数的作用是获取线程所属进程的指针。在枚举线程时，可以使用该函数获取线程所属进程的指针，进而获取进程的相关信息。

```
NTKERNELAPI PEPROCESS IoThreadToProcess(IN PETHREAD Thread)
```

其中，`PETHREAD`是一个指向线程对象的指针，该函数将返回一个指向线程所属进程的指针。

有了上述三个函数的支持，那么实现枚举线程就变得非常简单了，`EnumThread`则是用于实现线程枚举的核心代码；

- 首先，定义了一个用于循环遍历线程ID的变量i，并且初始化为4，因为Windows系统的线程ID从4开始。
- 定义了两个指针类型的变量ethrd和eproc，用于保存获取到的线程对象和线程所属进程对象的指针。
- 循环遍历线程ID，每次增加4，直到262144为止。这个范围应该是保守估计，实际上可能更小，因为一般来说系统中并不会存在那么多的线程。
- 调用LookupThread函数，根据线程ID查找线程对象。如果找到了线程对象，则获取线程所属进程对象的指针，并且判断该进程对象是否与指定的进程对象相同。
- 如果是指定的进程对象，则打印出线程对象和线程ID。最后释放线程对象的引用计数。

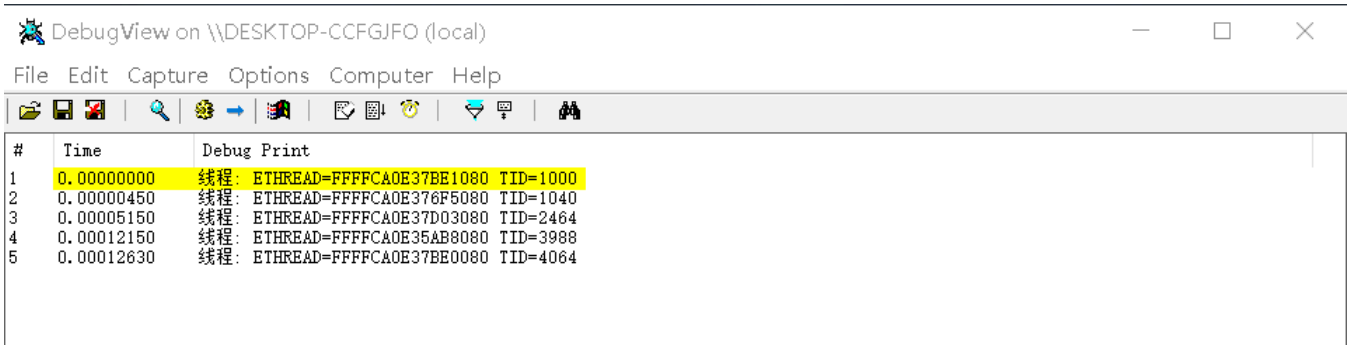
其完整实现代码如下所示；

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

// 声明API
NTKERNELAPI UCHAR* PsGetProcessImageFileName(IN PEPROCESS Process);
NTKERNELAPI NTSTATUS PsLookupProcessByProcessId(HANDLE Id, PEPROCESS *Process);
NTKERNELAPI NTSTATUS PsLookupThreadByThreadId(HANDLE Id, PETHREAD *Thread);
NTKERNELAPI PEPROCESS IoThreadToProcess(PETHREAD Thread);

// 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;
}

// 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;
}

// 枚举指定进程中的线程
VOID EnumThread(PEPROCESS Process)
{
    ULONG i = 0, c = 0;
    PETHREAD ethrd = NULL;
    PEPROCESS eproc = NULL;

  // 一般来说没有超过100000的PID和TID
    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("线程: ETHREAD=%p TID=%ld\n",ethrd,(ULONG)PsGetThreadId(ethrd));
            }
            ObDereferenceObject(ethrd);
        }
    }
}

// 通过枚举的方式定位到指定的进程，这里传递一个进程名称
VOID MyEnumThread(char *ProcessName)
{
    ULONG i = 0;
    PEPROCESS eproc = NULL;
    for (i = 4; i<100000000; i = i + 4)
    {
        eproc = LookupProcess((HANDLE)i);
        if (eproc != NULL)
        {
            ObDereferenceObject(eproc);
            if (strstr(PsGetProcessImageFileName(eproc), ProcessName) != NULL)
            {
        // 相等则说明是我们想要的进程，直接枚举其中的线程
                EnumThread(eproc);
            }
        }
    }
}

