



Code Breakers Journal

The CodeBreakers-Journal, Vol. 1, No. 2 (2004)
<http://www.CodeBreakers-Journal.com>

Hooking Windows API - Technics of hooking API functions on Windows

Author: Holy_Father <holy_father@phreaker.net>

Version: 1.1 english

Date: 6.10.2002

Original paper is available at <http://rootkit.host.sk>

Abstract

This text is about hooking API functions on OS Windows. All examples here completely works on Windows systems based on NT technology version NT 4.0 and higher (Windows NT 4.0, Windows 2000, Windows XP). Probably will also work on others Windows systems. You should be familiar with processes on Windows, assembler, PE files structure and some API functions to understand whole text. When using term "Hooking API" here, I mean the full change of API. So, when calling hooked API, our code is run immediately. I do not deal with cases of API monitoring only. I will write about complete hooking.

Contents

I	Introduction	2
II	Hooking methods	2
II-A	Hooking before running	2
II-B	Hooking during running	2
II-B.1	Own process hooking using IAT	3
II-B.2	Own process hooking using entry point rewriting	6
II-B.3	Original function saving	9
II-B.4	Other process hooking	28
III	Ending	30

I. Introduction

This text is about hooking API functions on OS Windows. All examples here completely works on Windows systems based on NT technology version NT 4.0 and higher (Windows NT 4.0, Windows 2000, Windows XP). Probably will also work on others Windows systems. You should be familiar with processes on Windows, assembler, PE files structure and some API functions to understand whole text. When using term "Hooking API" here, I mean the full change of API. So, when calling hooked API, our code is run immediately. I do not deal with cases of API monitoring only. I will write about complete hooking.

II. Hooking methods

Our goal is generally to replace the code of some function with our code. This problem can be sometimes solved before running the process. This can be done mostly with user level process which are run by us and the goal is e.g. to change the program behaviour. Example of this can be application crack. E.g. program which wants original CD-ROM during startup (this was in the game Atlantis) and we want to run it without CDs. If we change the function for getting a drive type we would be able to run this program from the hard drive. This can not be done or we do not want to do this when want to hook system process (e.g. services) or in the case we do not know which process will be the target. Then we will use hooking during running technic. Example of using this can be rootkit or virus with anti-antivirus technics.

A. Hooking before running

This is about physical module change (mostly .exe or .dll) when the function, which we want to change, is. We've got three possibilities at least here on how to do this. The first is to find entry point of that function and basically to rewrite its code. This is limited by the function size but we can load some other modules dynamically (API LoadLibrary), so it could be enough. Kernel functions (kernel32.dll) can be used in all cases because each process in windows has its own copy of this module. Other advantage is if we know on which OS will be changed module run. We can use direct pointers in this case for e.g. API LoadLibraryA. This is because the address of kernel module in memory is static in the scope of one OS Windows version. We can also use behaviour of dynamically loaded module. In this case its initialization part is run immediately after loading to the memory. We are not limited in initialization part of new module. Second possibility of replacing function in module is its extension. Then we have to choose between replacing first 5 bytes by relative jump or rewriting IAT. In the case of relative jump, this will redirect the code execution to our code. When calling function which IAT record is changed, our code will be executed directly after this call. But extension of the module is not so easy because we have to care about DLL header. Next one is replacing the whole module. That means we create own version of the module which can load the original one and call original functions which we are not interested in. But important functions will be totally new. This method is not so good for big modules which can contain hundreds of exports.

B. Hooking during running

Hooking before running is mostly very special and intimately oriented for concrete application (or module). If we replace function in kernel32.dll or in ntdll.dll (only on NT OS) we will get perfect replace of this function in all processes which will be run later, but it is so difficult to make it because we have to take care about accuracy and code perfection of new functions or whole new modules, but the main problem is that only process which will be run later will be hooked (so for all process we have to reboot system). Next problem could be access to these files because NT OS tries to protect them. Much more pretty solution is to hook process during running. This method require more knowledge but the result is perfect. Hooking during running can be done only on process for which we have writing access to their memory. For the writing in itself we will use API function WriteProcessMemory. We will start from hooking our own process during running.

1) *Own process hooking using IAT*: There are many possibilities here. At first I will show you how to hook function by rewriting IAT. Following picture shows structure of PE file:

MS DOS Header ("MZ") and stub	- offset 0
PE signature ("PE")	
.text Program Code	- module code
.data Initialized Data	- initialized (global static) data
.idata Import Table	- information for imported functions and data
.edata Export Table	- information for exported functions and data
Debug symbols	

Important part for us here is Import Address Table (IAT) in the .idata part. This part contains description of imports and mainly imported functions addresses. Now it is important to know how are PE files created. When calling arbitrary API indirectly in programming language (that means we call it using its name, no using its OS specific address) the compiler does not link direct calls to the module but it links call to IAT on jmp instruction which will be filled by process loader while OS is loading process to the memory. This is why we can use the same binary on two different version os Windows where modules can be loaded to another addresses. Process loader will fill out direct jmp instructions in IAT which is used by our calls from the program code. So, if we are able to find out specific function in IAT which we want to hook, we can easily change jmp instruction there and redirect code to our address. Every call after doing this will execute our code. Advantage of this method is its perfection. Disadvantage is often amount of functions which should be hooked (e.g. if we want to change program behaviour in the file searching APIs we will have to change functions FindFirstFile and FindNextFile, but we have to know that these functions have its ANSI and WIDE version, so we have to change IAT address for FindFirstFileA, FindFirstFileW, FindNextFileA and also FileNextFileW. But there still some others like FindFirstFileExA and its WIDE version FindFirstFileExW which are called by previous mentioned functions. We know that FindFirstFileW calls FindFirstFileExW but this is done directly - not using IAT. And still some others to go. There are e.g. ShellAPI functions like SHGetDesktopFolder which also directly calls FindFirstFileW or FindFirstFileExW). But if we will get all of them, the result will be perfect. We can use ImageDirectoryEntryToData from imagehlp.dll to find out IAT easily.

```
PVOID ImageDirectoryEntryToData(  
IN LPVOID Base,  
IN BOOLEAN MappedAsImage,  
IN USHORT DirectoryEntry,  
OUT PULONG Size  
);
```

