

在笔者上一篇文章《内核层InlineHook挂钩函数》中介绍了通过替换函数头部代码的方式实现 Hook 挂钩，对于 ARK 工具来说实现扫描与摘除 InlineHook 钩子也是最基本的功能，此类功能的实现一般可在应用层进行，而驱动层只需要保留一个读写字节的函数即可，将复杂的流程放在应用层实现是一个非常明智的选择，与《内核实现进程反汇编》中所使用的读写驱动基本一致，本篇文章中的驱动只保留两个功能，控制信号 IOCTL_GET_CUR_CODE 用于读取函数的前 16 个字节的内存，信号 IOCTL_SET_ORI_CODE 则用于设置前 16 个字节的内存。

之所以是前 16 个字节是因为一般的内联 Hook 只需要使用两条指令就可实现劫持，如下是通用 ARK 工具扫描到的被挂钩函数的样子。

序号	函数名称	当前函数地址	Hook	原始函数地址	当前函数地址所在模块
165	NtCreateEnlistment	0xFFFF8051B19C790	Inline Hook	0xFFFF8051B19C790	C:\Windows\system32\ntoskrnl.exe
185	NtCreateResourceManager	0xFFFF8051B19C7B0	Inline Hook	0xFFFF8051B19C7B0	C:\Windows\system32\ntoskrnl.exe
194	反汇编器				
195	地址:	0xFFFF8051B19C790	大小(字节)	0xC8	<input type="button" value="反汇编"/>
220	地址	二进制	汇编		
231	0xFFFF8051B19C790	4C 8B 15 D1 3E 3D 00	mov r10, qword ptr ds:[0xFFFF8051B5706]		
234	0xFFFF8051B19C797	E9 14 A5 15 01	jmp 0xFFFF8051C2F6CB0		
245	0xFFFF8051B19C79C	CC	int3		
279	0xFFFF8051B19C79D	CC	int3		
293					
299					
300					
302					

首先将内核驱动程序代码放到如下，内核驱动程序没有任何特别的，仅仅只是一个通用驱动模板，在其基础上使用 CR3 读写，如果不理解 CR3 读写的原理您可以去看《内核 CR3 切换读写内存》这一篇中的详细介绍。

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

#define DEVICE_NAME           L"\Device\WinDDK"
#define LINK_NAME              L"\DosDevices\WinDDK"
#define LINK_GLOBAL_NAME       L"\DosDevices\Global\WinDDK"

// 控制信号 IOCTL_GET_CUR_CODE 用于读 | IOCTL_SET_ORI_CODE 用于写
#define IOCTL_GET_CUR_CODE    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_SET_ORI_CODE    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801, METHOD_BUFFERED,
FILE_ANY_ACCESS)

// 引用 __readcr0 等函数必须增加
#pragma intrinsic(_disable)
#pragma intrinsic(_enable)

// 定义读写结构体
typedef struct
{
    PVOID Address;
    ULONG64 Length;
    UCHAR data[256];
} KF_DATA, *PKF_DATA;
```

```
KIRQL g_irql;

// 关闭写保护
void WPOFFx64()
{
    ULONG64 cr0;
    g_irql = KeRaiseIrqlToDpcLevel();
    cr0 = __readcr0();
    cr0 &= 0xfffffffffffffeffff;
    __writecr0(cr0);
    _disable();
}

// 开启写保护
void WPONx64()
{
    ULONG64 cr0;
    cr0 = __readcr0();
    cr0 |= 0x10000;
    _enable();
    __writecr0(cr0);
    KeLowerIrql(g_irql);
}

// 设备创建时触发
NTSTATUS DispatchCreate(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    pIrp->IoStatus.Status = STATUS_SUCCESS;
    pIrp->IoStatus.Information = 0;

    DbgPrint("[Lyshark] 设备已创建 \n");
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

// 设备关闭时触发
NTSTATUS DispatchClose(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    pIrp->IoStatus.Status = STATUS_SUCCESS;
    pIrp->IoStatus.Information = 0;

    DbgPrint("[Lyshark] 设备已关闭 \n");
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

// 主派遣函数
NTSTATUS DispatchIoctl(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    NTSTATUS status = STATUS_INVALID_DEVICE_REQUEST;
    PIO_STACK_LOCATION pIrpStack;
    ULONG uIoControlCode;
    PVOID pIoBuffer;
```

```
ULONG uInSize;
ULONG uOutSize;

// 获取当前设备栈
pIrpStack = IoGetCurrentIrpStackLocation(pIrp);
uIoControlCode = pIrpStack->Parameters.DeviceIoControl.IoControlCode;

// 获取缓冲区
pIoBuffer = pIrp->AssociatedIrp.SystemBuffer;

// 获取缓冲区长度
uInSize = pIrpStack->Parameters.DeviceIoControl.InputBufferLength;

// 输出缓冲区长度
uOutSize = pIrpStack->Parameters.DeviceIoControl.OutputBufferLength;

switch (uIoControlCode)
{
    // 读内存
    case IOCTL_GET_CUR_CODE:
    {
        KF_DATA dat = { 0 };

        // 将缓冲区格式化为KF_DATA结构体
        RtlCopyMemory(&dat, pIoBuffer, 16);
        WPOFFx64();

        // 将数据写回到缓冲区
        RtlCopyMemory(pIoBuffer, dat.Address, dat.Length);
        WPONx64();
        status = STATUS_SUCCESS;
        break;
    }
    // 写内存
    case IOCTL_SET_ORI_CODE:
    {
        KF_DATA dat = { 0 };

        // 将缓冲区格式化为KF_DATA结构体
        RtlCopyMemory(&dat, pIoBuffer, sizeof(KF_DATA));
        WPOFFx64();

        // 将数据写回到缓冲区
        RtlCopyMemory(dat.Address, dat.data, dat.Length);
        WPONx64();
        status = STATUS_SUCCESS;
        break;
    }
}

if (status == STATUS_SUCCESS)
    pIrp->IoStatus.Information = uOutSize;
else
```

```
    pIrp->IoStatus.Information = 0;

    pIrp->IoStatus.Status = status;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    return status;
}

// 驱动卸载
VOID DriverUnload(PDRIVER_OBJECT pDriverObj)
{
    UNICODE_STRING strLink;

    // 删除符号链接卸载设备
    RtlInitUnicodeString(&strLink, LINK_NAME);
    IoDeleteSymbolicLink(&strLink);
    IoDeleteDevice(pDriverObj->DeviceObject);
}

// 驱动程序入口
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObj, PUNICODE_STRING pRegistryString)
{
    NTSTATUS status = STATUS_SUCCESS;
    UNICODE_STRING ustrLinkName;
    UNICODE_STRING ustrDevName;
    PDEVICE_OBJECT pDevObj;

    // 初始化派遣函数
    pDriverObj->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
    pDriverObj->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;
    pDriverObj->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchIoctl;

    DbgPrint("hello lysahrk.com \n");

    // 初始化设备名
    RtlInitUnicodeString(&ustrDevName, DEVICE_NAME);

    // 创建设备
    status = IoCreateDevice(pDriverObj, 0, &ustrDevName, FILE_DEVICE_UNKNOWN, 0, FALSE,
    &pDevObj);
    if (!NT_SUCCESS(status))
    {
        return status;
    }

    // 创建符号链接
    RtlInitUnicodeString(&ustrLinkName, LINK_NAME);
    status = IoCreateSymbolicLink(&ustrLinkName, &ustrDevName);
    if (!NT_SUCCESS(status))
    {
        IoDeleteDevice(pDevObj);
        return status;
    }
}
```