VOID DriverUnload(IN PDRIVER_OBJECT DriverObject)
{

}
```

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)
{
    MyEnumThread("lyshark.exe");
    DriverObject->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

输出效果图如下所示：



内核中实现枚举进程模块

枚举进程中的所有模块信息，DLL模块信息被记录在`PEB`的`LDR`链表里，LDR是一个双向链表枚举链表即可。

在操作系统内核中实现枚举进程模块的过程中，需要首先访问进程的PEB（进程环境块）数据结构。PEB是一个系统数据结构，记录了进程的各种信息，包括进程的内存布局、环境变量、进程的模块列表等。

进程的模块信息被记录在PEB的LDR（Loader）链表中。这个链表是一个双向链表，记录了进程的所有模块，包括已加载和未加载的模块。

要枚举进程中的所有模块信息，需要遍历LDR链表。在遍历LDR链表时，可以通过遍历双向链表中的节点来获取每个模块的详细信息，如模块的基址、模块的大小、模块的名称等。

遍历LDR链表的过程中，可以使用双向链表的常见操作，如while循环遍历，或使用指针的操作来访问下一个或上一个节点。在访问每个节点时，可以通过节点的指针访问节点中记录的模块信息，例如通过节点的指针访问模块的基址、大小、名称等信息。

通过枚举LDR链表，可以获取进程中的所有模块信息，并且可以在内核中对这些模块进行操作，如卸载模块、加载模块等。

在开始实现枚举进程模块之前，我们需要手动寻找`peb.ldr`以及`peb.ldr.InLoadOrderModuleList`的实际偏移地址，该偏移地址在不同的系统内是不同的，通过`WinDBG`调试Windows系统，并输入如下命令，即可找到我们所需的内核偏移值；

```
1: kd> dt _PEB
ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 BitField         : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   +0x003 IsProtectedProcess : Pos 1, 1 Bit
   +0x003 IsImageDynamicallyRelocated : Pos 2, 1 Bit
   +0x003 SkipPatchingUser32Forwarders : Pos 3, 1 Bit
```

```
   +0x003 IsPackagedProcess : Pos 4, 1 Bit
   +0x003 IsAppContainer    : Pos 5, 1 Bit
   +0x003 IsProtectedProcessLight : Pos 6, 1 Bit
   +0x003 IsLongPathAwareProcess : Pos 7, 1 Bit
   +0x004 Padding0          : [4] UChar
   +0x008 Mutant            : Ptr64 Void
   +0x010 ImageBaseAddress  : Ptr64 Void
   +0x018 Ldr               : Ptr64 _PEB_LDR_DATA                    // LDR结构
   +0x020 ProcessParameters : Ptr64 _RTL_USER_PROCESS_PARAMETERS

1: kd> dt _PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
   +0x000 Length            : Uint4B
   +0x004 Initialized       : UChar
   +0x008 SsHandle          : Ptr64 Void
   +0x010 InLoadOrderModuleList : _LIST_ENTRY                    // 链表结构
   +0x020 InMemoryOrderModuleList : _LIST_ENTRY
   +0x030 InInitializationOrderModuleList : _LIST_ENTRY
   +0x040 EntryInProgress   : Ptr64 Void
   +0x048 ShutdownInProgress : UChar
   +0x050 ShutdownThreadId : Ptr64 Void
```

获取到这两个关键偏移值以后，接下来就是封装 `EnumModule` 实现函数了，如下方核心代码的核心是在内核模式下枚举指定进程的模块列表，并打印每个模块的基址、大小和路径。它首先获取指定进程的 PEB，然后通过访问进程的 Ldr 数据结构获取模块列表信息，并使用 ProbeForRead 函数测试访问内存的可读性。

通过循环将所有的 `Module` 格式化为 `PLDR_DATA_TABLE_ENTRY` 结构并打印每个模块的信息，输出结束后取消对进程的依附，以此来实现枚举进程内所有的加载模块信息；

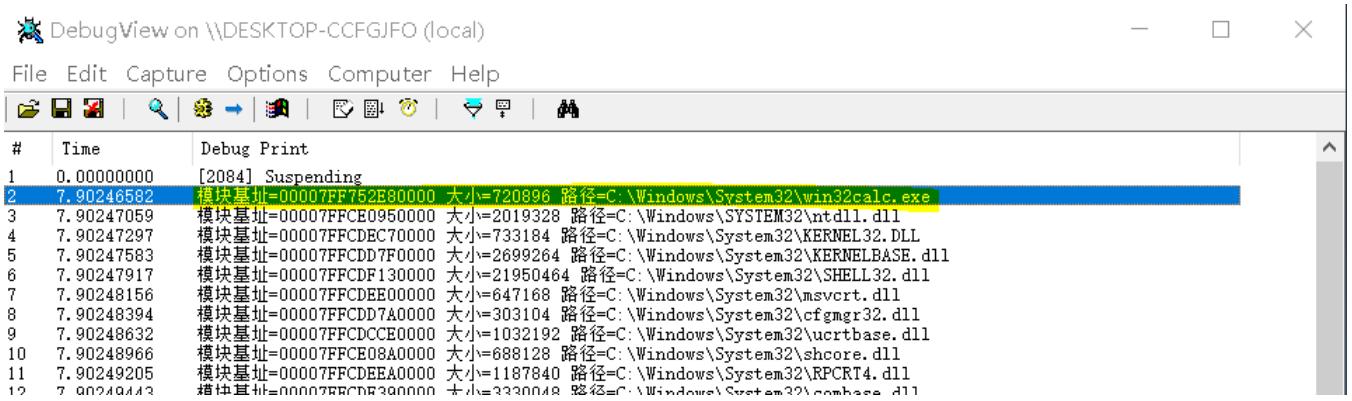
```
#include <ntddk.h>
#include <windef.h>