We will use Instance of our application as Base (Instance can be get by calling GetModuleHandle:

```
hInstance = GetModuleHandleA(NULL);
```

), and as DirectoryEntry we will use constant IMAGE_DIRECTORY_ENTRY_IMPORT.

```
#define IMAGE_DIRECTORY_ENTRY_IMPORT 1
```

Result of this function is pointer to the first IAT record. IAT records are structures which are defined by I IMAGE_IMPORT_DESCRIPTOR. So, the result is a pointer on IMAGE_IMPORT_DESCRIPTOR.

```
typedef struct _IMAGE_THUNK_DATA {  
union {  
PBYTE ForwarderString;  
PDWORD Function;  
DWORD Ordinal;  
PIMAGE_IMPORT_BY_NAME AddressOfData;  
} ;  
} IMAGE_THUNK_DATA, *PIMAGE_THUNK_DATA;
```

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {  
union {  
DWORD Characteristics;  
PIMAGE_THUNK_DATA OriginalFirstThunk;  
} ;  
DWORD TimeDateStamp;  
DWORD ForwarderChain;  
DWORD Name;  
PIMAGE_THUNK_DATA FirstThunk;  
} IMAGE_IMPORT_DESCRIPTOR, *PIMAGE_IMPORT_DESCRIPTOR;
```

The Name value in IMAGE_IMPORT_DESCRIPTOR is a relative reference to the name of module. If we want to hook a function e.g. from kernel32.dll we have to find out in imports which belongs to the descriptor with name kernel32.dll. We will call ImageDirectoryEntryToData at first and then we will try to find descriptor with name "kernel32.dll" (there can be more than one descriptor with this name). Finally we will have to find our function in the list of all functions in the record (address of our function can be get by GetProcAddress function). If we find it we must use VirtualProtect to change memory page protection and after then we can write to this part of memory. After rewriting the address we have to change the protection back. Before calling VirtualProtect we have to know some information about this memory page. This is done by VirtualQuery. We can add some tests in case some calls will fail (e.g. we will not continue if the first VirtualProtect call failed, etc)

```
PCSTR pszHookModName = "kernel32.dll", pszSleepName = "Sleep";
HMODULE hKernel = GetModuleHandle( pszHookModName );
PROC pfnNew = (PROC) 0x12345678, //new address will be here
pfnHookAPIAddr = GetProcAddress( hKernel, pszSleepName );

ULONG ulSize;
PIMAGE_IMPORT_DESCRIPTOR pImportDesc =
( PIMAGE_IMPORT_DESCRIPTOR ) ImageDirectoryEntryToData(
hInstance,
TRUE,
IMAGE_DIRECTORY_ENTRY_IMPORT,
&ulSize
);

while ( pImportDesc->Name )
{
PSTR pszModName = (PSTR) ( (PBYTE) hInstance + pImportDesc->Name );
if ( strcmp( pszModName, pszHookModName ) == 0 )
break;
pImportDesc++;
}

PIMAGE_THUNK_DATA pThunk =
( PIMAGE_THUNK_DATA ) ( (PBYTE) hInstance + pImportDesc->FirstThunk );

while ( pThunk->u1.Function )
{
PROC* ppfn = (PROC*) &pThunk->u1.Function;
BOOL bFound = (*ppfn == pfnHookAPIAddr);

if ( bFound )
{
MEMORY_BASIC_INFORMATION mbi;
VirtualQuery(
ppfn,
&mbi,
sizeof( MEMORY_BASIC_INFORMATION )
);
VirtualProtect(
mbi.BaseAddress,
mbi.RegionSize,
PAGE_READWRITE,
&mbi.Protect )
)

*ppfn = *pfnNew;

DWORD dwOldProtect;
```

```
VirtualProtect(  
mbi.BaseAddress,  
mbi.RegionSize,  
mbi.Protect,  
&dwOldProtect  
);  
break;  
}  
pThunk++;  
}
```

Result of calling Sleep(1000) can be for example this:

```
00407BD8: 68E8030000 push 0000003E8h  
00407BDD: E812FAFFFF call Sleep
```

```
Sleep:      ;this is jump on address in IAT  
004075F4: FF25BCA14000 jmp dword ptr [00040A1BCh]
```

```
original table:  
0040A1BC: 79 67 E8 77 00 00 00 00
```

```
new table:  
0040A1BC: 78 56 34 12 00 00 00 00
```

So the final jump is to 0x12345678.

2) *Own process hooking using entry point rewriting*: The method of rewriting first few instructions on the function entry point is really simple. As in the case of rewriting address in IAT we have to change a page protection at first. Here it will be first 5 bytes of the given function which we want to hook. For later usage we will use dynamical allocation of MEMORY_BASIC_INFORMATION structure. The beginning of the function is get by GetProcAddress again. On this address we will insert relative jump to our code. Following program calls Sleep(5000) (so it will wait for 5 seconds), than the Sleep functions is hooked and redirected to new_sleep, finally it calls Sleep(5000) again. Because new function new_sleep does nothing and returns immediately the whole program will take only 5 in place of 10 seconds.