```

pDriverObj->DriverUnload = DriverUnload;
return STATUS_SUCCESS;
}

```

接着来分析下应用层做了什么，首先 `GetKernelBase64` 函数的作用，该函数内部通过 `GetProcAddress()` 函数动态寻找到 `ZwQuerySystemInformation()` 函数的内存地址（此函数未被到处所以只能动态找到），找到后调用 `ZwQuerySystemInformation()` 直接拿到系统中的所有模块信息，通过 `pSystemModuleInformation->Module[0].Base` 得到系统中第一个模块的基址，此模块就是 `ntoskrnl.exe`，该模块也是系统运行后的第一个启动的，此时我们即可拿到 `KernelBase` 也就是系统内存中的基地址。

进程	驱动模块	内核	内核钩子	应用层钩子	网络	注册表	文件	启动信息	监控	系统杂项	关于
驱动名	基地址	大小		驱动对象						驱动路径	
ntoskrnl.exe	0xFFFFF8051B000000	0x00ab6000		-						C:\Windows\system32\ntoskrnl.exe	
hal.dll	0xFFFFF8051AF5D000	0x000a3000		-						C:\Windows\system32\hal.dll	
kd.dll	0xFFFFF8051C000000	0x0000b000		-						C:\Windows\system32\kd.dll	
mcupdate_Genui...	0xFFFFF8051C010000	0x00201000		-						C:\Windows\system32\mcupdate_GenuineIntel.dll	

此时通过 `LoadLibraryExA()` 函数动态加载，此时加载的是磁盘中的被 Hook 函数的所属模块，获得映射地址后将此地址装入 `hKernel` 变量内，此时我们拥有了内存中的 `KernelBase` 以及磁盘中加载的 `hKernel`，接着调用 `RepairRelocationTable()` 让两者的重定位表保持一致。

此时当用户调用 `GetSystemRoutineAddress()` 则执行如下流程，想要获取当前内存地址，则需要使用当前内存中的 `KernelBase` 模块基址加上通过 `GetProcAddress()` 动态获取到的磁盘基址中的函数地址减去磁盘中的基址，将内存中的 `KernelBase` 加上 磁盘中的相对偏移 就得到了当前内存中加载函数的实际地址。

- `address1 = KernelBase + (ULONG64)GetProcAddress(hKernel, "NtWriteFile") - (ULONG64)hKernel`
- `address2 = KernelBase - (ULONG64)hKernel + (ULONG64)GetProcAddress(hKernel, "NtWriteFile")`

调用 `GetOriginalMachineCode()` 则用于获取相对偏移地址，该地址的获取方式如下，用户传入一个 `Address` 当前地址，该地址减去 `KernelBase` 内存中的基址，然后再加上 `hKernel` 磁盘加载的基址来获取到相对偏移。

- `OffsetAddress = Address - KernelBase + hKernel`

有了这两条信息那么功能也就实现了，通过 `GetOriginalMachineCode()` 得到指定内存地址处原始机器码，通过 `GetCurrentMachineCode()` 得到当前内存机器码，两者通过 `memcmp()` 函数比对即可知是否被挂钩了，如果被挂钩则可以通过 CR3 切换将原始机器码覆盖到特定位置替换即可，这段程序的完整代码如下；

```

#include <stdio.h>
#include <windows.h>

#pragma comment(lib,"user32.lib")
#pragma comment(lib,"Advapi32.lib")

#ifndef NT_SUCCESS
#define NT_SUCCESS(status) ((NTSTATUS)(status) >= 0)
#endif

#define BYTE_ARRAY_LENGTH 16
#define SystemModuleInformation 11
#define STATUS_INFO_LENGTH_MISMATCH ((NTSTATUS)0xC0000004L)

typedef long(__stdcall *ZWQUERYSYSTEMINFORMATION)
(

```

```

    IN ULONG SystemInformationClass,
    IN PVOID SystemInformation,
    IN ULONG SystemInformationLength,
    IN PULONG ReturnLength OPTIONAL
);

typedef struct
{
    ULONG Unknow1;
    ULONG Unknow2;
    ULONG Unknow3;
    ULONG Unknow4;
    PVOID Base;
    ULONG Size;
    ULONG Flags;
    USHORT Index;
    USHORT NameLength;
    USHORT LoadCount;
    USHORT ModuleNameOffset;
    char ImageName[256];
} SYSTEM_MODULE_INFORMATION_ENTRY, *PSYSTEM_MODULE_INFORMATION_ENTRY;

typedef struct
{
    ULONG Count;
    SYSTEM_MODULE_INFORMATION_ENTRY Module[1];
} SYSTEM_MODULE_INFORMATION, *PSYSTEM_MODULE_INFORMATION;

typedef struct
{
    PVOID Address;
    ULONG64 Length;
    UCHAR data[256];
} KF_DATA, *PKF_DATA;

HANDLE hDriver = 0;
HMODULE hKernel = 0;
ULONG64 KernelBase = 0;
CHAR NtosFullName[260] = { 0 };

// 生成控制信号
DWORD CTL_CODE_GEN(DWORD lngFunction)
{
    return (FILE_DEVICE_UNKNOWN * 65536) | (FILE_ANY_ACCESS * 16384) | (lngFunction * 4) |
METHOD_BUFFERED;
}

// 发送控制信号的函数
BOOL IoControl(HANDLE hDrvHandle, DWORD dwIoControlCode, PVOID lpInBuffer, DWORD
nInBufferSize, PVOID lpOutBuffer, DWORD nOutBufferSize)
{
    DWORD lDrvRetSize;

```

```
    return DeviceIoControl(hDrvHandle, dwIoControlCode, lpInBuffer, nInBufferSize,
    lpOutBuffer, nOutBufferSize, &DrvRetSize, 0);
}

// 动态获取ntdll.dll模块的基地址
ULONG64 GetKernelBase64(PCHAR NtosName)
{
    ZWQUERYSYSTEMINFORMATION ZwQuerySystemInformation;
    PSYSTEM_MODULE_INFORMATION pSystemModuleInformation;
    ULONG NeedsSize, BufferSize = 0x5000;
    PVOID pBuffer = NULL;
    NTSTATUS Result;

    // 该函数只能通过动态方式得到地址
    ZwQuerySystemInformation =
    (ZWQUERYSYSTEMINFORMATION)GetProcAddress(GetModuleHandleA("ntdll.dll"),
    "ZwQuerySystemInformation");
    do
    {
        pBuffer = malloc(BufferSize);
        if (pBuffer == NULL) return 0;

        // 查询系统中的所有模块信息
        Result = ZwQuerySystemInformation(pSystemModuleInformation, pBuffer, BufferSize,
        &NeedsSize);
        if (Result == STATUS_INFO_LENGTH_MISMATCH)
        {
            free(pBuffer);
            BufferSize *= 2;
        }
        else if (!NT_SUCCESS(Result))
        {
            free(pBuffer);
            return 0;
        }
    } while (Result == STATUS_INFO_LENGTH_MISMATCH);

    // 取模块信息结构
    pSystemModuleInformation = (PSYSTEM_MODULE_INFORMATION)pBuffer;

    // 得到模块地址
    ULONG64 ret = (ULONG64)(pSystemModuleInformation->Module[0].Base);

    // 拷贝模块名
    if (NtosName != NULL)
    {
        strcpy(NtosName, pSystemModuleInformation->Module[0].ImageName +
        pSystemModuleInformation->Module[0].ModuleNameOffset);
    }

    free(pBuffer);
    return ret;
}
```

```
// 判断并修复重定位表
BOOL RepairRelocationTable(ULONG64 HandleInFile, ULONG64 BaseInKernel)
{
    PIMAGE_DOS_HEADER      pDosHeader;
    PIMAGE_NT_HEADERS64     pNtHeader;
    PIMAGE_BASE_RELOCATION   pRelocTable;
    ULONG i, dwOldProtect;

    // 得到DOS头并判断是否符合DOS规范
    pDosHeader = (PIMAGE_DOS_HEADER)HandleInFile;
    if (pDosHeader->e_magic != IMAGE_DOS_SIGNATURE)
    {
        return FALSE;
    }

    // 得到Nt头
    pNtHeader = (PIMAGE_NT_HEADERS64)((ULONG64)HandleInFile + pDosHeader->e_lfanew);

    // 是否存在重定位表
    if (pNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].Size)
    {
        // 获取到重定位表地址
        pRelocTable = (PIMAGE_BASE_RELOCATION)((ULONG64)HandleInFile + pNtHeader-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress);

        do
        {
            // 得到重定位号
            ULONG numofReloc = (pRelocTable->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION))
/ 2;
            SHORT minioffset = 0;

            // 得到重定位数据
            PUSHORT pRelocData = (PUSHORT)((ULONG64)pRelocTable +
sizeof(IMAGE_BASE_RELOCATION));

            // 循环或直接判断*pRelocData是否为0也可以作为结束标记
            for (i = 0; i < numofReloc; i++)
            {
                // 需要重定位的地址
                PULONG64 RelocAddress;

                // 重定位的高4位是重定位类型，判断重定位类型
                if (((*pRelocData) >> 12) == IMAGE_REL_BASED_DIR64)
                {
                    // 计算需要进行重定位的地址
                    // 重定位数据的低12位再加上本重定位块头的RVA即真正需要重定位的数据的RVA
                    minioffset = (*pRelocData) & 0FFF; // 小偏移

                    // 模块基址+重定位基址+每个数据表示的小偏移量
                    RelocAddress = (PULONG64)(HandleInFile + pRelocTable->VirtualAddress +
minioffset);
                }
            }
        }
    }
}
```

```

        // 直接在RING3修改：原始数据+基址-IMAGE_OPTIONAL_HEADER中的基址
        VirtualProtect((PVOID)RelocAddress, 4, PAGE_EXECUTE_READWRITE,
&dwOldProtect);

        // 因为是R3直接LOAD的所以要修改一下内存权限
        *RelocAddress = *RelocAddress + BaseInKernel - pNtHeader-
>OptionalHeader.ImageBase;
        VirtualProtect((PVOID)RelocAddress, 4, dwOldProtect, NULL);
    }

    // 下一个重定位数据
    pRelocData++;

}

// 下一个重定位块
pRelocTable = (PIMAGE_BASE_RELOCATION)((ULONG64)pRelocTable + pRelocTable-
>SizeofBlock);
} while (pRelocTable->VirtualAddress);

return TRUE;
}

return FALSE;
}

// 初始化
BOOL InitEngine(BOOL IsClear)
{
    if (IsClear == TRUE)
    {
        // 动态获取ntdll.dll模块的基地址
        KernelBase = GetKernelBase64(NtosFullName);
        printf("模块基址: %llx | 模块名: %s \n", KernelBase, NtosFullName);
        if (!KernelBase)
        {
            return FALSE;
        }

        // 动态加载模块到内存，并获取到模块句柄
        hKernel = LoadLibraryExA(NtosFullName, 0, DONT_RESOLVE_DLL_REFERENCES);

        if (!hKernel)
        {
            return FALSE;
        }

        // 判断并修复重定位表
        if (!RepairRelocationTable((ULONG64)hKernel, KernelBase))
        {
            return FALSE;
        }
        return TRUE;
    }
    else
    {

```

```
        FreeLibrary(hKernel);
        return TRUE;
    }
}

// 获取原始函数机器码
VOID GetOriginalMachineCode(ULONG64 Address, PUCHAR ba, SIZE_T Length)
{
    ULONG64 offsetAddress = Address - KernelBase + (ULONG64)hKernel;
    RtlCopyMemory(ba, (PVOID)offsetAddress, Length);
}

// 获取传入函数的内存地址
ULONG64 GetSystemRoutineAddress(PCHAR FuncName)
{
    return KernelBase + (ULONG64)GetProcAddress(hKernel, FuncName) - (ULONG64)hKernel;
}

// 获取当前函数机器码
VOID GetCurrentMachineCode(ULONG64 Address, PUCHAR ba, SIZE_T Length)
{
    ULONG64 dat[2] = { 0 };
    dat[0] = Address;
    dat[1] = Length;
    IoControl(hDriver, CTL_CODE_GEN(0x800), dat, 16, ba, Length);
}

// 清除特定位置的机器码
VOID ClearInlineHook(ULONG64 Address, PUCHAR ba, SIZE_T Length)
{
    KF_DATA dat = { 0 };
    dat.Address = (PVOID)Address;
    dat.Length = Length;

    // 直接调用写出控制码
    RtlCopyMemory(dat.data, ba, Length);
    IoControl(hDriver, CTL_CODE_GEN(0x801), &dat, sizeof(KF_DATA), 0, 0);
}

// 打印数据
VOID PrintBytes(PCHAR DescriptionString, PUCHAR ba, UINT Length)
{
    printf("%s", DescriptionString);
    for (UINT i = 0; i < Length; i++)
    {
        printf("%02x ", ba[i]);
    }
    printf("\n");
}

int main(int argc, char *argv[])
{
    UCHAR OriginalMachineCode[BYTE_ARRAY_LENGTH];
```

```
UCHAR CurrentMachineCode[BYTE_ARRAY_LENGTH];
ULONG64 Address = 0;

hDriver = CreateFileA("\\\\.\\"WinDDK\"", GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