// 声明结构体
typedef struct _KAPC_STATE
{
    LIST_ENTRY ApcListHead[2];
    PKPROCESS Process;
    UCHAR KernelApcInProgress;
    UCHAR KernelApcPending;
    UCHAR UserApcPending;
} KAPC_STATE, *PKAPC_STATE;

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;

// peb.ldr
ULONG64 LdrInPebOffset = 0x018;

// peb.ldr.InLoadOrderModuleList
ULONG64 ModListInPebOffset = 0x010;

// 声明API
NTKERNELAPI UCHAR* PsGetProcessImageFileName(IN PEPROCESS Process);
NTKERNELAPI PPEB PsGetProcessPeb(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;
}

// 枚举指定进程的模块
VOID EnumModule(PEPROCESS Process)
{
    SIZE_T Peb = 0;
    SIZE_T 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 = (SIZE_T)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 + (SIZE_T)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("模块基址=%p 大小=%ld 路径=%wZ\n",(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){;}

  // 取消依附进程
    KeUnstackDetachProcess(&ks);
}

// 通过枚举的方式定位到指定的进程，这里传递一个进程名称
VOID MyEnumModule(char *ProcessName)
{
    ULONG i = 0;
    PEPROCESS eproc = NULL;
    for (i = 4; i<100000000; i = i + 4)
    {
        eproc = LookupProcess((HANDLE)i);
        if (eproc != NULL)
        {
```

```
            ObDereferenceObject(eproc);
            if (strstr(PsGetProcessImageFileName(eproc), ProcessName) != NULL)
            {
        // 相等则说明是我们想要的进程，直接枚举其中的线程
                EnumModule(eproc);
            }
        }
    }
}

VOID DriverUnload(IN PDRIVER_OBJECT DriverObject)
{

}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)
{
    MyEnumModule("calc.exe");
    DriverObject->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

输出效果图如下所示：



内核中实现枚举加载的驱动

内核中的SYS文件也是通过双向链表的方式相连接的，我们可以通过遍历驱动自身`LDR_DATA_TABLE_ENTRY`结构(遍历自身DriverSection成员)，就能够得到全部的模块信息。

在操作系统内核中，SYS文件通常作为设备驱动程序的一部分加载到内存中。为了管理这些模块，Windows使用了一个双向链表来维护已加载模块的信息。链表中的每个节点是一个`LDR_DATA_TABLE_ENTRY`结构，它包含了模块的各种信息，如模块名、模块基地址、模块大小、模块导入表等等。

当一个SYS文件被加载到内存中时，系统会创建一个`LDR_DATA_TABLE_ENTRY`结构并将其插入到内核模块列表的末尾。在插入时，系统会将新节点的前一个节点的`ForwardLink`指向新节点，将新节点的`BackLink`指向前一个节点，并将新节点的`ForwardLink`指向链表尾部的哨兵节点。

遍历内核模块列表时，可以通过遍历`LDR_DATA_TABLE_ENTRY`结构中的`DriverSection`成员，找到所有已加载的SYS文件，并获得它们的基本信息。从链表头部开始遍历链表，可以使用`ForwardLink`指针来访问下一个节点，直到访问到链表尾部的哨兵节点为止。