```
.386p
.model flat, stdcall

includelib lib\kernel32.lib
Sleep PROTO :DWORD
GetModuleHandleA PROTO :DWORD
GetProcAddress PROTO :DWORD,:DWORD
VirtualQuery PROTO :DWORD,:DWORD,:DWORD
VirtualProtect PROTO :DWORD,:DWORD,:DWORD,:DWORD
VirtualAlloc PROTO :DWORD,:DWORD,:DWORD,:DWORD
VirtualFree PROTO :DWORD,:DWORD,:DWORD
FlushInstructionCache PROTO :DWORD,:DWORD,:DWORD
GetCurrentProcess PROTO
ExitProcess PROTO :DWORD
```

```
.data
```

```
kernel_name db "kernel32.dll",0
sleep_name db "Sleep",0
old_protect dd ?
```

```
MEMORY_BASIC_INFORMATION_SIZE equ 28
```

```
PAGE_READWRITE dd 000000004h
PAGE_EXECUTE_READWRITE dd 000000040h
MEM_COMMIT dd 000001000h
MEM_RELEASE dd 000008000h
```

```
.code
```

```
start:
```

```
push 5000
call Sleep
```

```
do_hook:
```

```
push offset kernel_name
call GetModuleHandleA
push offset sleep_name
push eax
call GetProcAddress
mov edi,eax ;finally got Sleep address
```

```
push PAGE_READWRITE
push MEM_COMMIT
push MEMORY_BASIC_INFORMATION_SIZE
push 0
call VirtualAlloc
```

```
test eax,eax
jz do_sleep
mov esi,eax ;allocation for MBI

push MEMORY_BASIC_INFORMATION_SIZE
push esi
push edi
call VirtualQuery ;inforamtion about the memory page
test eax,eax
jz free_mem

call GetCurrentProcess
push 5
push edi
push eax
call FlushInstructionCache ;just to be sure :)

lea eax,[esi+014h]
push eax
push PAGE_EXECUTE_READWRITE
lea eax,[esi+00Ch]
push [eax]
push [esi]
call VirtualProtect ;we will change protection for a moment
;so we will be able to write there
test eax,eax
jz free_mem

mov byte ptr [edi],0E9h ;to write relative jump
mov eax,offset new_sleep
sub eax,edi
sub eax,5
inc edi
stosd ;this is relative address for jump

push offset old_protect
lea eax,[esi+014h]
push [eax]
lea eax,[esi+00Ch]
push [eax]
push [esi]
call VirtualProtect ;return back the protection of page

free_mem:
push MEM_RELEASE
push 0
push esi
call VirtualFree ;free memory
```



```
do_sleep:
push 5000
call Sleep
push 0
call ExitProcess
new_sleep:
ret 004h
end start
```

Result of the second call Sleep is this:

```
004010A4: 6888130000 push 000001388h
004010A9: E80A000000 call Sleep
```

```
Sleep: ;toto je jump na adresu v IAT
004010B8: FF2514204000 jmp dword ptr [000402014h]
```

```
tabulka:
00402014: 79 67 E8 77 6C 7D E8 77
```

```
Kernel32.Sleep:
77E86779: E937A95788 jmp 0004010B5h
```

```
new_sleep:
004010B5: C20400 ret 004h
```

3) *Original function saving*: Mostly we need more than a function hook. For example in case when we don't want to replace the given function but only to check its result, or in case when we want to replace the function only sometimes e.g. when it is called with specific arguments. Good example of this is already mentioned files hiding done by replacing FindXXXFile functions. So if we want to hide specific files and don't want to be noticeable we have to leave original function for all other files without changing the functions behavior. This is simple when using method of rewriting IAT. For calling original function we can get its original address with GetProcAddress and then call it directly. But the problem occurs when using rewriting entry point method. By rewriting those 5 bytes at the functions entry point we lost original function irrecoverably. So we need to save first instructions. We can use following technic. We know we will rewrite only first 5 bytes but don't know how many instructions there are or how long they are. We have to reserve enough memory for first instructions. 16 bytes could be enough because there are usually not long instructions at the begin of function. Probably we can use less then 16. Whole reserved memory will be filled with 0x90 (0x90 = nop) in case there are shorter instructions. Next 5 bytes will be relative jump which will be filled later.

```
old_hook: db 090h,090h,090h,090h,090h,090h,090h,090h
db 090h,090h,090h,090h,090h,090h,090h,090h
db 0E9h,000h,000h,000h,000h
```

Now we are ready to copy first instructions. It is a long stuff to get instruction length, this is why we will work with the complete engine. This was made by ZOMBiE. Input argument is instruction address for which we want to get length. Output is commonly in eax.

```

; LDE32, Length-Disassembler Engine, 32-bit, (x) 1999-2000 ZOMBiE
; special edition for REVERT tool

; version 1.05

C_MEM1          equ      0001h          ; |
C_MEM2          equ      0002h          ; |may be used simultaneously
C_MEM4          equ      0004h          ; |
C_DATA1         equ      0100h          ; |
C_DATA2         equ      0200h          ; |may be used simultaneously
C_DATA4         equ      0400h          ; |
C_67            equ      0010h          ; used with C_PREFIX
C_MEM67         equ      0020h          ; C_67 ? C_MEM2 : C_MEM4
C_66            equ      1000h          ; used with C_PREFIX
C_DATA66        equ      2000h          ; C_66 ? C_DATA2 : C_DATA4
C_PREFIX        equ      0008h          ; prefix. take opcode again
C_MODRM         equ      4000h          ; MODxxxR/M
C_DATAW0        equ      8000h          ; opc&1 ? C_DATA66 : C_DATA1

                p386
                model flat
                locals @@

                .code

public          disasm_main
public         _disasm_main
public         @disasm_main
public         DISASM_MAIN

disasm_main:
_disasm_main:
@disasm_main:
DISASM_MAIN:

; returns opcode length in EAX or -1 if error

; input: pointer to opcode

; __fastcall          EAX
; __cdecl             [ESP+4]

;this is my first change here, it's the label only for calling this function
get_instr_len:

                mov     ecx, [esp+4]      ; ECX = opcode ptr

                xor     edx, edx          ; flags

```

```

                                xor     eax, eax
@@prefix:                       and     dl, not C_PREFIX

                                mov     al, [ecx]
                                inc     ecx

                                or      edx, table_1[eax*4]

                                test    dl, C_PREFIX
                                jnz     @@prefix

                                cmp     al, 0F6h
                                je      @@test
                                cmp     al, 0F7h
                                je      @@test

                                cmp     al, 0CDh
                                je      @@int

                                cmp     al, 0Fh
                                je      @@0F

@@cont:                          test    dh, C_DATAW0 shr 8
                                jnz     @@dataw0

@@dataw0done:                   test    dh, C_MODRM shr 8
                                jnz     @@modrm

@@exitmodrm:                   test    dl, C_MEM67
                                jnz     @@mem67

@@mem67done:                   test    dh, C_DATA66 shr 8
                                jnz     @@data66

@@data66done:                  mov     eax, ecx
                                sub     eax, [esp+4]

                                and     edx, C_MEM1+C_MEM2+C_MEM4+C_DATA1+C_DATA2+C_DATA4
                                add     al, dl
                                add     al, dh

;my second change heer, there was retn only in original version
@@exit:                         ret     00004h

@@test:                         or      dh, C_MODRM shr 8
                                test    byte ptr [ecx], 00111000b ; F6/F7 -- test
                                jnz     @@cont

```

```
        or     dh, C_DATAW0 shr 8
        jmp    @@cont

@@int:   or     dh, C_DATA1 shr 8
        cmp    byte ptr [ecx], 20h
        jne    @@cont
        or     dh, C_DATA4 shr 8
        jmp    @@cont

@@0F:   mov     al, [ecx]
        inc    ecx
        or     edx, table_0F[eax*4]

        cmp    edx, -1
        jne    @@cont

@@error: mov     eax, edx
        jmp    @@exit

@@dataw0: xor     dh, C_DATA66 shr 8
        test   al, 00000001b
        jnz    @@dataw0done
        xor     dh, (C_DATA66+C_DATA1) shr 8
        jmp    @@dataw0done

@@mem67: xor     dl, C_MEM2
        test   dl, C_67
        jnz    @@mem67done
        xor     dl, C_MEM4+C_MEM2
        jmp    @@mem67done

@@data66: xor     dh, C_DATA2 shr 8
        test   dh, C_66 shr 8
        jnz    @@data66done
        xor     dh, (C_DATA4+C_DATA2) shr 8
        jmp    @@data66done

@@modrm: mov     al, [ecx]
        inc    ecx

        mov     ah, al ; ah=mod, al=rm

        and    ax, 0C007h
        cmp    ah, 0C0h
        je     @@exitmodrm

        test   dl, C_67
        jnz    @@modrm16
```

```

@@modrm32:      cmp     al, 04h
                jne     @@a

                mov     al, [ecx]      ; sib
                inc     ecx
                and     al, 07h

@@a:           cmp     ah, 40h
                je      @@mem1
                cmp     ah, 80h
                je      @@mem4

                cmp     ax, 0005h
                jne     @@exitmodrm

@@mem4:       or      dl, C_MEM4
                jmp     @@exitmodrm

@@mem1:       or      dl, C_MEM1
                jmp     @@exitmodrm

@@modrm16:    cmp     ax, 0006h
                je      @@mem2
                cmp     ah, 40h
                je      @@mem1
                cmp     ah, 80h
                jne     @@exitmodrm

@@mem2:       or      dl, C_MEM2
                jmp     @@exitmodrm

                endp

                .data

;0F          -- analyzed in code, no flags (i.e.flags must be 0)
;F6,F7      -- --//-- (ttt=000 -- 3 bytes, otherwise 2 bytes)
;CD         -- --//-- (6 bytes if CD 20, 2 bytes otherwise)

table_1          label    dword    ; normal instructions

dd C_MODRM      ; 00
dd C_MODRM      ; 01
dd C_MODRM      ; 02
dd C_MODRM      ; 03
dd C_DATAW0     ; 04
dd C_DATAW0     ; 05

```

```
dd 0 ; 06
dd 0 ; 07
dd C_MODRM ; 08
dd C_MODRM ; 09
dd C_MODRM ; 0A
dd C_MODRM ; 0B
dd C_DATAW0 ; 0C
dd C_DATAW0 ; 0D
dd 0 ; 0E
dd 0 ; 0F
dd C_MODRM ; 10
dd C_MODRM ; 11
dd C_MODRM ; 12
dd C_MODRM ; 13
dd C_DATAW0 ; 14
dd C_DATAW0 ; 15
dd 0 ; 16
dd 0 ; 17
dd C_MODRM ; 18
dd C_MODRM ; 19
dd C_MODRM ; 1A
dd C_MODRM ; 1B
dd C_DATAW0 ; 1C
dd C_DATAW0 ; 1D
dd 0 ; 1E
dd 0 ; 1F
dd C_MODRM ; 20
dd C_MODRM ; 21
dd C_MODRM ; 22
dd C_MODRM ; 23
dd C_DATAW0 ; 24
dd C_DATAW0 ; 25
dd C_PREFIX ; 26
dd 0 ; 27
dd C_MODRM ; 28
dd C_MODRM ; 29
dd C_MODRM ; 2A
dd C_MODRM ; 2B
dd C_DATAW0 ; 2C
dd C_DATAW0 ; 2D
dd C_PREFIX ; 2E
dd 0 ; 2F
dd C_MODRM ; 30
dd C_MODRM ; 31
dd C_MODRM ; 32
dd C_MODRM ; 33
dd C_DATAW0 ; 34
dd C_DATAW0 ; 35
```

