在上一章 《驱动开发：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。

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。

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。



// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com) #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++)

{

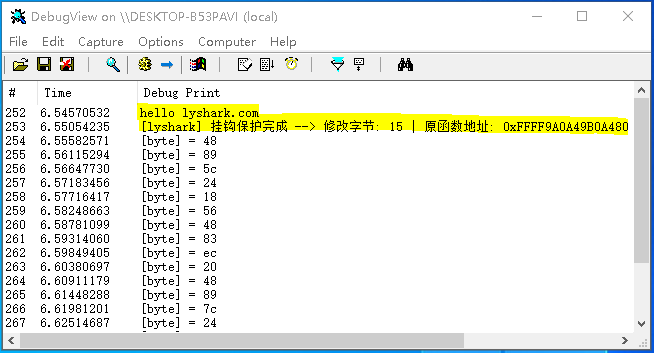
DbgPrint("[byte] = %x", head\_n\_byte[i]);

}

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

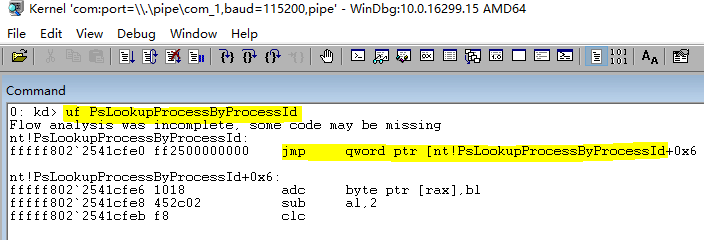
}

# 运行这段驱动程序，会输出挂钩保护的具体地址信息；

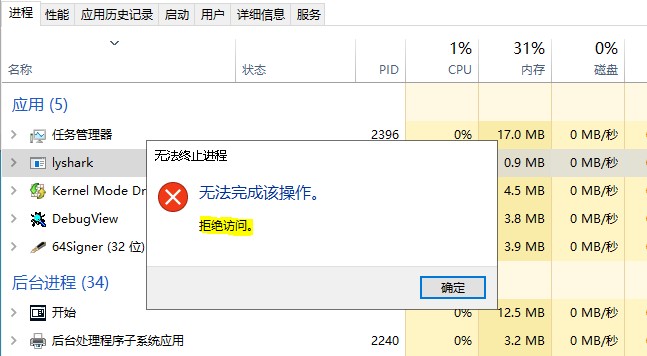


使用 WinDBG 观察，会发现挂钩后原函数已经被替换掉了，而被替换的地址就是我们自己的

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。



**参考文献**

# h[ttps://www.docin.com/p-1508418694.htm](http://www.docin.com/p-1508418694.html)l

作者： 王瑞 (LyShark)

作者邮箱： m [e@lyshark.com](mailto:e@lyshark.com)

版权声明：本博客文章与代码均为学习时整理的笔记，文章 [均为原创] 作品，转载文章请遵守

《中华人民共和国著作权法》相关法律规定或遵守《署名CC BY-ND 4.0国际》规范，合理合规携带原创出处转载，如果不携带文章出处，并恶意转载多篇原创文章被本人发现，本人保留起诉权！