如下代码中，在`DriverEntry()`开始处，定义了一些变量，包括`pLdr、pListEntry、pModule`和`pCurrentListEntry`，它们分别代表当前驱动程序的`LDR_DATA_TABLE_ENTRY`结构、模块列表中的链表头、当前模块的`LDR_DATA_TABLE_ENTRY`结构和当前遍历到的链表节点。

接着，使用`DriverObject->DriverSection`获取当前驱动程序的`LDR_DATA_TABLE_ENTRY`结构，并通过`pLdr->InLoadOrderLinks.Flink`获取模块列表中的链表头。使用`pListEntry->Flink`获取链表中的第一个节点，并将其赋值给`pCurrentListEntry`。

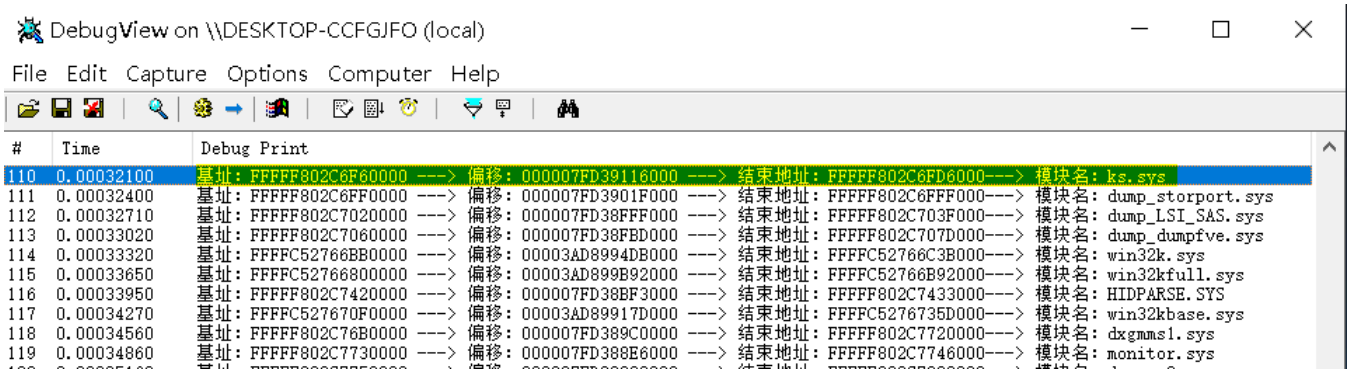
之后，通过一个循环遍历整个模块列表。在每次循环中，使用`CONTAINING_RECORD`宏获取当前节点对应的`LDR_DATA_TABLE_ENTRY`结构，并检查该模块的基本信息是否为空。如果不为空，将该模块的基址、结束地址、大小和模块名打印到调试窗口中。