```
dd C_PREFIX          ; 36
dd 0                  ; 37
dd C_MODRM           ; 38
dd C_MODRM           ; 39
dd C_MODRM           ; 3A
dd C_MODRM           ; 3B
dd C_DATAW0          ; 3C
dd C_DATAW0          ; 3D
dd C_PREFIX          ; 3E
dd 0                  ; 3F
dd 0                  ; 40
dd 0                  ; 41
dd 0                  ; 42
dd 0                  ; 43
dd 0                  ; 44
dd 0                  ; 45
dd 0                  ; 46
dd 0                  ; 47
dd 0                  ; 48
dd 0                  ; 49
dd 0                  ; 4A
dd 0                  ; 4B
dd 0                  ; 4C
dd 0                  ; 4D
dd 0                  ; 4E
dd 0                  ; 4F
dd 0                  ; 50
dd 0                  ; 51
dd 0                  ; 52
dd 0                  ; 53
dd 0                  ; 54
dd 0                  ; 55
dd 0                  ; 56
dd 0                  ; 57
dd 0                  ; 58
dd 0                  ; 59
dd 0                  ; 5A
dd 0                  ; 5B
dd 0                  ; 5C
dd 0                  ; 5D
dd 0                  ; 5E
dd 0                  ; 5F
dd 0                  ; 60
dd 0                  ; 61
dd C_MODRM           ; 62
dd C_MODRM           ; 63
dd C_PREFIX          ; 64
dd C_PREFIX          ; 65
```

```
dd C_PREFIX+C_66      ; 66
dd C_PREFIX+C_67      ; 67
dd C_DATA66           ; 68
dd C_MODRM+C_DATA66   ; 69
dd C_DATA1            ; 6A
dd C_MODRM+C_DATA1    ; 6B
dd 0                  ; 6C
dd 0                  ; 6D
dd 0                  ; 6E
dd 0                  ; 6F
dd C_DATA1            ; 70
dd C_DATA1            ; 71
dd C_DATA1            ; 72
dd C_DATA1            ; 73
dd C_DATA1            ; 74
dd C_DATA1            ; 75
dd C_DATA1            ; 76
dd C_DATA1            ; 77
dd C_DATA1            ; 78
dd C_DATA1            ; 79
dd C_DATA1            ; 7A
dd C_DATA1            ; 7B
dd C_DATA1            ; 7C
dd C_DATA1            ; 7D
dd C_DATA1            ; 7E
dd C_DATA1            ; 7F
dd C_MODRM+C_DATA1    ; 80
dd C_MODRM+C_DATA66   ; 81
dd C_MODRM+C_DATA1    ; 82
dd C_MODRM+C_DATA1    ; 83
dd C_MODRM            ; 84
dd C_MODRM            ; 85
dd C_MODRM            ; 86
dd C_MODRM            ; 87
dd C_MODRM            ; 88
dd C_MODRM            ; 89
dd C_MODRM            ; 8A
dd C_MODRM            ; 8B
dd C_MODRM            ; 8C
dd C_MODRM            ; 8D
dd C_MODRM            ; 8E
dd C_MODRM            ; 8F
dd 0                  ; 90
dd 0                  ; 91
dd 0                  ; 92
dd 0                  ; 93
dd 0                  ; 94
dd 0                  ; 95
```



```
dd 0 ; 96
dd 0 ; 97
dd 0 ; 98
dd 0 ; 99
dd C_DATA66+C_MEM2 ; 9A
dd 0 ; 9B
dd 0 ; 9C
dd 0 ; 9D
dd 0 ; 9E
dd 0 ; 9F
dd C_MEM67 ; A0
dd C_MEM67 ; A1
dd C_MEM67 ; A2
dd C_MEM67 ; A3
dd 0 ; A4
dd 0 ; A5
dd 0 ; A6
dd 0 ; A7
dd C_DATA1 ; A8
dd C_DATA66 ; A9
dd 0 ; AA
dd 0 ; AB
dd 0 ; AC
dd 0 ; AD
dd 0 ; AE
dd 0 ; AF
dd C_DATA1 ; B0
dd C_DATA1 ; B1
dd C_DATA1 ; B2
dd C_DATA1 ; B3
dd C_DATA1 ; B4
dd C_DATA1 ; B5
dd C_DATA1 ; B6
dd C_DATA1 ; B7
dd C_DATA66 ; B8
dd C_DATA66 ; B9
dd C_DATA66 ; BA
dd C_DATA66 ; BB
dd C_DATA66 ; BC
dd C_DATA66 ; BD
dd C_DATA66 ; BE
dd C_DATA66 ; BF
dd C_MODRM+C_DATA1 ; C0
dd C_MODRM+C_DATA1 ; C1
dd C_DATA2 ; C2
dd 0 ; C3
dd C_MODRM ; C4
dd C_MODRM ; C5
```