// 初始化
if (!InitEngine(TRUE) || hDriver == 0)
{
    return 0;
}

// 需要获取的函数列表
CHAR *FunctionList[128] = { "PsLookupProcessByProcessId", "NtCommitEnlistment",
"NtCommitComplete", "NtCommitTransaction" };

for (size_t i = 0; i < 4; i++)
{
    // 清空缓存
    RtlZeroMemory(OriginalMachineCode, 0, BYTE_ARRAY_LENGTH);
    RtlZeroMemory(CurrentMachineCode, 0, BYTE_ARRAY_LENGTH);

    // 获取到当前函数地址
    Address = GetSystemRoutineAddress(FunctionList[i]);

    printf("\n函数地址: %p | 函数名: %s\n", Address, FunctionList[i]);
    if (Address == 0 || Address < KernelBase)
    {
        return 0;
    }

    GetOriginalMachineCode(Address, OriginalMachineCode, BYTE_ARRAY_LENGTH);
    PrintBytes("原始机器码: ", OriginalMachineCode, BYTE_ARRAY_LENGTH);

    GetCurrentMachineCode(Address, CurrentMachineCode, BYTE_ARRAY_LENGTH);
    PrintBytes("当前机器码: ", CurrentMachineCode, BYTE_ARRAY_LENGTH);

    /*
    // 不相同则询问是否恢复
    if (memcmp(OriginalMachineCode, CurrentMachineCode, BYTE_ARRAY_LENGTH))
    {
        printf("按下[ENTER]恢复钩子");
        getchar();
        ClearInlineHook(Address, OriginalMachineCode, BYTE_ARRAY_LENGTH);
    }
    */
}

// 注销
InitEngine(FALSE);
system("pause");

return 0;
```

}

首先编译驱动程序 WinDDK.sys 并通过 KmdManager 将驱动程序拉起来，运行客户端 lyshark.exe 程序会输出当前 FunctionList 列表中，指定的4个函数的挂钩情况。

进程	驱动模块	内核	内核钩子	应用层钩子	网络	注册表	文件	启动信息	监控	系统杂项	关于		
SSDT	ShadowSSDT	FSD	键盘	I8042prt	鼠标	Partmgr	Disk	Atapi	Acpi	Scsi	内核钩子	Object钩子	系统中断表
序号	函数名称			当前函数地址		Hook		原始函数地址		当前函数地址所在模块			
148	NtCommitComplete			0xFFFFF8051B19C730		Inline Hook		0xFFFFF8051B19C730		C:\Windows\system32\ntoskrnl.exe			
149	NtCommitEnlistment			0xFFFFF8051B19C750		Inline Hook		0xFFFFF8051B19C750		C:\Windows\system32\ntoskrnl.exe			
151	NtCommitTransaction			0xFFFFF8051B19C770		Inline Hook		0xFFFFF8051B19C770		C:\Windows\system32\ntoskrnl.exe			
165	NtCreateEnlistment			0xFFFFF8051B19C790		Inline Hook		0xFFFFF8051B19C790		C:\Windows\system32\ntoskrnl.exe			

模块基址: fffff8051b000000 | 模块名: ntoskrnl.exe

函数地址: FFFF8051B61CFE0 | 函数名: PsLookupProcessByProcessId
原始机器码: 48 89 5c 24 18 56 48 83 ec 20 48 89 7c 24 38 48
当前机器码: 48 89 5c 24 18 56 48 83 ec 20 48 89 7c 24 38 48

函数地址: FFFF8051B19C750 | 函数名: NtCommitEnlistment
原始机器码: 48 ff 25 a1 3d 3d 00 cc cc cc cc cc cc cc cc cc
当前机器码: 4c 8b 15 a1 3d 3d 00 e9 74 a4 15 01 cc cc cc cc

函数地址: FFFF8051B19C730 | 函数名: NtCommitComplete
原始机器码: 48 ff 25 c9 3d 3d 00 cc cc cc cc cc cc cc cc cc
当前机器码: 4c 8b 15 c9 3d 3d 00 e9 b4 a3 15 01 cc cc cc cc

函数地址: FFFF8051B19C770 | 函数名: NtCommitTransaction
原始机器码: 48 ff 25 79 3d 3d 00 cc cc cc cc cc cc cc cc cc
当前机器码: 4c 8b 15 79 3d 3d 00 e9 34 f3 15 01 cc cc cc cc

请按任意键继续. . .