最后，在函数结尾处设置了驱动程序的卸载例程`DriverUnload`，并返回`STATUS_SUCCESS`表示函数执行成功，至此枚举内核模块就完成了，其完整代码如下；

```
#include <ntddk.h>
#include <wdm.h>

typedef struct _LDR_DATA_TABLE_ENTRY {
    LIST_ENTRY InLoadOrderLinks;
    LIST_ENTRY InMemoryOrderLinks;
    LIST_ENTRY InInitializationOrderLinks;
    PVOID DllBase;
    PVOID EntryPoint;
    ULONG SizeOfImages;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT TlsIndex;
    union {
        LIST_ENTRY HashLinks;
        struct {
            PVOID SectionPointer;
            ULONG CheckSum;
        };
    };
    union {
        struct {
            ULONG TimeDateStamp;
        };
        struct {
            PVOID LoadedImports;
        };
    };
}LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;

VOID DriverUnload(IN PDRIVER_OBJECT DriverObject)
{

}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
```

```
{
    ULONG count = 0;
    NTSTATUS Status;
    DriverObject->DriverUnload = DriverUnload;

    PLDR_DATA_TABLE_ENTRY pLdr = NULL;
    PLIST_ENTRY pListEntry = NULL;
    PLDR_DATA_TABLE_ENTRY pModule = NULL;
    PLIST_ENTRY pCurrentListEntry = NULL;

    pLdr = (PLDR_DATA_TABLE_ENTRY)DriverObject->DriverSection;
    pListEntry = pLdr->InLoadOrderLinks.Flink;
    pCurrentListEntry = pListEntry->Flink;

    while (pCurrentListEntry != pListEntry)
    {
        pModule = CONTAINING_RECORD(pCurrentListEntry, LDR_DATA_TABLE_ENTRY,
InLoadOrderLinks);
        if (pModule->BaseDllName.Buffer != 0)
        {
            DbgPrint("基址：%p ---> 偏移：%p ---> 结束地址：%p---> 模块名：%wZ \r\n", pModule-
>DllBase, pModule->SizeOfImages - (LONGLONG)pModule->DllBase,
                (LONGLONG)pModule->DllBase + pModule->SizeOfImages,pModule->BaseDllName);
        }
        pCurrentListEntry = pCurrentListEntry->Flink;
    }

    DriverObject->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

输出效果图如下所示：



内核中实现获取特定进程PID

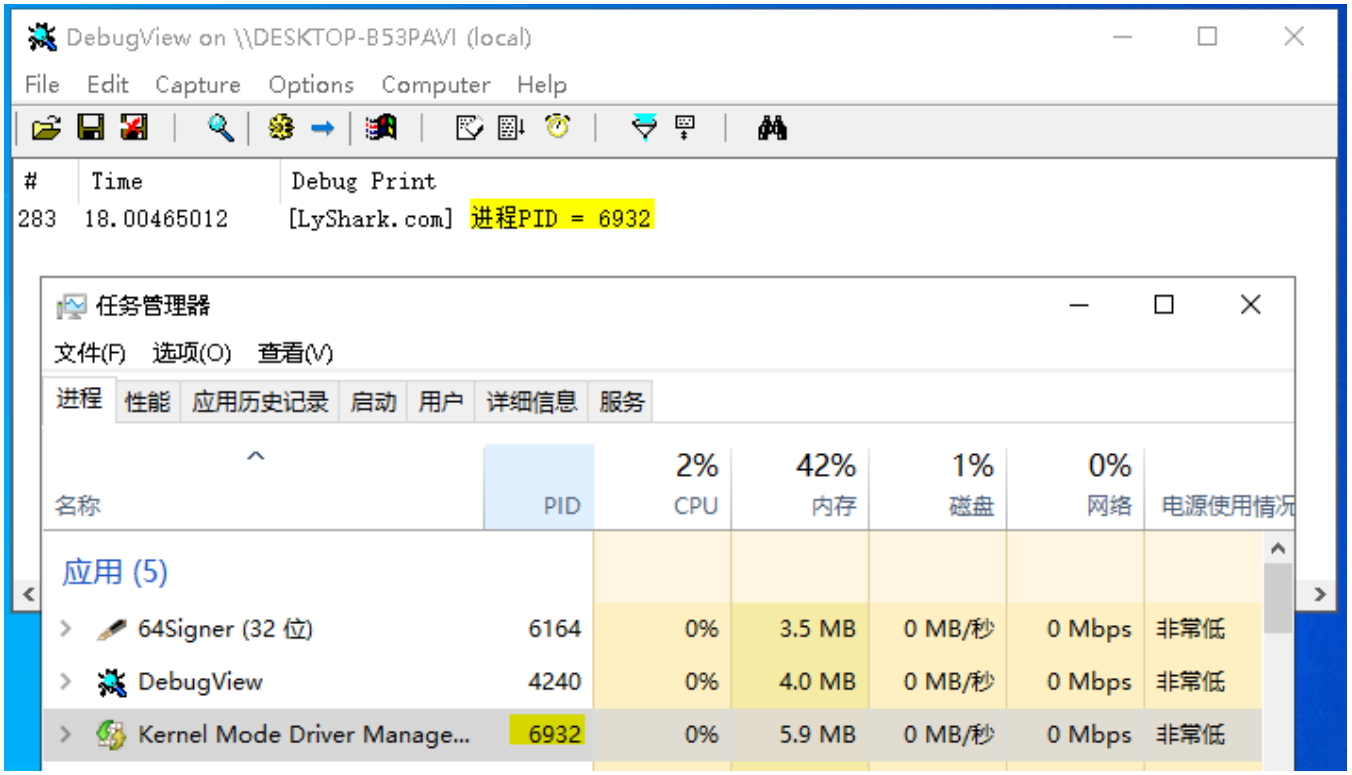
用户传入指定进程名称，调用`GetPidByProcessName()`可得到该进程名称所对应的进程PID号。

这段代码其大多数功能实现已经在前面的章节中实现了，需要注意的是`GetProcessID()`函数内部，通过`strstr(PsGetProcessImageFileName(eproc), ProcessName) != NULL`对比如果是我们所需要提取的进程结构，则直接`PsGetProcessId(eproc)`返回该进程的PID号。

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