```
dd C_MODRM+C_DATA1      ; C6
dd C_MODRM+C_DATA66     ; C7
dd C_DATA2+C_DATA1      ; C8
dd 0                     ; C9
dd C_DATA2              ; CA
dd 0                     ; CB
dd 0                     ; CC
dd 0                     ; CD
dd 0                     ; CE
dd 0                     ; CF
dd C_MODRM              ; D0
dd C_MODRM              ; D1
dd C_MODRM              ; D2
dd C_MODRM              ; D3
dd C_DATA1              ; D4
dd C_DATA1              ; D5
dd 0                     ; D6
dd 0                     ; D7
dd C_MODRM              ; D8
dd C_MODRM              ; D9
dd C_MODRM              ; DA
dd C_MODRM              ; DB
dd C_MODRM              ; DC
dd C_MODRM              ; DD
dd C_MODRM              ; DE
dd C_MODRM              ; DF
dd C_DATA1              ; E0
dd C_DATA1              ; E1
dd C_DATA1              ; E2
dd C_DATA1              ; E3
dd C_DATA1              ; E4
dd C_DATA1              ; E5
dd C_DATA1              ; E6
dd C_DATA1              ; E7
dd C_DATA66             ; E8
dd C_DATA66             ; E9
dd C_DATA66+C_MEM2     ; EA
dd C_DATA1              ; EB
dd 0                     ; EC
dd 0                     ; ED
dd 0                     ; EE
dd 0                     ; EF
dd C_PREFIX             ; F0
dd 0                     ; F1
dd C_PREFIX             ; F2
dd C_PREFIX             ; F3
dd 0                     ; F4
dd 0                     ; F5
```


dd -1 ; 23
dd -1 ; 24
dd -1 ; 25
dd -1 ; 26
dd -1 ; 27
dd -1 ; 28
dd -1 ; 29
dd -1 ; 2A
dd -1 ; 2B
dd -1 ; 2C
dd -1 ; 2D
dd -1 ; 2E
dd -1 ; 2F
dd -1 ; 30
dd -1 ; 31
dd -1 ; 32
dd -1 ; 33
dd -1 ; 34
dd -1 ; 35
dd -1 ; 36
dd -1 ; 37
dd -1 ; 38
dd -1 ; 39
dd -1 ; 3A
dd -1 ; 3B
dd -1 ; 3C
dd -1 ; 3D
dd -1 ; 3E
dd -1 ; 3F
dd -1 ; 40
dd -1 ; 41
dd -1 ; 42
dd -1 ; 43
dd -1 ; 44
dd -1 ; 45
dd -1 ; 46
dd -1 ; 47
dd -1 ; 48
dd -1 ; 49
dd -1 ; 4A
dd -1 ; 4B
dd -1 ; 4C
dd -1 ; 4D
dd -1 ; 4E
dd -1 ; 4F
dd -1 ; 50
dd -1 ; 51
dd -1 ; 52

```
dd -1 ; 53
dd -1 ; 54
dd -1 ; 55
dd -1 ; 56
dd -1 ; 57
dd -1 ; 58
dd -1 ; 59
dd -1 ; 5A
dd -1 ; 5B
dd -1 ; 5C
dd -1 ; 5D
dd -1 ; 5E
dd -1 ; 5F
dd -1 ; 60
dd -1 ; 61
dd -1 ; 62
dd -1 ; 63
dd -1 ; 64
dd -1 ; 65
dd -1 ; 66
dd -1 ; 67
dd -1 ; 68
dd -1 ; 69
dd -1 ; 6A
dd -1 ; 6B
dd -1 ; 6C
dd -1 ; 6D
dd -1 ; 6E
dd -1 ; 6F
dd -1 ; 70
dd -1 ; 71
dd -1 ; 72
dd -1 ; 73
dd -1 ; 74
dd -1 ; 75
dd -1 ; 76
dd -1 ; 77
dd -1 ; 78
dd -1 ; 79
dd -1 ; 7A
dd -1 ; 7B
dd -1 ; 7C
dd -1 ; 7D
dd -1 ; 7E
dd -1 ; 7F
dd C_DATA66 ; 80
dd C_DATA66 ; 81
dd C_DATA66 ; 82
```

```
dd C_DATA66 ; 83
dd C_DATA66 ; 84
dd C_DATA66 ; 85
dd C_DATA66 ; 86
dd C_DATA66 ; 87
dd C_DATA66 ; 88
dd C_DATA66 ; 89
dd C_DATA66 ; 8A
dd C_DATA66 ; 8B
dd C_DATA66 ; 8C
dd C_DATA66 ; 8D
dd C_DATA66 ; 8E
dd C_DATA66 ; 8F
dd C_MODRM ; 90
dd C_MODRM ; 91
dd C_MODRM ; 92
dd C_MODRM ; 93
dd C_MODRM ; 94
dd C_MODRM ; 95
dd C_MODRM ; 96
dd C_MODRM ; 97
dd C_MODRM ; 98
dd C_MODRM ; 99
dd C_MODRM ; 9A
dd C_MODRM ; 9B
dd C_MODRM ; 9C
dd C_MODRM ; 9D
dd C_MODRM ; 9E
dd C_MODRM ; 9F
dd 0 ; A0
dd 0 ; A1
dd 0 ; A2
dd C_MODRM ; A3
dd C_MODRM+C_DATA1 ; A4
dd C_MODRM ; A5
dd -1 ; A6
dd -1 ; A7
dd 0 ; A8
dd 0 ; A9
dd 0 ; AA
dd C_MODRM ; AB
dd C_MODRM+C_DATA1 ; AC
dd C_MODRM ; AD
dd -1 ; AE
dd C_MODRM ; AF
dd C_MODRM ; B0
dd C_MODRM ; B1
dd C_MODRM ; B2
```

