

内核 INLINE HOOK 是各种 RK、AV 和 ARK 最常用的手段之一了，如果让我在 WIN64 上使用 HOOK 的话，我会首推 INLINE HOOK，毕竟 SSDT HOOK 和 SHADOW SSDT 非常麻烦，不好修改。目前来说支持 WIN64 的 INLINE HOOK 引擎有很多，比如 MHOOK 和 MINI HOOK ENGINE，不过要移植到内核里还是很麻烦的，所以我特地做了一个适用于 WIN64 的 RING0 INLINE HOOK ENGINE。当然，有矛必有盾，有了 HOOK ENGINE，必然还有一个用于 UNHOOK 的玩意儿。

一、实现 RING0 INLINE HOOK

无论是用户态 Inline Hook 还是内核级 Inline Hook，都要遵循一个原则，就是指令不能截断，否则会出大错误。所以，反汇编引擎在 Inline Hook 引擎中的作用，就是判断指令的长度。首先介绍我找到的 x64 反汇编引擎，LDE64。LDE64 是 Length Disassemble Engine for x64 的缩写，此反汇编引擎小巧玲珑，最大的优点就是能带进驱动里使用（很多外国开源的反汇编引擎都无法带进驱动，还需要很多莫名其妙的非标准 C++ 库）。不过，LDE64 的作者也够小气的，没有直接给出源代码，但是给了一个十几 KB 的 shellcode，当你需要反汇编时，直接调用 shellcode 就行了。核心代码如下：

```
unsigned char szShellCode[12800] = {...} //详细机器码请见源码文件
typedef int (*LDE_DISASM)(void *p, int dw);
LDE_DISASM LDE;
void LDE_init()
{
    LDE=ExAllocatePool(NonPagedPool, 12800);
    memcpy(LDE, szShellCode, 12800);
}
```

需要使用时先调用 LDE_init 进行初始化，然后再调用 LDE 函数即可。LDE 函数要求输入地址和平台类型，返回一条指令的字节长度。接下来编写一个自定义函数，返回要 Patch 的字节数目。虽然在 Win64 上写一个跨 4G 跳转指令理论上只需要 14 字节，但是 14 字节并不一定就是 N 个完整的指令，所以必须得到 N 个完整指令的长度（N 条指令的长度要大于等于 14）：

```
ULONG GetPatchSize(PUCHAR Address)
{
    ULONG LenCount=0, Len=0;
    while(LenCount<=14)    //至少需要 14 字节
    {
        Len=LDE(Address, 64);
        Address=Address+Len;
        LenCount=LenCount+Len;
    }
    return LenCount;
}
```

接下来，说一下实现内核级 Inline Hook 的思路：

1. 获得待 HOOK 函数的地址（Address）

2. 获得要修改的字节数目（N）
3. 保存这头 N 字节的机器码
4. 创建『原函数』（把复制头 N 字节，再跳转到 Address+N 的地方）
5. 修改函数头，跳转到代理函数里

解释一下跳转的代码。我之前使用的跳转流程是：

```
MOV RAX, 绝对地址
JMP RAX
```

后来感觉修改 RAX 不太好（虽然 RAX 是易失性寄存器），于是换了方式：

```
JMP QWORD PTR [本条指令结束后的地址]
```

以上指令的机器码是：FF 25 00 00 00 00。代码如下：

```
//传入：待 HOOK 函数地址，代理函数地址，接收原始函数地址的指针，接收补丁长度的指针；返回：原来
//头 N 字节的数据
PVOID HookKernelApi(IN PVOID ApiAddress, IN PVOID Proxy_ApiAddress, OUT PVOID
*Original_ApiAddress, OUT ULONG *PatchSize)
{
    KIRQL irql;
    UINT64 tmpv;
    PVOID head_n_byte, ori_func;
    UCHAR jmp_code[] = "\xFF\x25\x00\x00\x00\x00\xFF\xFF\xFF\xFF\xFF\xFF\xFF";
    UCHAR jmp_code_orifunc[] = "\xFF\x25\x00\x00\x00\x00\xFF\xFF\xFF\xFF\xFF\xFF\xFF";
    //How many bytes shoule be patch
    *PatchSize = GetPatchSize((PUCHAR)ApiAddress);
    //step 1: Read current data
    head_n_byte = kmalloc(*PatchSize);
    irql = WPOFFx64();
    memcpy(head_n_byte, ApiAddress, *PatchSize);
    WPONx64(irql);
    //step 2: Create ori function
    ori_func = kmalloc(*PatchSize + 14); //原始机器码+跳转机器码
    RtlFillMemory(ori_func, *PatchSize + 14, 0x90);
    tmpv = (ULONG64)ApiAddress + *PatchSize; //跳转到没被打补丁的那个字节
    memcpy(jmp_code_orifunc + 6, &tmpv, 8);
    memcpy((PUCHAR)ori_func, head_n_byte, *PatchSize);
    memcpy((PUCHAR)ori_func + *PatchSize, jmp_code_orifunc, 14);
    *Original_ApiAddress = ori_func;
    //step 3: fill jmp code
    tmpv = (UINT64)Proxy_ApiAddress;
    memcpy(jmp_code + 6, &tmpv, 8);
    //step 4: Fill NOP and hook
    irql = WPOFFx64();
    RtlFillMemory(ApiAddress, *PatchSize, 0x90);
```

```
memcpy(ApiAddress, jmp_code, 14);
WPONx64(irq1);
//return ori code
return head_n_byte;
}
```

反挂钩就简单了，直接把头 N 字节回复即可：

```
//传入：被 HOOK 函数地址，原始数据，补丁长度
VOID UnhookKernelApi(IN PVOID ApiAddress, IN PVOID OriCode, IN ULONG PatchSize)
{
    KIRQL irq1;
    irq1=WPONx64();
    memcpy(ApiAddress, OriCode, PatchSize);
    WPONx64(irq1);
}
```

接下来是示例，很简单地调用一下以上两个函数即可！

```
NTSTATUS Proxy_PsLookupProcessByProcessId(HANDLE ProcessId, PEPROCESS *Process)
{
    NTSTATUS st;
    st=((PSLOOKUPPROCESSBYPROCESSID)ori_pslp)(ProcessId,Process);
    if(NT_SUCCESS(st))
    {
        if(*Process==(PEPROCESS)my_eprocess)
        {
            *Process=0;
            st=STATUS_ACCESS_DENIED;
        }
    }
    return st;
}

VOID HookPsLookupProcessByProcessId()
{
    pslp_head_n_byte = HookKernelApi(GetFunctionAddr(L"PsLookupProcessByProcessId"),
                                     (PVOID)Proxy_PsLookupProcessByProcessId,
                                     &ori_pslp,
                                     &pslp_patch_size);
}

VOID UnhookPsLookupProcessByProcessId()
{
    UnhookKernelApi(GetFunctionAddr(L"PsLookupProcessByProcessId"),
                   pslp_head_n_byte,
```

```

        pslp_patch_size);
}

```

效果如下：



用 WINDBG 检测一下：

挂钩前：

```

lkd> u pslookupprocessbyprocessid
nt!PsLookupProcessByProcessId:
fffff800`0194c750 48895c2408      mov     qword ptr [rsp+8],rbx
fffff800`0194c755 48896c2410      mov     qword ptr [rsp+10h],rbp
fffff800`0194c75a 4889742418      mov     qword ptr [rsp+18h],rsi
fffff800`0194c75f 57             push    rdi
fffff800`0194c760 4154          push    r12
fffff800`0194c762 4155          push    r13
fffff800`0194c764 4883ec20      sub     rsp,20h
fffff800`0194c768 65488b3c2588010000 mov     rdi,qword ptr gs:[188h]

```

挂钩后：

```

lkd> u pslookupprocessbyprocessid
nt!PsLookupProcessByProcessId:
fffff800`0194c750 ff2500000000      jmp     qword ptr [nt!PsLookupProcessByProcessId+0x6
(fffff800`0194c756)]
fffff800`0194c756 dc50e9          fcom    qword ptr [rax-17h]

```

```

fffff800`0194c759 06          ???
fffff800`0194c75a 80f8ff      cmp     al, 0FFh
fffff800`0194c75d ff9057415441 call    qword ptr [rax+41544157h]
fffff800`0194c763 55          push    rbp
fffff800`0194c764 4883ec20     sub     rsp, 20h
fffff800`0194c768 65488b3c2588010000 mov     rdi, qword ptr gs:[188h]

```

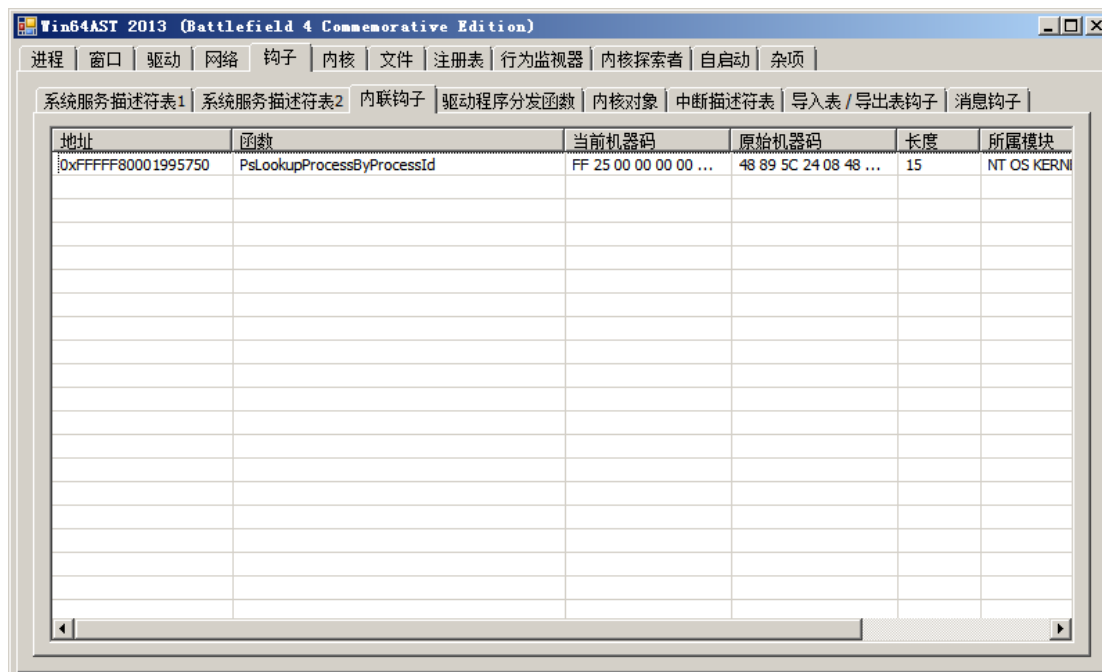
看样子代码好像乱了，其实不是的。因为跨 4G 跳转指令是 14 字节，而我们修改了 PsLookupProcessByProcessId 的头 15 字节（正好三条指令），前 6 字节是指令，后 9 字节并不是指令，而是数据（前 8 字节是绝对地址）和填充码（最后 1 字节没有意义）。所以这么看就对了：

```

lkd> u PsLookupProcessByProcessId+0xf
nt!PsLookupProcessByProcessId+0xf:
fffff800`0194c75f 57          push    rdi
fffff800`0194c760 4154        push    r12
fffff800`0194c762 4155        push    r13
fffff800`0194c764 4883ec20     sub     rsp, 20h
fffff800`0194c768 65488b3c2588010000 mov     rdi, qword ptr gs:[188h]
fffff800`0194c771 4533e4      xor     r12d, r12d
fffff800`0194c774 488bea      mov     rbp, rdx
fffff800`0194c777 66ff8fc4010000 dec     word ptr [rdi+1C4h]