```
#include <windef.h>

// 声明API
NTKERNELAPI UCHAR* PsGetProcessImageFileName(IN PEPROCESS Process);
NTKERNELAPI PPEB PsGetProcessPeb(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;
}

// 根据用户传入进程名得到该进程PID
HANDLE GetProcessID(char *ProcessName)
{
  ULONG i = 0;
  PEPROCESS eproc = NULL;
  for (i = 4; i<100000000; i = i + 4)
  {
    eproc = LookupProcess((HANDLE)i);
    if (eproc != NULL)
    {
      ObDereferenceObject(eproc);
      if (strstr(PsGetProcessImageFileName(eproc), ProcessName) != NULL)
      {
        return PsGetProcessId(eproc);
      }
    }
  }
  return NULL;
}

VOID DriverUnload(IN PDRIVER_OBJECT DriverObject)
{

}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)
{
  HANDLE ref = GetProcessID("KmdManager.exe");
  DbgPrint("[LyShark.com] 进程PID = %d \n", ref);

  DriverObject->DriverUnload = DriverUnload;
  return STATUS_SUCCESS;
}
```

输出效果图如下所示：



内核中实现判断进程状态

内核中实现判断进程状态的方法，通过传入一个 `EProcess` 结构体来判断指定进程的状态，包括进程是否存在、是否为僵尸进程等。这些功能通常被用于反内核工具的开发。

接下来，将逐个介绍并实现几个相关的功能，包括 `IsProcessDie` 函数用于验证进程空间是否有效，`IsRealProcess` 函数用于验证进程是否是真实进程，以及 `GetProcessCreateTime` 函数用于获取进程创建时间戳等功能。

IsProcessDie 函数用于验证特定进程空间是否有效，函数接受一个 `PEPROCESS` 类型的参数 `EProcess`，表示待验证的进程。函数会检查传入的 `EProcess` 参数是否为有效地址，并且会检查进程对象表的地址是否为有效地址。如果传入的参数或进程对象表地址无效，函数将返回 TRUE，表示进程空间已经无效或不存在。反之，如果地址有效，函数将返回 FALSE，表示进程空间有效。

函数的执行步骤如下：

- 首先判断 `MmIsAddressValid` 函数是否存在且有效，如果无效则直接返回 TRUE，表示进程空间无效。
- 检查传入的 `EProcess` 参数是否为有效地址，如果地址无效则直接返回 TRUE，表示进程空间无效。
- 通过计算 `EProcess` 结构体中进程对象表的偏移量，并检查该地址是否为有效地址。如果进程对象表地址无效，表示进程空间已经无效或不存在，直接返回 TRUE。
- 如果传入的参数和进程对象表地址均为有效地址，则获取进程对象表指针并进行进一步检查。
- 如果进程对象表指针为 NULL 或者其地址无效，则表示进程空间已经无效或不存在，返回 TRUE，否则返回 FALSE，表示进程空间有效。

```
// 验证进程空间是否有效
BOOLEAN IsProcessDie(PEPROCESS EProcess)
{
  BOOLEAN bDie = FALSE;

  if (MmIsAddressValid &&
```

```
    EProcess &&
    MmIsAddressValid(EProcess) &&
    MmIsAddressValid((PVOID)((ULONG_PTR)EProcess + ObjectTableOffsetOf_EPROCESS)))
  {
    PVOID ObjectTable = *(PVOID*)((ULONG_PTR)EProcess + ObjectTableOffsetOf_EPROCESS);

    if (!ObjectTable || !MmIsAddressValid(ObjectTable))
    {
      bDie = TRUE;
    }
  }
  else
  {
    bDie = TRUE;
  }
  return bDie;
}
```

IsRealProcess 函数的功能是验证进程是否是僵尸进程。该函数接受一个 `PEPROCESS` 类型的参数 `EProcess`，表示待验证的进程。函数内部会先通过 `KeGetObjectType` 函数获取传入的进程对象的类型，然后将其与进程类型进行比较，如果相同且进程空间有效，则说明该进程不是僵尸进程，返回 TRUE，否则返回 FALSE。

在 `KeGetObjectType` 函数中，先判断输入参数是否为有效地址，如果无效则返回 NULL，表示取对象类型失败。如果地址有效，则通过 `GetFunctionAddressByName` 函数获取 `ObGetObjectType` 函数的地址，然后调用 `ObGetObjectType` 函数获取对象类型。最后将对象类型作为返回值返回。

在`IsRealProcess`主函数中，首先获取进程类型，然后检查传入的进程对象是否为有效地址。如果进程类型和获取的对象类型相同，且进程空间有效，则说明该进程不是僵尸进程，返回 TRUE。反之，如果进程对象无效或进程类型不匹配，则说明该进程是僵尸进程，返回 FALSE。