```
dd C_MODRM          ; B3
dd C_MODRM          ; B4
dd C_MODRM          ; B5
dd C_MODRM          ; B6
dd C_MODRM          ; B7
dd -1               ; B8
dd -1               ; B9
dd C_MODRM+C_DATA1 ; BA
dd C_MODRM          ; BB
dd C_MODRM          ; BC
dd C_MODRM          ; BD
dd C_MODRM          ; BE
dd C_MODRM          ; BF
dd C_MODRM          ; C0
dd C_MODRM          ; C1
dd -1               ; C2
dd -1               ; C3
dd -1               ; C4
dd -1               ; C5
dd -1               ; C6
dd -1               ; C7
dd 0                ; C8
dd 0                ; C9
dd 0                ; CA
dd 0                ; CB
dd 0                ; CC
dd 0                ; CD
dd 0                ; CE
dd 0                ; CF
dd -1               ; D0
dd -1               ; D1
dd -1               ; D2
dd -1               ; D3
dd -1               ; D4
dd -1               ; D5
dd -1               ; D6
dd -1               ; D7
dd -1               ; D8
dd -1               ; D9
dd -1               ; DA
dd -1               ; DB
dd -1               ; DC
dd -1               ; DD
dd -1               ; DE
dd -1               ; DF
dd -1               ; E0
dd -1               ; E1
dd -1               ; E2
```

```
dd -1 ; E3
dd -1 ; E4
dd -1 ; E5
dd -1 ; E6
dd -1 ; E7
dd -1 ; E8
dd -1 ; E9
dd -1 ; EA
dd -1 ; EB
dd -1 ; EC
dd -1 ; ED
dd -1 ; EE
dd -1 ; EF
dd -1 ; F0
dd -1 ; F1
dd -1 ; F2
dd -1 ; F3
dd -1 ; F4
dd -1 ; F5
dd -1 ; F6
dd -1 ; F7
dd -1 ; F8
dd -1 ; F9
dd -1 ; FA
dd -1 ; FB
dd -1 ; FC
dd -1 ; FD
dd -1 ; FE
dd -1 ; FF
```

end

Now we are able to get instruction length on arbitrary address. We will repeat this call until 5 bytes are read. After this we will copy these bytes to `old_hook`. We know how long are first instructions, so we can fill out the relative jump address on the next instruction in original function.

```
.386p
.model flat, stdcall

...

.data

kernel_name db "kernel32.dll",0
sleep_name db "Sleep",0

...

MEM_RELEASE dd 000008000h

;16 nops + one relative jump
old_sleep db 090h,090h,090h,090h,090h,090h,090h,090h,
            090h,090h,090h,090h,090h,090h,090h,
            0E9h,000h,000h,000h,000h

.code
start:
push 5000
call Sleep

do_hook:
push offset kernel_name
call GetModuleHandleA
push offset sleep_name
push eax
call GetProcAddress
    push eax
mov esi,eax

xor ecx,ecx
mov ebx,esi
get_five_bytes:
push ecx
push ebx
call get_instr_len ;calling LDE32
pop ecx
add ecx,eax
add ebx,eax
cmp ecx,5
```

```
jnb get_five_bytes
mov edi,offset old_sleep ;counting relative jump address
mov [edi+011h],ebx
sub [edi+011h],edi
sub dword ptr [edi+011h],015h
rep movsb
pop edi
```

;following code was above, so without comments

```
push PAGE_READWRITE
push MEM_COMMIT
push MEMORY_BASIC_INFORMATION_SIZE
push 0
call VirtualAlloc
test eax,eax
jz do_sleep
mov esi,eax
```

```
push MEMORY_BASIC_INFORMATION_SIZE
push esi
push edi
call VirtualQuery
test eax,eax
jz free_mem
```

```
call GetCurrentProcess
push 5
push edi
push eax
call FlushInstructionCache
```

```
lea eax,[esi+014h]
push eax
push PAGE_EXECUTE_READWRITE
lea eax,[esi+00Ch]
push [eax]
push [esi]
call VirtualProtect
test eax,eax
jz free_mem
```

```
mov byte ptr [edi],0E9h
mov eax,offset new_sleep
sub eax,edi
sub eax,5
inc edi
stosd
```

```
push offset old_protect
lea eax,[esi+014h]
push [eax]
lea eax,[esi+00Ch]
push [eax]
push [esi]
call VirtualProtect

    free_mem:
push MEM_RELEASE
push 0
push esi
call VirtualFree
    do_sleep:
push 5000
call Sleep
push 0
call ExitProcess
    new_sleep:
mov eax,dword ptr [esp+004h]
add eax,eax ;doubling timeout
push eax
mov eax,offset old_sleep          ;calling old function
call eax
ret 004h
```

After the hook it will look like this:

```
004010CC: 6888130000 push 000001388h
004010D1: E818090000 call Sleep
```

```
Sleep:          ;this is jump on address in IAT
004019EE: FF2514204000 jmp dword ptr [000402014h]
```

```
tabulka:
00402014: 79 67 E8 77 6C 7D E8 77
```

```
Kernel32.Sleep:
77E86779: E95FA95788 jmp 0004010DDh
```

```
new_sleep:
004010DD: 8B442404 mov eax,dword ptr [esp+4]
004010E1: 03C0 add eax,eax
004010E3: 50 push eax
004010E4: B827304000 mov eax,000403027h
004010E9: FFD0 call eax
```

```
old_sleep:
00403027: 6A00 push 0
00403029: FF742408 push dword ptr [esp+8]
0040302D: 90 nop
0040302E: 90 nop
0040302F: 90 nop
00403030: 90 nop
00403031: 90 nop
00403032: 90 nop
00403033: 90 nop
00403034: 90 nop
00403035: 90 nop
00403036: 90 nop
00403037: E94337A877 jmp Kernel32.77E8677F
```

;this instruction is placed 1 byte after first instruction at Kernel32.Sleep
(77E86779)

```
Kernel32.77E8677F:
77E8677F: E803000000 call Kernel32.SleepEx
... ;following is unimportant
```

To make this clearer, this is how the original version of Kernel32.Sleep looks:

```
Kernel32.Sleep:
77E86779: 6A00 push 0
77E8677B: FF742408 push dword ptr [esp+8]
77E8677F: E803000000 call Kernel32.SleepEx
77E86784: C20400 ret 00004h
```

As you can see we copied first and second instruction (it was 6 bytes here) and the relative jump pointed on the next instruction and that is how it should be. We have to supposed here that relative jumps are not placed as the first bytes of functions. If there would be we've got a problem. Next problem is with APIs like ntdll.DbgBreakPoint. These are too short for this method of hooking. And forasmuch as it is called by Kernel32.DebugBreak, it is not hookable by changing IAT. But who want to hook function which does only the int 3 call? But nothing is impossible. You can think about it and you can find how to solve this. As I was thinking about this you can hook the following function after this one (it would be damaged by rewritng first 5 bytes of the previous function). Function DbgBreakPoint is 2 bytes long, so we can set some flags here and try to write conditional jump on the begining of the second function ... But this is not our problem now. With the problem of saving original function relates then unhooking. Unhooking is changing replaces bytes back to the original state. When rewriting IAT you will have to return original address to the table if you want to do unhooking. When using the five byte patch you will have to copy first original instructions back. Both ways are realy simple and no need to write more about this.

4) *Other process hooking*: Now we will do something practical with hooking during running. Who want to deal with hooking own process? That is good only for learning basics but it is not much practical. I'll show you three methods of other process hooking. Two of them use API CreateRemoteThread which is only in Windows with NT technology. The problem of hooking is not so interesing for older windows version for me. After all I will try to

explain third method which I didn't practise, so it could be unfunctional. At first few about CreateRemoteThread. As the help says this function creates new thread in any process and runs its code.

```
HANDLE CreateRemoteThread(  
HANDLE hProcess,  
LPSECURITY_ATTRIBUTES lpThreadAttributes,  
DWORD dwStackSize,  
LPTHREAD_START_ROUTINE lpStartAddress,  
LPVOID lpParameter,  
DWORD dwCreationFlags,  
LPDWORD lpThreadId  
);
```

The handle hProcess can be get by OpenProcess. Here we have to have necessary rights. The pointer lpStartAddress points on memory place in TARGET process where the first instruction for new thread is. Because new thread is created in target process it is in memory of target process. The pointer lpParameter points on argument which will be referred to the new thread.

a) DLL Injection: We are able to run new thread from any place in target process memory. This is useless unless we have own code in it. The first method cheats on this. It uses GetProcAddress to get actual address for LoadLibrary. Then routes lpStartAddress to the address of LoadLibrary. Function LoadLibrary has only one parameter like the function for new thread in target process.

```
HINSTANCE LoadLibrary(  
LPCTSTR lpLibFileName  
);
```

We will use this similarity and we will refer the name of our DLL library as lpParameter. After running new thread lpParameter will be on a place of lpLibFileName. The most important thing here is behavior described above. After loading new module into target process memory the initialization part is executed. If we place specific functions which will hook functions we want to we will win this stuff. After execution of initialization part, the thread will have nothing to do and close but our module is still in memory. This method is really nice and easy to implement. This is called DLL Injection. But if you are like I am, you don't like must of having DLL library. But if one doesn't care about having this library it is the easiest and the fastest method (from the programmers sight).

b) Independent code: Going on the way of independent code is very difficult but also very impressive thing. Independent code is the code without any statical addresses. Everything is relative in it towards some specific place in itself. This code is mostly done if we don't know the address where this code will be executed. Sure, it is possible to get this address and then to relink our code so as it will behave on the new address without errors but this is even harder than coding independent code. Example of this kind of code can be the virus code. The virus which infects executables in the way it adds itself somewhere into this executable. In different executables will be the virus code on different places depended e.g. on file structure on length. At first we have to insert our code into target process. Then function CreateRemoteThread will take care of running our code. So, at first we have to get some information about target process and get handle with OpenProcess. Then VirtualAllocEx will alloc some space in remote process memory for our code. Finally we will use WriteProcessMemory to write our code on allocated memory and run it. In CreateRemoteThread lpStartAddress will refer to allocated memory and lpParameter can be whatever we want. Because I really don't like any unnecessary files I use this method.

c) Raw change: There is not `CreateRemoteThread` in older windows version (without NT). So we can't use this for hooking. There are probably other and better methods how to hook than the method I will talk about now. In fact I don't know if this will work in practice (one never know when use Windows) but theoretically is everything ok. We don't need to have our code in target process to hook its functions at all. We have function `WriteProcessMemory` (this should be in all Windows version) and we have `OpenProcess`, too. Last thing we need is `VirtualProtectEx` which can change access to memory pages in target process. I can't see any reason why not to hook target process functions directly from our process...

III. Ending

This small document ends. I will greet any extension which will describe unmentione methods of hook, I am sure there are a lot of them. I will also greet any extensions in the parts which were not described so intimately. You can also send me some source codes if they are fecund for the problem of hooking e.g. for parts where I was lazy to write the code. The goal of this document is to show deatails of every technics of hooking. I hope I've done the part of this. Special thanks to ZOMBiE for his work, so I haven't to code it myself and spend ages by studying tables for getting instruction length.