一、RING3 INLINE HOOK

在 WIN64 系统上进行 RING0 INLINE HOOK 比较麻烦（要破 PatchGuard），但进行 RING3 INLINE HOOK 还是比较轻松的。在 Win64 下进行 Ring 3 Inline Hook 的基本流程跟 Win32 下差不多，首先要编写一个能挂钩进程自身内部函数的 DLL，再把这个 DLL 注射进别的进程体内。一个标准的 DLL 注入程序如下：

```
BOOL WINAPI InjectProxyW(DWORD dwPID, PCWSTR pwszProxyFile)
{
    BOOL ret = FALSE;
    HANDLE hToken = NULL;
    HANDLE hProcess = NULL;
    HANDLE hThread = NULL;
    FARPROC pfnThreadRtn = NULL;
    PWSTR pwszPara = NULL;
    hProcess = OpenProcess(PROCESS_ALL_ACCESS,FALSE, dwPID);
    pfnThreadRtn = GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");
    size_t iProxyFileLen = wcslen(pwszProxyFile)*sizeof(WCHAR);
    pwszPara = (PWSTR)VirtualAllocEx(hProcess, NULL, iProxyFileLen, MEM_COMMIT,
PAGE_READWRITE);
    WriteProcessMemory(hProcess, pwszPara, (PVOID)pwszProxyFile, iProxyFileLen, NULL);
    hThread = CreateRemoteThread(hProcess, NULL, 1024, (LPTHREAD_START_ROUTINE)pfnThreadRtn,
pwszPara, 0, NULL);
    WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hThread);
    VirtualFreeEx(hProcess, pwszPara, 0, MEM_RELEASE);
    CloseHandle(hProcess);
    return(TRUE);
}
```

以上代码必须编译成 64 位 EXE 才能。接下来要编写用来 Hook API 的 DLL 了。由于这是 Win64，跟 Win32 有了天壤之别，挂钩绝对不再是“原函数头 5 字节改 JMP 跳到伪函数，伪函数处理完毕后再跳到原函数地址+5 的地方”这么简单了。现在我们遇到两个难题：1.JMP 的最大跳转范围是 4GB，然而原函数所在的 DLL 通常不和伪函数所在的 DLL 在同一个 4GB 中；那么，E9 XX XX XX XX 也要改成 FF 15 XX XX XX XX；2.Win64 上函数的前 N 字节通常都是和堆栈有关的指令，不是 mov edi,edi 等无意义的指令，如果乱改，堆栈会被搞乱的。堆栈一乱，进程就挂了。