```
// 取出对象类型
ULONG_PTR KeGetObjectType(PVOID Object)
{
  ULONG_PTR ObjectType = NULL;
  pfnObGetObjectType ObGetObjectType = NULL;

  if (!MmIsAddressValid || !Object || !MmIsAddressValid(Object))
  {
    return NULL;
  }
  ObGetObjectType = (pfnObGetObjectType)GetFunctionAddressByName(L"ObGetObjectType");
  if (ObGetObjectType)
  {
    ObjectType = ObGetObjectType(Object);
  }
  return ObjectType;
}

// 验证进程是否是僵尸进程
BOOLEAN IsRealProcess(PEPROCESS EProcess)
{
  ULONG_PTR ObjectType;
```

```
  ULONG_PTR    ObjectTypeAddress;
  BOOLEAN bRet = FALSE;

  ULONG_PTR ProcessType = ((ULONG_PTR)*PsProcessType);

  if (ProcessType && MmIsAddressValid && EProcess && MmIsAddressValid((PVOID)(EProcess)))
  {
    ObjectType = KeGetObjectType((PVOID)EProcess);
    if (ObjectType &&
      ProcessType == ObjectType &&
      !IsProcessDie(EProcess))
    {
      bRet = TRUE;
    }
  }

  return bRet;
}
```

GetProcessCreateTime 函数用于获取指定进程的创建时间戳。通过调用 `PsLookupProcessByProcessId` 函数获取到进程对象，然后调用 `PsGetProcessCreateTimeQuadPart` 函数获取进程的创建时间戳。在获取时间戳之前，需要将当前线程的 `Previous Mode` 设置为内核模式，以便访问 `EPROCESS` 结构体中的成员。在获取时间戳之后，需要将 `Previous Mode` 恢复到之前的值，并释放进程对象。

函数的执行步骤如下：

- 通过调用 `PsLookupProcessByProcessId` 函数获取指定进程的进程对象。
- 调用 `PsGetCurrentThread` 函数获取当前线程的 `ETHREAD` 对象，调用 `ChangePreMode` 函数将当前线程的 `Previous Mode` 设置为内核模式，并保存之前的 `Previous Mode` 的值。
- 调用 `PsGetProcessCreateTimeQuadPart` 函数获取指定进程的创建时间戳，并将时间戳保存到 `OutputBuffer` 指向的缓冲区中。
- 最后调用 `RecoverPreMode` 函数将当前线程的 `Previous Mode` 恢复到之前的值，并释放进程对象。

```
// 获取进程时间戳
BOOLEAN GetProcessCreateTime(ULONG_PTR ProcessID, LONGLONG* OutputBuffer)
{
  NTSTATUS  Status;
  PEPROCESS EProcess = NULL;
  PETHREAD  EThread = NULL;
  CHAR      PreMode = 0;

  Status = PsLookupProcessByProcessId((HANDLE)ProcessID, &EProcess);
  if (!NT_SUCCESS(Status))
  {
    return FALSE;
  }

  EThread = PsGetCurrentThread();
  PreMode = ChangePreMode(EThread);
  *OutputBuffer = PsGetProcessCreateTimeQuadPart(EProcess);
  RecoverPreMode(EThread, PreMode);
```

```
  ObfDereferenceObject(EProcess);
  return TRUE;
}
```

我们将上述三个功能进行整合，并最终得到一段完整的代码，如下所示；

```
#include <ntifs.h>

ULONG_PTR ObjectTableOffsetOf_EPROCESS = 0;    // 句柄表偏移
ULONG_PTR PreviousModeOffsetOf_KTHREAD = 0;    // 权限相关的偏移

typedef ULONG_PTR(*pfnObGetObjectType)(PVOID pObject);