```

用 WIN64AST 检测：



二、恢复 RING0 INLINE HOOK

首先说说清除 INLINE HOOK 的一般步骤（无论是 X86 还是 X64 平台，无论是用户态还是内核级的 INLINE HOOK，都适用）：

1. 找到被挂钩函数的地址
2. 获得原始的机器码
3. 把原始机器码写入到指定的地址

以下表格说明了实现上述 3 大步骤的 N 个小步骤：

<ol style="list-style-type: none">1. 找到被挂钩函数的地址<ol style="list-style-type: none">1.1 获得被 HOOK 函数所属模块在内存里的基址 KernelBase1.2 在进程里用 LoadLibraryEx 加载一份被 HOOK 函数的所属模块，获得映射地址 hKernel1.3 获得函数的实际地址：Address=KernelBase-hKernel+ GetProcAddress(hKernel, 函数名称)
<ol style="list-style-type: none">2. 获得原始的机器码<ol style="list-style-type: none">2.1 根据 KernelBase 和 hKernel 进行重定位2.2 根据计算公式 OffsetAddress=Address-KernelBase+hKernel 获得偏移地址2.3 使用 memcpy 复制指定长度的数据，这个数据就是原始机器码了
<ol style="list-style-type: none">3. 把原始机器码写入到指定的地址<ol style="list-style-type: none">3.1 把原始机器码传到驱动里3.2 把 IRQL 提升到和 DPC 同样的级别3.3 通过修改 CR0 寄存器的值来关闭内存写保护3.4 使用 RtlCopyMemory 写内存3.5 通过修改 CR0 寄存器的值来打开内存写保护3.6 把 IRQL 降低到 PASSIVE_LEVEL

1. 获得内核文件的加载基址：

```
ULONG64 GetKernelBase64(char *NtosName)
{
    typedef long (__stdcall *ZWQUERYSYSTEMINFORMATION)
    (
        IN ULONG SystemInformationClass,
        IN PVOID SystemInformation,
        IN ULONG SystemInformationLength,
        IN PULONG ReturnLength OPTIONAL
    );
    typedef struct _SYSTEM_MODULE_INFORMATION_ENTRY
    {
        ULONG Unknown1;
        ULONG Unknown2;
        ULONG Unknown3;
        ULONG Unknown4;
        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 _SYSTEM_MODULE_INFORMATION
    {
        ULONG Count;
        SYSTEM_MODULE_INFORMATION_ENTRY Module[1];
    } SYSTEM_MODULE_INFORMATION, *PSYSTEM_MODULE_INFORMATION;
    #define SystemModuleInformation 11
    #define STATUS_INFO_LENGTH_MISMATCH ((NTSTATUS)0xC0000004L)
    ZWQUERYSYSTEMINFORMATION ZwQuerySystemInformation;
    PSYSTEM_MODULE_INFORMATION pSystemModuleInformation;
    ULONG NeedSize, BufferSize = 0x5000;
    PVOID pBuffer = NULL;
    NTSTATUS Result;
    ZwQuerySystemInformation=(ZWQUERYSYSTEMINFORMATION)GetProcAddress(GetModuleHandleA("ntdll.dll"), "ZwQuerySystemInformation");
    do
    {
        pBuffer = malloc( BufferSize );
        if( pBuffer == NULL ) return 0;
        Result = ZwQuerySystemInformation( SystemModuleInformation, pBuffer, BufferSize,
        &NeedSize );
        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 );
    pSystemModuleInformation = (PSYSTEM_MODULE_INFORMATION)pBuffer;
    ULONG64 ret=(ULONG64) (pSystemModuleInformation->Module[0].Base);
    if (NtosName!=NULL)

        strcpy(NtosName, pSystemModuleInformation->Module[0].ImageName+pSystemModuleInformation->Module[0].ModuleNameOffset);
        free(pBuffer);
        return ret;
}

```

2. 内核文件重定位：

```

int Reloc(ULONG64 HandleInFile, ULONG64 BaseInKernel)
{
    PIMAGE_DOS_HEADER      pDosHeader;
    PIMAGE_NT_HEADERS64     pNtHeader;
    PIMAGE_BASE_RELOCATION  pRelocTable;
    ULONG i, dwOldProtect;
    pDosHeader = (PIMAGE_DOS_HEADER)HandleInFile;
    if ( pDosHeader->e_magic != IMAGE_DOS_SIGNATURE )
        return 0;
    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)
    {
        pRelocTable=(PIMAGE_BASE_RELOCATION) ((ULONG64)HandleInFile +
        pNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress);
        do{
            ULONG      numofReloc=(pRelocTable->SizeOfBlock-
sizeof(IMAGE_BASE_RELOCATION))/2;
            SHORT      minioffset=0;
            PUSHORT
pRelocData=(PUSHORT) ((ULONG64)pRelocTable+sizeof(IMAGE_BASE_RELOCATION));
            //循环，或直接判断*pData 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
                if (((*pRelocData)>>12)== IMAGE_REL_BASED_DIR64)//判断重定位类型
                {
                    //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)&0xFFF;//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;
}

```

3. 把内核文件加载到进程、获得函数地址和原始机器码：

```

int InitGetOriCode(int bClear)
{
    if( bClear==0 )
    {
        KernelBase=GetKernelBase64(NtosFullName);
        //printf("KB: %llx\nKN: %s\n", KernelBase, NtosFullName);
        if(!KernelBase) return 0;
        hKernel=LoadLibraryExA(NtosFullName, 0, DONT_RESOLVE_DLL_REFERENCES);
        //printf("KH: %llx\n", hKernel);
        if(!hKernel) return 0;
        if(!Reloc((ULONG64)hKernel, KernelBase)) return 0;
        return 1;
    }
    else
    {
        FreeLibrary(hKernel);
        return 1;
    }
}

```

```

void GetOriCode(ULONG64 Address, PCHAR ba, SIZE_T Length)
{
    ULONG64 OffsetAddress=Address-KernelBase+(ULONG64)hKernel;
    memcpy(ba, (PVOID)OffsetAddress, Length);
}

```

```

ULONG64 GetSystemRoutineAddress(char *FuncName)
{
    return KernelBase+(ULONG64)GetProcAddress(hKernel, FuncName)-(ULONG64)hKernel;
}

```

4. 获得当前在内存里的机器码和清除 INLINE HOOK（这两步本质上就是调用 RtlCopyMemory 而已，只是第一个参数和第二个参数互换一下位置）：

```

//RING3 源码：

```

```
void GetCurCode(ULONG64 Address, PCHAR ba, SIZE_T Length)
{
    ULONG64 dat[2]={0};
    dat[0]=Address;
    dat[1]=Length;
    IoControl(hDriver ,CTL_CODE_GEN(0x800), dat, 16, ba, Length);
}

void ClearInlineHook(ULONG64 Address, PCHAR ba, SIZE_T Length)
{
    typedef struct _KF_DATA
    {
        PVOID Address;
        ULONG64 Length;
        UCHAR data[256];
    } KF_DATA, *PKF_DATA;
    KF_DATA dat={0};
    dat.Address=(PVOID)Address;
    dat.Length=Length;
    memcpy(dat.data, ba, Length);
    IoControl(hDriver ,CTL_CODE_GEN(0x801), &dat, sizeof(KF_DATA), 0, 0);
}
```

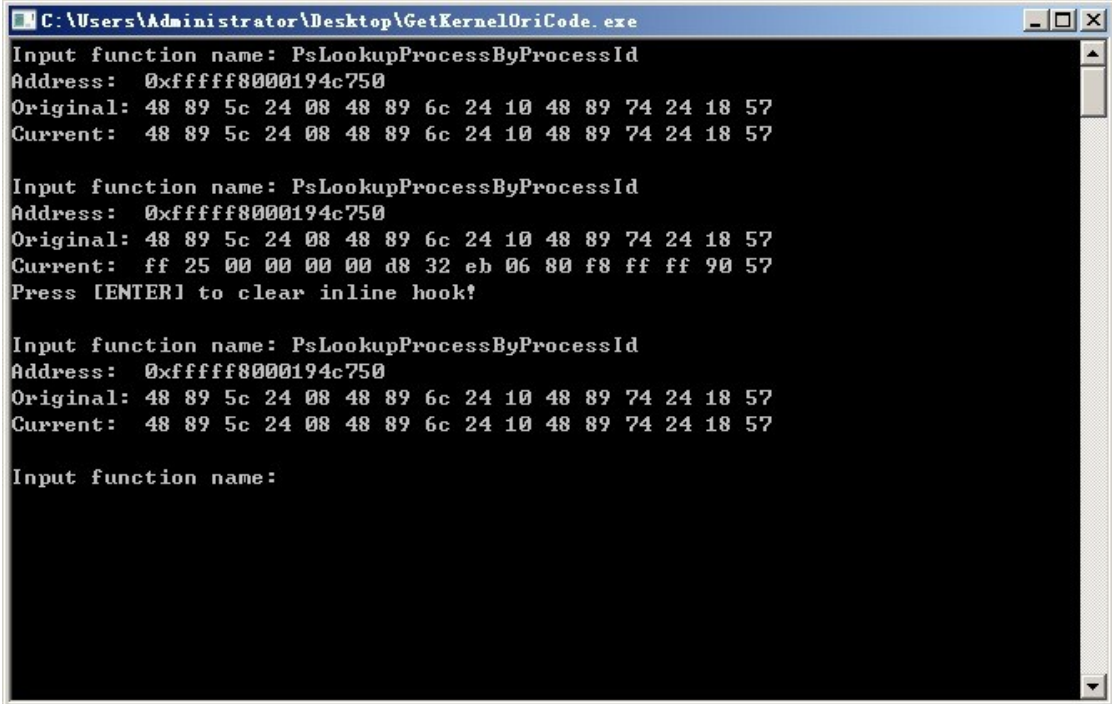
//RINGO 源码:

```
case IOCTL_GET_CUR_CODE:
{
    KF_DATA dat={0};
    memcpy(&dat, pIoBuffer, 16); //只用前两个成员
    WPOFFx64();
    RtlCopyMemory(pIoBuffer, dat.Address, dat.Length);
    WPONx64();
    status = STATUS_SUCCESS;
    break;
}

case IOCTL_SET_ORI_CODE:
{
    KF_DATA dat={0};
    memcpy(&dat, pIoBuffer, sizeof(KF_DATA));
    WPOFFx64();
    RtlCopyMemory(dat.Address, dat.data, dat.Length);
    WPONx64();
    status = STATUS_SUCCESS;
    break;
}
```

其它界面处理的源码就不贴出来了，下面主要看一下实验效果（使用 HOOK

PsLookupProcessByProcessId 为例子)：



```
C:\Users\Administrator\Desktop\GetKernelOriCode.exe
Input function name: PsLookupProcessByProcessId
Address: 0xffffffff8000194c750
Original: 48 89 5c 24 08 48 89 6c 24 10 48 89 74 24 18 57
Current: 48 89 5c 24 08 48 89 6c 24 10 48 89 74 24 18 57

Input function name: PsLookupProcessByProcessId
Address: 0xffffffff8000194c750
Original: 48 89 5c 24 08 48 89 6c 24 10 48 89 74 24 18 57
Current: ff 25 00 00 00 00 d8 32 eb 06 80 f8 ff ff 90 57
Press [ENTER] to clear inline hook!

Input function name: PsLookupProcessByProcessId
Address: 0xffffffff8000194c750
Original: 48 89 5c 24 08 48 89 6c 24 10 48 89 74 24 18 57
Current: 48 89 5c 24 08 48 89 6c 24 10 48 89 74 24 18 57

Input function name:
```

第一堆数据是在挂钩 PsLookupProcessByProcessId 之前的；第二堆数据是在挂钩 PsLookupProcessByProcessId 之后的；第三堆数据是在清除 PsLookupProcessByProcessId 的钩子之后的（和第一堆数据相同）。

课后作业：挂钩 KeInsertQueueApc 实现进程防杀。