不过，这个麻烦的问题外国朋友早解决了。国外比较成熟的 RING3 INLINE HOOK 库有 mHOOK 和 MiniHookEngine，个人比较倾向于用 MiniHookEngine。MiniHookEngine 的作者是 Daniel Pistelli，MiniHookEngine 同时支持 WIN32 和 WIN64。MiniHookEngine 的核心代码如下：

```
// 10000 hooks should be enough
#define MAX_HOOKS 10000
```

```
typedefstruct_HOOK_INFO
{
    ULONG_PTR Function;    // Address of the original function
    ULONG_PTR Hook;        // Address of the function to call
                           // instead of the original
    ULONG_PTR Bridge;   // Address of the instruction bridge
                           // necessary because of the hook jmp
                           // which overwrites instructions
} HOOK_INFO, *PHOOK_INFO;
HOOK_INFO HookInfo[MAX_HOOKS];
UINTNumberOfHooks = 0;
BYTE*pBridgeBuffer = NULL; // Here are going to be stored all the bridges
UINTCurrentBridgeBufferSize = 0; // This number is incremented as
                                 // the bridge buffer is growing
HOOK_INFO *GetHookInfoFromFunction(ULONG_PTR OriginalFunction)
{
    if (NumberOfHooks == 0)
        return NULL;
    for (UINT x = 0; x < NumberOfHooks; x++)
    {
        if (HookInfo[x].Function == OriginalFunction)
            return &HookInfo[x];
    }
    return NULL;
}
//
// This function  retrieves the necessary size for the jump
//
UINTGetJumpSize(ULONG_PTR PosA, ULONG_PTR PosB)
{
    ULONG_PTR res = max(PosA, PosB) - min(PosA, PosB);
    // if you want to handle relative jumps
    /*if (res <= (ULONG_PTR) 0x7FFF0000)
     {
         return 5; // jmp rel
     }
     else
     {*/
        // jmp [xxx] + addr
#ifdef _M_IX86
        return 10;
#else ifdef _M_AMD64
        return 14;
#endif
```

```
    //}
    return 0; // error
}
//
// This function writes unconditional jumps
// both for x86 and x64
//
VOIDWriteJump(VOID *pAddress, ULONG_PTR JumpTo)
{
    DWORD dwOldProtect = 0;
    VirtualProtect(pAddress, JUMP_WORST, PAGE_EXECUTE_READWRITE, &dwOldProtect);
    BYTE *pCur = (BYTE *) pAddress;
    // if you want to handle relative jumps
    /*if (JumpTo - (ULONG_PTR) pAddress <= (ULONG_PTR) 0x7FFF0000)
     {
         *pCur = 0xE9;  // jmp rel
         DWORD RelAddr = (DWORD) (JumpTo - (ULONG_PTR) pAddress) - 5;
         memcpy(++pCur, &RelAddr, sizeof (DWORD));
     }
     else
     {*/
#ifdef _M_IX86
        *pCur = 0xff;     // jmp [addr]
        *(++pCur) = 0x25;
        pCur++;
        *((DWORD *) pCur) = (DWORD)(((ULONG_PTR) pCur) + sizeof (DWORD));
        pCur += sizeof (DWORD);
        *((ULONG_PTR *)pCur) = JumpTo;
#else ifdef _M_AMD64
        *pCur = 0xff;        // jmp [rip+addr]
        *(++pCur) = 0x25;
        *((DWORD *) ++pCur) = 0; // addr = 0
        pCur += sizeof (DWORD);
        *((ULONG_PTR *)pCur) = JumpTo;
#endif
    //}
    DWORD dwBuf = 0;    // nessary othewrise the function fails
    VirtualProtect(pAddress, JUMP_WORST, dwOldProtect, &dwBuf);
}
//
// This function creates a bridge of the original function
//
VOID*CreateBridge(ULONG_PTR Function, const UINT JumpSize)
{
```

```
    if (pBridgeBuffer == NULL) return NULL;
#define MAX_INSTRUCTIONS 100
    _DecodeResult res;
    _DecodedInst decodedInstructions[MAX_INSTRUCTIONS];
    unsigned int decodedInstructionsCount = 0;
#ifdef _M_IX86
    _DecodeType dt = Decode32Bits;
#else ifdef _M_AMD64
    _DecodeType dt = Decode64Bits;
#endif
    _OffsetType offset = 0;
    res = distorm_decode(offset,     // offset for buffer
        (const BYTE *) Function,     // buffer to disassemble
        50,                             // function size (code size to disasm)
                                     // 50 instr should be _quite_ enough
        dt,                             // x86 or x64?
        decodedInstructions,         // decoded instr
        MAX_INSTRUCTIONS,             // array size
        &decodedInstructionsCount    // how many instr were disassembled?
        );
    if (res == DECRES_INPUTERR)
        return NULL;
    DWORD InstrSize = 0;
    VOID *pBridge = (VOID *) &pBridgeBuffer[CurrentBridgeBufferSize];
    for (UINT x = 0; x < decodedInstructionsCount; x++)
    {
        if (InstrSize >= JumpSize)
            break;
        BYTE *pCurInstr = (BYTE *) (InstrSize + (ULONG_PTR) Function);
        //
        // This is an sample attempt of handling a jump
        // It works, but it converts the jz to jmp
        // since I didn't write the code for writing
        // conditional jumps
        //
        /*
         if (*pCurInstr == 0x74) // jz near
         {
             ULONG_PTR Dest = (InstrSize + (ULONG_PTR) Function)
                 + (char) pCurInstr[1];

             WriteJump(&pBridgeBuffer[CurrentBridgeBufferSize], Dest);

             CurrentBridgeBufferSize += JumpSize;
```

```
        }
        else
        {*/
            memcpy(&pBridgeBuffer[CurrentBridgeBufferSize],
                (VOID *) pCurInstr, decodedInstructions[x].size);
            CurrentBridgeBufferSize += decodedInstructions[x].size;
        //}
        InstrSize += decodedInstructions[x].size;
    }
    WriteJump(&pBridgeBuffer[CurrentBridgeBufferSize], Function + InstrSize);
    CurrentBridgeBufferSize += GetJumpSize((ULONG_PTR)
&pBridgeBuffer[CurrentBridgeBufferSize],
        Function + InstrSize);
    return pBridge;
}
//
// Hooks a function
//
extern"C"__declspec(dllexport)
BOOL__cdecl HookFunction(ULONG_PTR OriginalFunction, ULONG_PTR NewFunction)
{
    //
    // Check if the function has already been hooked
    // If so, no disassembling is necessary since we already
    // have our bridge
    //
    HOOK_INFO *hinfo = GetHookInfoFromFunction(OriginalFunction);
    if (hinfo)
    {
        WriteJump((VOID *) OriginalFunction, NewFunction);
    }
    else
    {
        if (NumberOfHooks == (MAX_HOOKS - 1))
            return FALSE;
        VOID *pBridge = CreateBridge(OriginalFunction, GetJumpSize(OriginalFunction,
NewFunction));
        if (pBridge == NULL)
            return FALSE;
        HookInfo[NumberOfHooks].Function = OriginalFunction;
        HookInfo[NumberOfHooks].Bridge = (ULONG_PTR) pBridge;
        HookInfo[NumberOfHooks].Hook = NewFunction;
        NumberOfHooks++;
        WriteJump((VOID *) OriginalFunction, NewFunction);
```

```
    }
    return TRUE;
}
//
// Unhooks a function
//
extern"C"__declspec(dllexport)
VOID__cdecl UnhookFunction(ULONG_PTR Function)
{
    //
    // Check if the function has already been hooked
    // If not, I can't unhook it
    //
    HOOK_INFO *hinfo = GetHookInfoFromFunction(Function);
    if (hinfo)
    {
        //
        // Replaces the hook jump with a jump to the bridge
        // I'm not completely unhooking since I'm not
        // restoring the original bytes
        //

        WriteJump((VOID *) hinfo->Function, hinfo->Bridge);
    }
}
//
// Get the bridge to call instead of the original function from hook
//
extern"C"__declspec(dllexport)
ULONG_PTR__cdecl GetOriginalFunction(ULONG_PTR Hook)
{
    if (NumberOfHooks == 0)
        return NULL;
    for (UINT x = 0; x < NumberOfHooks; x++)
    {
        if (HookInfo[x].Hook == Hook)
            return HookInfo[x].Bridge;
    }
    return NULL;
}
```

把上述代码编译成 DLL 才能在别的进程中使用。不过我们用来 HOOK API 的 DLL 还要另写一个。在下个 DLL 中，我们要通过 Hook OpenProcess 来保护计算器不被非法关闭，并告诉大家一个在 Win64 上进行全局 HOOK 的方法。

第一步：编写伪函数 Fake_OpenProcess，用来保护计算器进程

```
HANDLE Fake_OpenProcess(DWORD da, BOOL ih, DWORD PId)
{
    HWND myhwnd=0;
    DWORD mypid=0;
    HANDLE (WINAPI *pOpenProcess)(DWORD da, BOOL ih, DWORD PId);
    pOpenProcess=(HANDLE                                                    (WINAPI
*)(DWORD,BOOL,DWORD))GetOriginalFunction( (ULONG_PTR)Fake_OpenProcess );
    myhwnd=pfnFindWindowW(L"CalcFrame",L"Calculator");
    if (myhwnd==0)
        myhwnd=pfnFindWindowW(L"CalcFrame",L"计算器");
    if (myhwnd==0)
    {
        return pOpenProcess(da,ih,PId);
    }
    else
    {
        pfnGetWindowThreadProcessId(myhwnd,&mypid);
        if (PId==mypid)
            return pOpenProcess(da,ih,0); //set pid as 0
        else
            return pOpenProcess(da,ih,PId);
    }
}
```

第二步：编写伪函数 Fake_CreateProcess，来把现在写的这个 DLL 注入进新创建的进程。这里说一句，在 Win64 上 Hook NtResumeThread 是没用的（假设现在写的这个 DLL 放在 C 盘根目录下，名为 HookDll.dll）。

```
BOOLWINAPI Fake_CreateProcessW
(
  __in_opt     LPCWSTR lpApplicationName,
  __inout_opt  LPWSTR lpCommandLine,
  __in_opt     LPSECURITY_ATTRIBUTES lpProcessAttributes,
  __in_opt     LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in         BOOL bInheritHandles,
  __in         DWORD dwCreationFlags,
  __in_opt     LPVOID lpEnvironment,
  __in_opt     LPCWSTR lpCurrentDirectory,
  __in         LPSTARTUPINFO lpStartupInfo,
  __out        LPPROCESS_INFORMATION lpProcessInformation
)
{
    BOOL cpwret;
```

```
    BOOL (WINAPI *pCreateProcessW)(LPCWSTR lpApplicationName,LPWSTR
lpCommandLine,LPSECURITY_ATTRIBUTES lpProcessAttributes,LPSECURITY_ATTRIBUTES
lpThreadAttributes,BOOL bInheritHandles,DWORD dwCreationFlags,LPVOID lpEnvironment,LPCWSTR
lpCurrentDirectory,LPSTARTUPINFO lpStartupInfo,LPPROCESS_INFORMATION lpProcessInformation);
    pCreateProcessW=(BOOL (WINAPI
*)(LPCWSTR,LPWSTR,LPSECURITY_ATTRIBUTES,LPSECURITY_ATTRIBUTES,BOOL,DWORD,LPVOID,LPCWSTR,LPST
ARTUPINFO,LPPROCESS_INFORMATION))GetOriginalFunction( (ULONG_PTR)Fake_CreateProcessW );
cpwret=pCreateProcessW(lpApplicationName,lpCommandLine,lpProcessAttributes,lpThreadAttribute
s,bInheritHandles,dwCreationFlags|CREATE_SUSPENDED,lpEnvironment,lpCurrentDirectory,lpStartu
pInfo,lpProcessInformation);
    InjectProxyW(lpProcessInformation->dwProcessId,L"C:\\HOOKDLL.DLL");
    HANDLE hp=OpenProcess(PROCESS_ALL_ACCESS,0,lpProcessInformation->dwProcessId);
    NtResumeProcess(hp);
    CloseHandle(hp);
    return cpwret;
}
```

第三步：获得 user32.dll 中某些函数的地址并挂钩 API（把 MiniHookEngine 编译的 DLL 放在 C 盘根目录，更名为 NtHookEngine.dll。需要特别注意的是，Kernel32.dll 导出的那个 OpenProcess 函数只是个 stub，真正实现“打开进程”这个功能的 OpenProcess 函数在 KernelBase.dll 里，所以我们这里挂钩的是位于 KernelBase.dll 中的 OpenProcess 函数。而 CreateProcess 无此情况，直接挂钩 Kernel32.dll 导出的 CreateProcess 即可）。

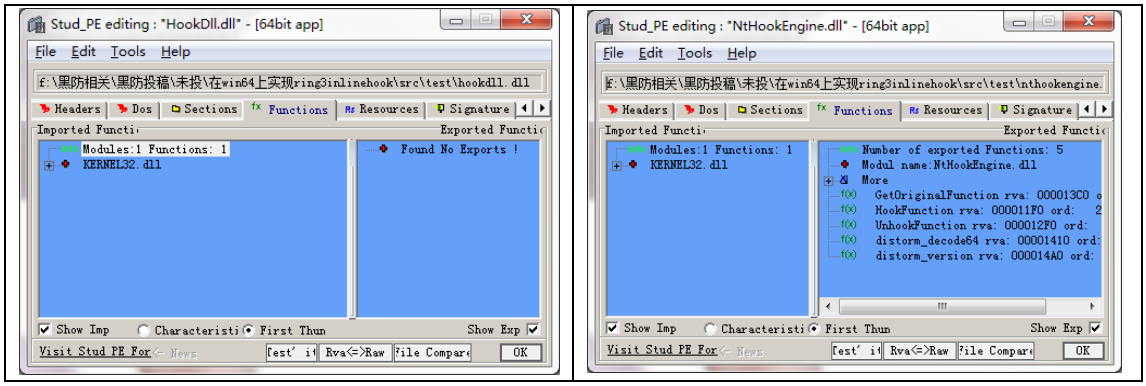
```
Void InitHook()
{
    HMODULE hHookEngineDll = LoadLibraryW(L"C:\\NtHookEngine.dll");
    HookFunction = (BOOL (__cdecl *)(ULONG_PTR, ULONG_PTR))GetProcAddress(hHookEngineDll,
"HookFunction");
    UnhookFunction = (VOID (__cdecl *)(ULONG_PTR))GetProcAddress(hHookEngineDll,
"UnhookFunction");
    GetOriginalFunction = (ULONG_PTR (__cdecl *)(ULONG_PTR))GetProcAddress(hHookEngineDll,
"GetOriginalFunction");
    if (HookFunction == NULL || UnhookFunction == NULL || GetOriginalFunction == NULL)
        return;
    HookFunction((ULONG_PTR)
GetProcAddress(GetModuleHandleW(L"kernelbase.dll"),"OpenProcess"), (ULONG_PTR)
&Fake_OpenProcess);
    HookFunction((ULONG_PTR)
GetProcAddress(GetModuleHandleW(L"kernel32.dll"),"CreateProcessW"), (ULONG_PTR)
&Fake_CreateProcessW);
    NtResumeProcess = (NTRESUMEPROCESS)GetProcAddress( GetModuleHandleW(L"ntdll.dll"),
"NtResumeProcess" );
    pfnFindWindowW = (FINDWINDOWW)GetProcAddress( LoadLibraryW(L"user32.dll"),
```

```
"FindWindowW" );
    pfnGetWindowThreadProcessId                                    =
(GETWINDOWTHREADPROCESSID)GetProcAddress(                    LoadLibraryW(L"user32.dll"),
"GetWindowThreadProcessId");
}
```

第四部：卸载 API HOOK（防止进程退出时出现错误）：

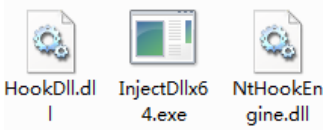
```
Case DLL_PROCESS_DETACH:
    {
UnhookFunction( (ULONG_PTR)GetProcAddress(GetModuleHandleW(L"kernelbase.dll"),"OpenProcess")
 );
UnhookFunction( (ULONG_PTR)GetProcAddress(GetModuleHandleW(L"kernel32.dll"),"CreateProcessW"
) );
        break;
    }
```

把上述两段代码分别编译成 64 位的 DLL，并更名为 NtHookEngine.dll 和 HookDll.DLL。为了让这些 DLL 能在没有安装.NET Framework 4.0 的电脑上使用，在编译前请选择“在静态库中使用 MFC”。编译好后，用 stud_PE 查看导入表，会发现没有 MSVCRT100.DLL 只有 Kernel32.DLL：

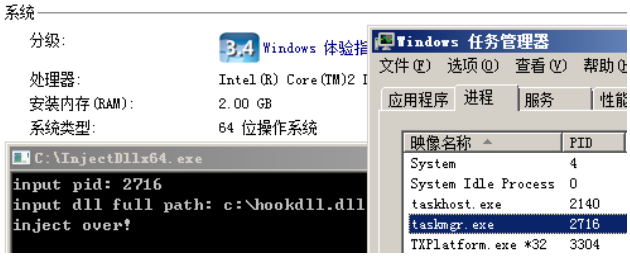


测试效果如下：

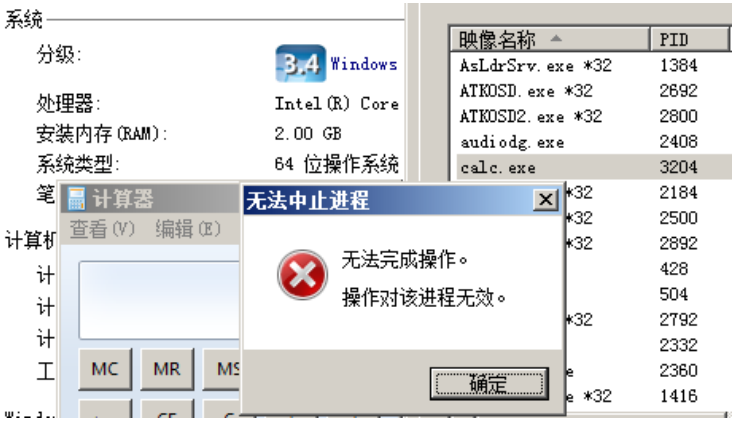
1. 先把那两个 DLL 和一个 EXE 放到 C 盘根目录（一定要这么做!!）：



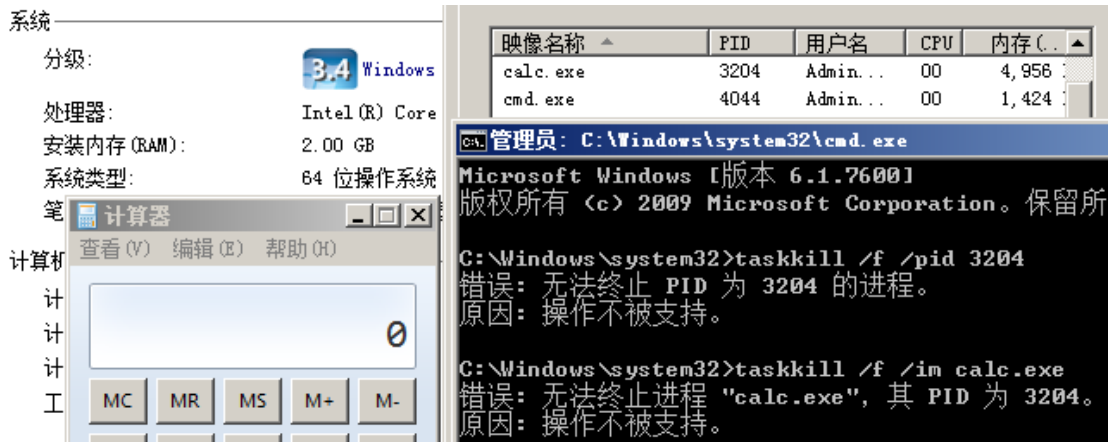
2. 启动 InbjectDllx64.exe，输入“任务管理器”的进程 ID 和要注入的 DLL：



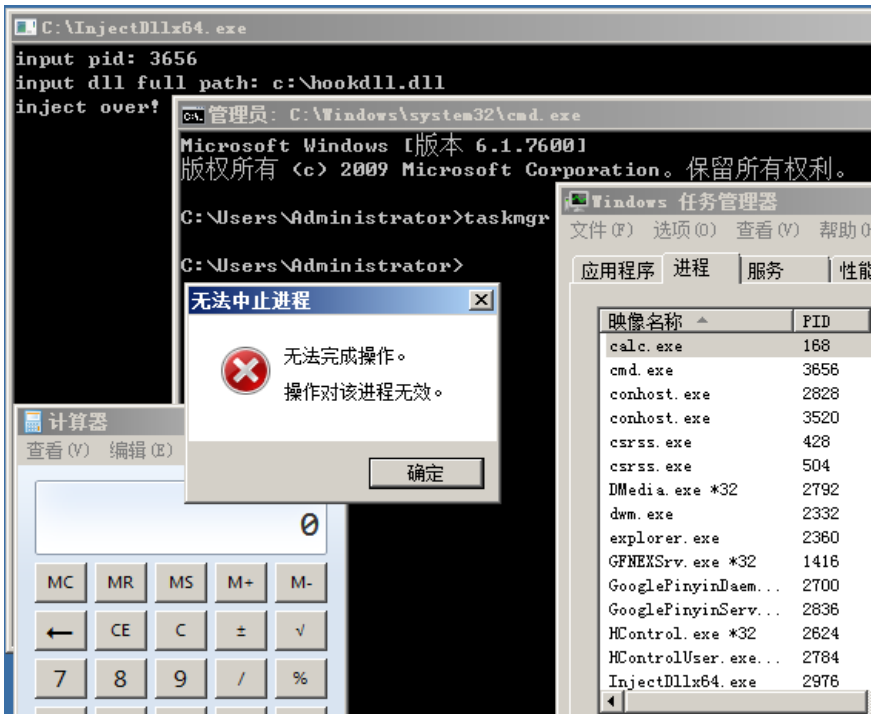
3.尝试结束“计算器”进程：



4.利用任务管理器启动控制台程序 CMD.EXE，并在 CMD.EXE 里启动控制台程序 taskkill.exe 结束“计算器”进程：



5.关闭一切刚才启动的程序，然后启动 CMD.EXE->把 DLL 注入 CMD.EXE 中->利用 CMD.EXE 启动“任务管理器”（taskmgr.exe）->用“任务管理器”结束计算器进程：



请注意：无论是注入 DLL 还是被注入 DLL 的进程必须都是 64 位进程。以上四张截图不仅证明我的钩子起了作用，而且在[被注入了 DLL 的进程 A]启动[新进程 B]时，进程 A 同时也把我们的 DLL 注入到了进程 B 中，无论进程 B 是 CUI 程序还是 GUI 程序。

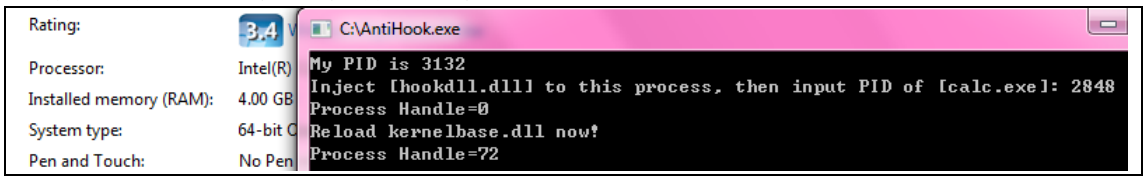
二、ANTI RING3 INLINE HOOK

有了 RING3 INLINE HOOK，自然有人需要反制它。比如 kernelbase!OpenProcess 被挂钩了，怎么绕过钩子呢？我们可以把 kernelbase.dll 复制到 C 盘，并改名为 basekernel.dll，再用 LoadLibraryW 加载这个 DLL。当我要调用 OpenProcess 时，首先获得 basekernel.dll 中 OpenProcess 的地址，再用函数指针的方式调用即可：

```
typedef HANDLE(__stdcall *MYOPENPROCESS)(DWORD,BOOL,DWORD);
int _tmain(int argc, _TCHAR* argv[])
{
    DWORD dwpid=0, mypid=GetCurrentProcessId();
    printf("My PID is %ld\n",mypid);
    printf("Inject [hookdll.dll] to this process, then input PID of [calc.exe]: ");
    scanf("%ld",&dwpid);
    HANDLE h=OpenProcess(1,0,dwpid);
    printf("Process Handle=%ld\n",h);
    TerminateProcess(h,0);
    getchar();
    printf("Reload kernelbase.dll now!");
    getchar();
    CopyFileW(L"c:\\windows\\system32\\kernelbase.dll",L"c:\\basekernel.dll",0);
    HMODULE hlib=LoadLibraryW(L"c:\\basekernel.dll");
    if (hlib==0)
    {
        printf("Cannot load [c:\\basekernel.dll]!");
        getchar();
        return 0;
    }
    MYOPENPROCESS MyOpenProcess=(MYOPENPROCESS)GetProcAddress(hlib,"OpenProcess");
    h=MyOpenProcess(1,0,dwpid);
    printf("Process Handle=%ld\n",h);
    TerminateProcess(h,0);
    CloseHandle(h);
    FreeLibrary(hlib);
    DeleteFileW(L"c:\\basekernel.dll");
    getchar();
    return 0;
```

```
}
```

把含上述代码的进程中的 kernelbase!OpenProcess 勾住无法阻止它杀进程，因为它在打开进程的过程中压根没有用到 kernelbase.dll 中的函数，用的是 basekernel!OpenProcess。尽管 basekernel,dll 根本就是 kernelbase.dll 的马甲，但是系统竟然没有认出来。



三、ANTI-ANTI RING3 INLINE HOOK

如果 RING3 INLINE HOOK 这么容易被绕过，那就太没意思了，针对这种 ANTI 的方法，我又想出了对抗的方法，称为“反反 HOOK”。“反反 HOOK”的方法也很简单暴力，挂钩 ZwReadFile，直接拒绝对被 HOOK DLL 磁盘文件的读取。下面直接贴出对 ZwReadFile 的处理：

```
BOOL __stdcall Fake_ZwReadFile(
    IN HANDLE  FileHandle,
    IN HANDLE  Event  OPTIONAL,
    IN PVOID  ApcRoutine  OPTIONAL,
    IN PVOID  ApcContext  OPTIONAL,
    OUT PIO_STATUS_BLOCK  IoStatusBlock,
    OUT PVOID  Buffer,
    IN ULONG  Length,
    IN PLARGE_INTEGER  ByteOffset  OPTIONAL,
    IN PULONG  Key  OPTIONAL)
{
    long (WINAPI *pZwReadFile)(
    IN HANDLE  FileHandle,
    IN HANDLE  Event  OPTIONAL,
    IN PVOID  ApcRoutine  OPTIONAL,
    IN PVOID  ApcContext  OPTIONAL,
    OUT PIO_STATUS_BLOCK  IoStatusBlock,
    OUT PVOID  Buffer,
    IN ULONG  Length,
    IN PLARGE_INTEGER  ByteOffset  OPTIONAL,
    IN PULONG  Key  OPTIONAL);
    pZwReadFile=(long                                                    (WINAPI
*)(HANDLE,HANDLE,PVOID,PVOID,PIO_STATUS_BLOCK,PVOID,ULONG,PLARGE_INTEGER,PULONG))GetOriginal
Function( (ULONG_PTR)Fake_ZwReadFile );
    char lszsrc[522];
    char dllname[]="kernelbase.dll";
```

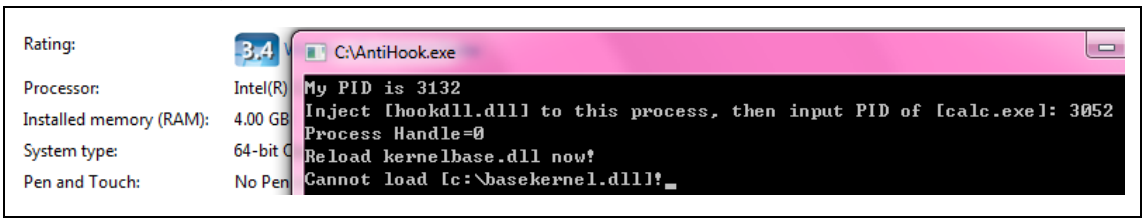
```
    char dllname2[]="ntdll.dll";
    char *ptrx,*ptry;
    WCHAR wszsrc[MAX_PATH+1];
    GetFileNameFromHandleW3(FileHandle,wszsrc);
    PWSTR2PCHAR(wszsrc, lszsrc);
    ptrx = strstr(strlwr(lszsrc), dllname);
    ptry = strstr(strlwr(lszsrc), dllname2);
    if (ptrx!=NULL || ptry!=NULL)
        return 0x80070000;
    else
        return
pZwReadFile(FileHandle,Event,ApcRoutine,ApcContext,IoStatusBlock,Buffer,Length,ByteOffset,Ke
y);
}
```

简单解释以上代码：首先从文件句柄中获得文件名，如果文件名中包含 kernelbase.dll 或者 ntdll.dll 的字样，就返回 0x80070000，否则不做处理。从文件句柄获取文件名的代码如下：

```
LPWSTR GetFileNameFromHandleW3(HANDLE hFile, LPWSTR lpFilePath)
{
    const int FileNameInformation = 9;  // enum FILE_INFORMATION_CLASS;
    typedef struct _IO_STATUS_BLOCK {
        union {
            ULONG  Status;   // NTSTATUS
            PVOID  Pointer;
        };
        ULONG_PTR  Information;
    } IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;  // Defined in Wdm.h
    typedef struct _FILE_NAME_INFORMATION {
        ULONG  FileNameLength;
        WCHAR  FileName[MAX_PATH];
    } FILE_NAME_INFORMATION, *PFILE_NAME_INFORMATION;
    typedef LONG (CALLBACK* ZWQUERYINFORMATIONFILE)(
        HANDLE  FileHandle,
        IO_STATUS_BLOCK *IoStatusBlock,
        PVOID  FileInformation,
        ULONG  Length,
        ULONG  FileInformationClass
    );
    lpFilePath[0] = 0x00;
    HMODULE hNtDLL = GetModuleHandleW(L"ntdll.dll");
    if (hNtDLL == 0x00)                 { return 0x00; }
    ZWQUERYINFORMATIONFILE                      ZwQueryInformationFile                      =
(ZWQUERYINFORMATIONFILE)GetProcAddress(hNtDLL, "ZwQueryInformationFile");
```

```
    if (ZwQueryInformationFile == 0x00) { return 0x00; }
    FILE_NAME_INFORMATION fni;
    IO_STATUS_BLOCK isb;
    if (ZwQueryInformationFile(hFile, &isb, &fni, sizeof(fni), FileNameInformation) != 0)
{ return 0x00; }
    fni.FileName[fni.FileNameLength / sizeof(WCHAR)] = 0x00;
    BY_HANDLE_FILE_INFORMATION fi;
    if(!GetFileInformationByHandle(hFile, &fi) || (fi.dwFileAttributes &
FILE_ATTRIBUTE_DIRECTORY)) { return 0x00; }
    WCHAR szDrive [MAX_PATH];
    WCHAR *lpDrive = szDrive;
    int iPathLen;
    if (GetLogicalDriveStringsW(MAX_PATH - 1, szDrive) >= MAX_PATH)
{ return 0x00; }
    while ((iPathLen = lstrlenW(lpDrive)) != 0) {
        DWORD dwVolumeSerialNumber;
        if(!GetVolumeInformation(lpDrive, NULL, NULL, &dwVolumeSerialNumber,NULL, NULL, NULL,
NULL)) { return 0x00; }
        if (dwVolumeSerialNumber == fi.dwVolumeSerialNumber) {
            lstrcpynW(lpFilePath, lpDrive, lstrlen(lpDrive));
            lstrcatW(lpFilePath, fni.FileName);
            break;
        }
        lpDrive += iPathLen + 1;
    }
    return lpFilePath;
}
```

经测试结果如下：



由截图可知，ZwReadFile 钩子起到了作用。因为无法复制文件，导致重载 kernelbase.dll 失败。不过，这份 ANTI-ANTI INLINE HOOK 的代码要用在商业上，还有很多需要修改的地方（比如禁止从网络上下载 DLL 文件到本地进行 RELOAD）。

思考题：针对 NTDLL 的 RING3 INLINE HOOK 有实际使用价值么？为什么？如果你是一个 RING3 病毒的作者，你会用什么终极办法反制一切 RING3 INLINE HOOK？