// 验证进程空间是否有效
BOOLEAN IsProcessDie(PEPROCESS EProcess)
{
    BOOLEAN bDie = FALSE;

    if (MmIsAddressValid &&
        EProcess &&
        MmIsAddressValid(EProcess) &&
        MmIsAddressValid((PVOID)((ULONG_PTR)EProcess + ObjectTableOffsetOf_EPROCESS)))
    {
        PVOID ObjectTable = *(PVOID*)((ULONG_PTR)EProcess + ObjectTableOffsetOf_EPROCESS);

        if (!ObjectTable || !MmIsAddressValid(ObjectTable))
        {
            bDie = TRUE;
        }
    }
    else
    {
        bDie = TRUE;
    }
    return bDie;
}

//通过 函数名称 得到函数地址
PVOID GetFunctionAddressByName(WCHAR *szFunction)
{
    UNICODE_STRING uniFunction;
    PVOID AddrBase = NULL;

    if (szFunction && wcslen(szFunction) > 0)
    {
        RtlInitUnicodeString(&uniFunction, szFunction);
        AddrBase = MmGetSystemRoutineAddress(&uniFunction);
    }
    return AddrBase;
}

// 取出对象类型
ULONG_PTR KeGetObjectType(PVOID Object)
```

```
{
    ULONG_PTR ObjectType = NULL;
    pfnObGetObjectType ObGetObjectType = NULL;

    if (!MmIsAddressValid || !Object || !MmIsAddressValid(Object))
    {
        return NULL;
    }
    ObGetObjectType = (pfnObGetObjectType)GetFunctionAddressByName(L"ObGetObjectType");
    if (ObGetObjectType)
    {
        ObjectType = ObGetObjectType(Object);
    }
    return ObjectType;
}

// 验证进程是否是僵尸进程
BOOLEAN IsRealProcess(PEPROCESS EProcess)
{
    ULONG_PTR ObjectType;
    ULONG_PTR    ObjectTypeAddress;
    BOOLEAN bRet = FALSE;

    ULONG_PTR ProcessType = ((ULONG_PTR)*PsProcessType);

    if (ProcessType && MmIsAddressValid && EProcess && MmIsAddressValid((PVOID)(EProcess)))
    {
        ObjectType = KeGetObjectType((PVOID)EProcess);
        if (ObjectType &&
            ProcessType == ObjectType &&
            !IsProcessDie(EProcess))
        {
            bRet = TRUE;
        }
    }

    return bRet;
}

CHAR ChangePreMode(PETHREAD EThread)
{
    CHAR PreMode = *(PCHAR)((ULONG_PTR)EThread + PreviousModeOffsetOf_KTHREAD);
    *(PCHAR)((ULONG_PTR)EThread + PreviousModeOffsetOf_KTHREAD) = KernelMode;
    return PreMode;
}

VOID RecoverPreMode(PETHREAD EThread, CHAR PreMode)
{
    *(PCHAR)((ULONG_PTR)EThread + PreviousModeOffsetOf_KTHREAD) = PreMode;
}

// 获取进程时间戳
BOOLEAN GetProcessCreateTime(ULONG_PTR ProcessID, LONGLONG* OutputBuffer)
```

```
{
    NTSTATUS  Status;
    PEPROCESS EProcess = NULL;
    PETHREAD  EThread = NULL;
    CHAR PreMode = 0;

    Status = PsLookupProcessByProcessId((HANDLE)ProcessID, &EProcess);
    if (!NT_SUCCESS(Status))
    {
        return FALSE;
    }

    EThread = PsGetCurrentThread();
    PreMode = ChangePreMode(EThread);
    *OutputBuffer = PsGetProcessCreateTimeQuadPart(EProcess);
    RecoverPreMode(EThread, PreMode);
    ObfDereferenceObject(EProcess);
    return TRUE;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载成功 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    NTSTATUS Status = STATUS_UNSUCCESSFUL;
    PEPROCESS EProcess = NULL;
    HANDLE pid = (HANDLE)6932;

    // 根据PID获取进程EProcess结构
    Status = PsLookupProcessByProcessId(pid, &EProcess);

    // 判断进程是否有效
    if (NT_SUCCESS(Status) && IsProcessDie(EProcess))
    {
        DbgPrint("[LyShark.com] 进程有效 \n");
    }

    // 判断是否为僵尸进程
    if (NT_SUCCESS(Status) && IsRealProcess(EProcess))
    {
        DbgPrint("[LyShark.com] 僵尸进程 \n");
    }

    // 验证进程时间戳
    LONGLONG time;
    BOOLEAN ref = GetProcessCreateTime(pid, &time);
    if (NT_SUCCESS(Status) && ref)
    {
        DbgPrint("[LyShark.com] 该进程时间戳: %x \n", time);
    }
```

```
    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

输出效果图如下所示: