在上一章《内核LDE64引擎计算汇编长度》中，LyShark教大家如何通过LDE64引擎实现计算反汇编指令长度，本章将在此基础之上实现内联函数挂钩，内核中的InlineHook函数挂钩其实与应用层一致，都是使用劫持执行流并跳转到我们自己的函数上来做处理，唯一的不同的是内核Hook只针对内核API函数，但由于其身处在最底层所以一旦被挂钩其整个应用层都将会受到影响，这就直接决定了在内核层挂钩的效果是应用层无法比拟的，对于安全从业者来说学会使用内核挂钩也是很重要。

内核挂钩的原理是一种劫持系统函数调用的技术，用于在运行时对系统函数进行修改或者监控。其基本思想是先获取要被劫持的函数的地址，然后将该函数的前15个字节的指令保存下来，接着将自己的代理函数地址写入到原始函数上，这样当API被调用时，就会默认转向到自己的代理函数上执行，从而实现函数的劫持。

挂钩的具体步骤如下：

- 1.使用MmGetSystemRoutineAddress函数获取要被劫持的函数地址。
- 2.使用自己的代理函数取代原始函数，代理函数和原始函数具有相同的参数和返回值类型，并且在代理函数中调用原始函数。
- 3.保存原始函数的前15个字节的指令，因为这些指令通常被认为是函数的前导码或是修饰码。
- 4.在原始函数的前15个字节位置写入jmp MyPsLookupProcessByProcessId的指令，使得API调用会跳转到我们的代理函数。
- 5.当代理函数被调用时，执行我们自己的逻辑，然后在适当的时候再调用原始函数，最后将其返回值返回给调用者。
- 6.如果需要恢复原始函数的调用，将保存的前15个字节的指令写回原始函数即可。

挂钩的原理可以总结为，通过MmGetSystemRoutineAddress得到原函数地址，然后保存该函数的前15个字节的指令，将自己的MyPsLookupProcessByProcessId代理函数地址写出到原始函数上，此时如果有API被调用则默认会转向到我们自己的函数上面执行，恢复原理则是将提前保存好的前15个原始字节写回则恢复原函数的调用。

而如果需要恢复挂钩状态，则只需要还原提前保存的机器码即可，恢复内核挂钩的原理是将先前保存的原始函数前15个字节的指令写回到原始函数地址上，从而还原原始函数的调用。具体步骤如下：

- 1.获取原函数地址，可以通过MmGetSystemRoutineAddress函数获取。
- 2.将保存的原始函数前15个字节的指令写回到原始函数地址上，可以使用memcpy等函数实现。
- 3.将代理函数的地址清除，可以将地址设置为NULL。

原理很简单，基本上InlineHook类的代码都是一个样子，如下是一段完整的挂钩PsLookupProcessByProcessId的驱动程序，当程序被加载时则默认会保护lyshark.exe进程，使其无法被用户使用任务管理器结束掉。

```
#include "lyshark_lde64.h"
#include <ntifs.h>
#include <windef.h>
#include <intrin.h>

#pragma  intrinsic(_disable)
#pragma  intrinsic(_enable)

// ----------------------------------------------------------------
// 汇编计算方法
// ----------------------------------------------------------------
// 计算地址处指令有多少字节
// address = 地址
// bits 32位驱动传入0 64传入64
```

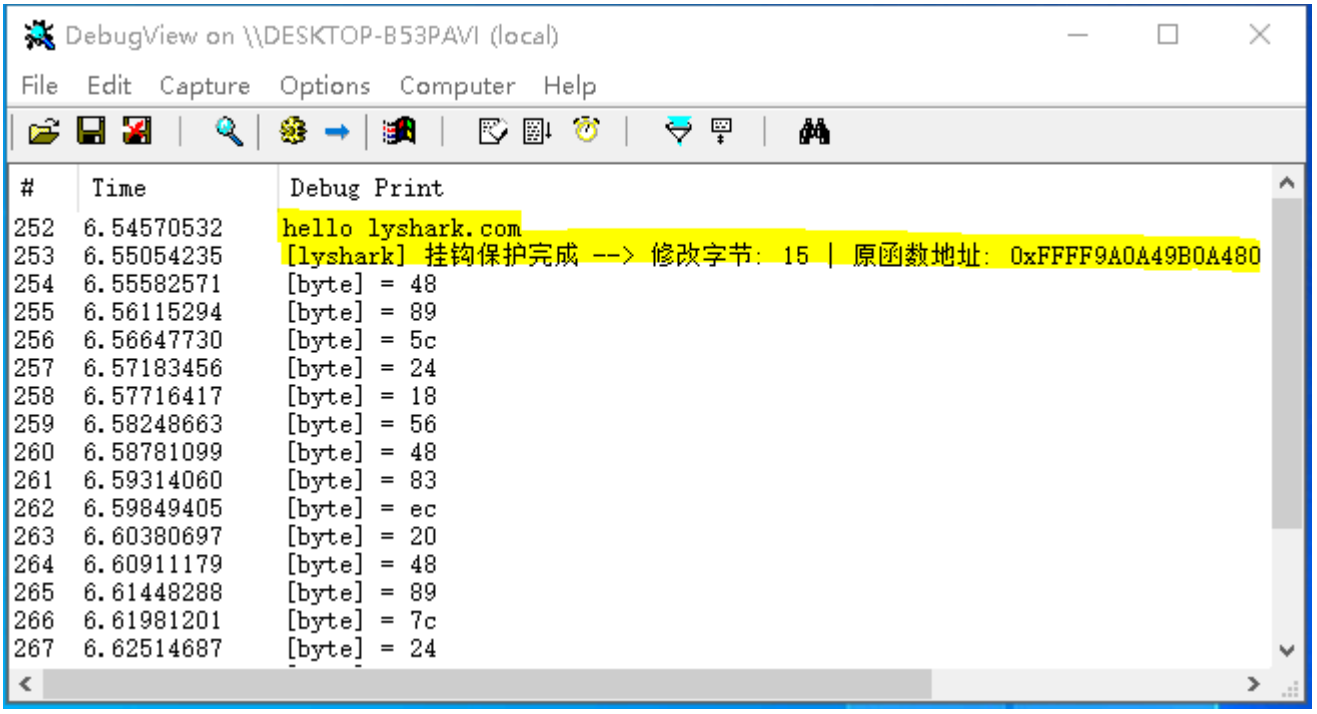
```
typedef INT(*LDE_DISASM)(PVOID address, INT bits);

LDE_DISASM lde_disasm;

// 初始化引擎
VOID lde_init()
{
    lde_disasm = ExAllocatePool(NonPagedPool, 12800);
    memcpy(lde_disasm, szShellCode, 12800);
}

// 得到完整指令长度,避免截断
ULONG GetFullPatchSize(PUCHAR Address)
{
    ULONG LenCount = 0, Len = 0;

    // 至少需要14字节
    while (LenCount <= 14)
    {
        Len = lde_disasm(Address, 64);
        Address = Address + Len;
        LenCount = LenCount + Len;
    }
    return LenCount;
}

// ------------------------------------------------------------------
// Hook函数封装
// ------------------------------------------------------------------

// 定义指针方便调用
typedef NTSTATUS(__fastcall *PSLOOKUPPROCESSBYPROCESSID)(HANDLE ProcessId, PEPROCESS
*Process);

ULONG64 protect_eprocess = 0;           // 需要保护进程的eprocess
ULONG patch_size = 0;                   // 被修改了几个字节
PUCHAR head_n_byte = NULL;              // 前几个字节数组
PVOID original_address = NULL;          // 原函数地址

KIRQL WPOFFx64()
{
    KIRQL irql = KeRaiseIrqlToDpcLevel();
    UINT64 cr0 = __readcr0();
    cr0 &= 0xfffffffffffeffff;
    __writecr0(cr0);
    _disable();
    return irql;
}

VOID WPONx64(KIRQL irql)
{
    UINT64 cr0 = __readcr0();
    cr0 |= 0x10000;
```

```
    _enable();
    __writecr0(cr0);
    KeLowerIrql(irql);
}

// 动态获取内存地址
PVOID GetProcessAddress(PCWSTR FunctionName)
{
    UNICODE_STRING UniCodeFunctionName;
    RtlInitUnicodeString(&UniCodeFunctionName, FunctionName);
    return MmGetSystemRoutineAddress(&UniCodeFunctionName);
}

/*
    InlineHookAPI 挂钩地址

    参数1：待HOOK函数地址
    参数2：代理函数地址
    参数3：接收原始函数地址的指针
    参数4：接收补丁长度的指针
    返回：原来头N字节的数据
*/
PVOID KernelHook(IN PVOID ApiAddress, IN PVOID Proxy_ApiAddress, OUT PVOID
*Original_ApiAddress, OUT ULONG *PatchSize)
{
    KIRQL irql;
    UINT64 tmpv;
    PVOID head_n_byte, ori_func;

    // 保存跳转指令 JMP QWORD PTR [本条指令结束后的地址]
    UCHAR jmp_code[] = "\xFF\x25\x00\x00\x00\x00\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF";

    // 保存原始指令
    UCHAR jmp_code_orifunc[] = "\xFF\x25\x00\x00\x00\x00\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF";

    // 获取函数地址处指令长度
    *PatchSize = GetFullPatchSize((PUCHAR)ApiAddress);

    // 分配空间
    head_n_byte = ExAllocatePoolWithTag(NonPagedPool, *PatchSize, "LyShark");

    irql = WPOFFx64();

    // 跳转地址拷贝到原函数上
    RtlCopyMemory(head_n_byte, ApiAddress, *PatchSize);
    WPONx64(irql);

    // 构建跳转

    // 1.原始机器码+跳转机器码
    ori_func = ExAllocatePoolWithTag(NonPagedPool, *PatchSize + 14, "LyShark");
    RtlFillMemory(ori_func, *PatchSize + 14, 0x90);
```

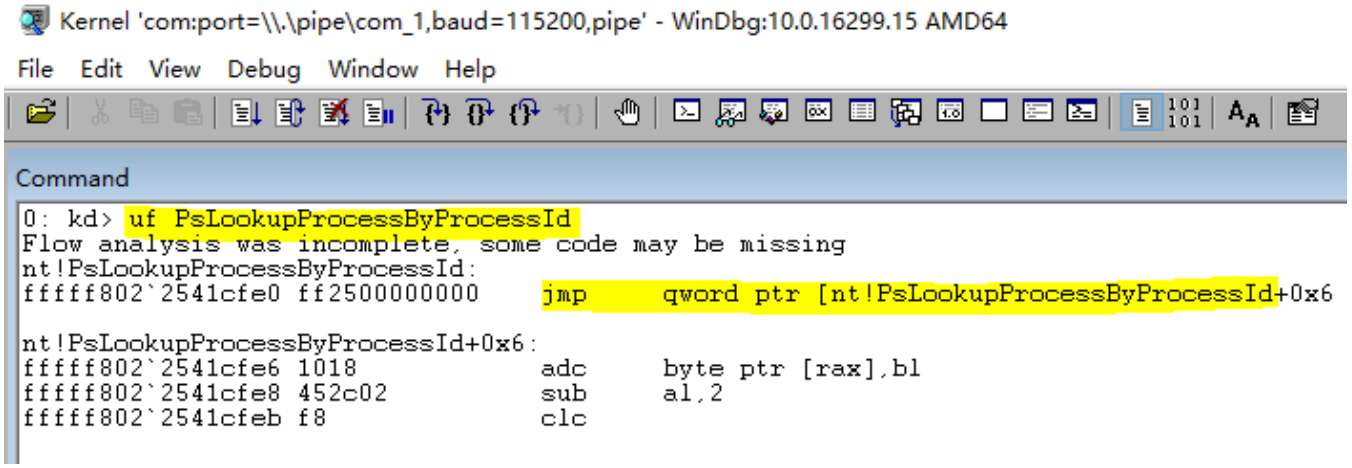
```
    // 2.跳转到没被打补丁的那个字节
    tmpv = (ULONG64)ApiAddress + *PatchSize;
    RtlCopyMemory(jmp_code_orifunc + 6, &tmpv, 8);
    RtlCopyMemory((PUCHAR)ori_func, head_n_byte, *PatchSize);
    RtlCopyMemory((PUCHAR)ori_func + *PatchSize, jmp_code_orifunc, 14);
    *Original_ApiAddress = ori_func;

    // 3.得到代理地址
    tmpv = (UINT64)Proxy_ApiAddress;
    RtlCopyMemory(jmp_code + 6, &tmpv, 8);

    //4.打补丁
    irql = WPOFFx64();
    RtlFillMemory(ApiAddress, *PatchSize, 0x90);
    RtlCopyMemory(ApiAddress, jmp_code, 14);
    WPONx64(irql);

    return head_n_byte;
}

/*
    InlineHookAPI 恢复挂钩地址

    参数1：被HOOK函数地址
    参数2：原始数据
    参数3：补丁长度
*/
VOID KernelUnHook(IN PVOID ApiAddress, IN PVOID OriCode, IN ULONG PatchSize)
{
    KIRQL irql;
    irql = WPOFFx64();
    RtlCopyMemory(ApiAddress, OriCode, PatchSize);
    WPONx64(irql);
}

// 实现我们自己的代理函数
NTSTATUS MyPsLookupProcessByProcessId(HANDLE ProcessId, PEPROCESS *Process)
{
    NTSTATUS st;
    st = ((PSLOOKUPPROCESSBYPROCESSID)original_address)(ProcessId, Process);
    if (NT_SUCCESS(st))
    {
        // 判断是否是需要保护的进程
        if (*Process == (PEPROCESS)protect_eprocess)
        {
            *Process = 0;
            DbgPrint("[lyshark] 拦截结束进程 \n");
            st = STATUS_ACCESS_DENIED;
        }
    }
    return st;
}
```

```
VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动已卸载 \n");

    // 恢复Hook
    KernelUnHook(GetProcessAddress(L"PsLookupProcessByProcessId"), head_n_byte, patch_size);
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    // 初始化反汇编引擎
    lde_init();

    // 设置需要保护进程EProcess
    /*
    lyshark.com: kd> !process 0 0 lyshark.exe
        PROCESS ffff9a0a44ec4080
            SessionId: 1  Cid: 05b8    Peb: 0034d000  ParentCid: 13f0
            DirBase: 12a7d2002  ObjectTable: ffffd60bc036f080  HandleCount: 159.
            Image: lyshark.exe
    */
    protect_eprocess = 0xffff9a0a44ec4080;

    // Hook挂钩函数
    head_n_byte = KernelHook(GetProcessAddress(L"PsLookupProcessByProcessId"),
(PVOID)MyPsLookupProcessByProcessId, &original_address, &patch_size);

    DbgPrint("[lyshark] 挂钩保护完成 --> 修改字节: %d | 原函数地址: 0x%p \n", patch_size,
original_address);

    for (size_t i = 0; i < patch_size; i++)
    {
        DbgPrint("[byte] = %x", head_n_byte[i]);
    }

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

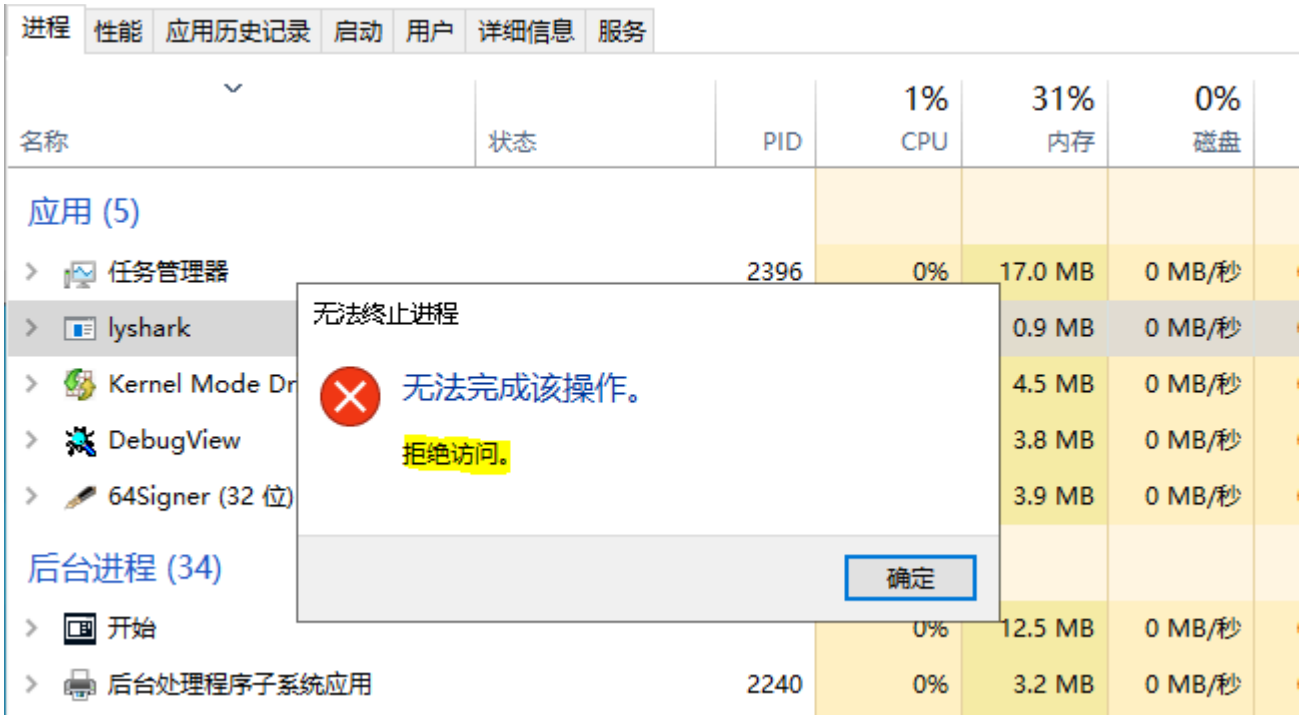
运行这段驱动程序，会输出挂钩保护的具体地址信息；



使用 `WinDBG` 观察，会发现挂钩后原函数已经被替换掉了，而被替换的地址就是我们自己的 `MyPsLookupProcessByProcessId` 函数。



当你尝试使用任务管理器结束掉 `lyshark.exe` 进程时，则会提示拒绝访问。


进程
性能
应用历史记录
启动
用户
详细信息
服务
名称
状态
PID
1%
CPU
31%
内存
0%
磁盘
应用 (5)
任务管理器
2396
0%
17.0 MB
0 MB/秒
lyshark
0.9 MB
0 MB/秒
Kernel Mode Dr
4.5 MB
0 MB/秒
DebugView
3.8 MB
0 MB/秒
64Signer (32 位)
3.9 MB
0 MB/秒
后台进程 (34)
开始
0%
12.5 MB
0 MB/秒
后台处理程序子系统应用
2240
0%
3.2 MB
0 MB/秒
无法终止进程
无法完成该操作。
拒绝访问。
确定