

Powerful x86/x64 Mini Hook-Engine

- [Download demo projects and sources](#)



Introduction

I wrote this little hook-engine for a much bigger article. Sometimes it seems such a waste to write valuable code for large articles whose topic isn't directly related to the code. This often leads to the problem that the code won't be found by the people who are looking for it.

Personally, I would've used Microsoft's Detour hook engine, but the free license only applies to x86 applications, and that seemed a little bit too restrictive to me. So, I decided to write my own engine in order to support x64 as well. I've never downloaded Detour nor have I ever seen its APIs, but from the general overview given by Microsoft it's easy to guess how it works.

As I said, this is only a part of something bigger. It's not perfect, but it can easily become such. Since this is not a beginner's guide about hooking, I assume that the reader already possesses the necessary knowledge to understand the material. If you never heard about this subject, you'd better start with another article. There's plenty of guides out there, no need to repeat the same things here.

As everybody knows there's only one easy and secure way to hook a Win32 API: to put an unconditional jump at the beginning of the code to redirect it to the hooked function. And by secure I just mean that our hook can't be bypassed. Of course, there are some other ways, but they're either complicated or insane or both. A proxy dll, for instance, might work in some cases, but it's rather insane for system dlls. Overwriting the IAT is unsecure for two reasons:

- a) The program might use GetProcAddress to retrieve the address of an API (and in that case we should handle this API as well).
- b) It's not always possible, there are many cases as for packed programs where the IAT gets built by the protection code and not by the Windows loader.

Ok, I guess you're convinced. Let's just say that there's a reason why Microsoft uses the method presented in this article.

How it works

A common technique used in combination with the unconditional jump is:

Hooked API

Jump to our code

Our code restores
API's original bytes

We call the API

We restore the hook

This approach may seem unsafe in a multi-threading environment and it is. It might work, but our technique is much more powerful. Well, nothing new, we just put our unconditional jump at the beginning of the code we want to hook and we put the original instructions of the API elsewhere in memory. When the hooked function jumps to our code we can call the bridge we created, which, after the first instructions, will jump to the API code which follows our unconditional jump:

Hooked API

Jump to our code

Our code calls
the bridge

The bridge jumps to the
API code after our jump

Let's make a real world example. If the first instructions of the function/API we want to hook are:

```
mov edi, edi  
push ebp  
mov ebp, esp  
xor ecx, ecx
```

They will be replaced by our:

```
00400000 jmp our_code  
00400005 xor ecx, ecx
```

Our bridge will look like this:

```
mov edi, edi  
push ebp  
mov ebp, esp  
jmp 00400005
```

Of course, to know the size of the instructions we're going to replace we need a disassembler both for x86 and x64. I searched on google for an x64 disassembler and found the [diStorm64 disassembler](#). I quote from its homepage:

diStorm64 is a professional quality open source disassembler library for AMD64, licensed under the BSD license.

diStorm is a binary stream disassembler. It's capable of disassembling 80x86 instructions in 64 bits (AMD64, X86-64) and both in 16 and 32 bits. In addition, it disassembles FPU, MMX, SSE, SSE2, SSE3, SSSE3, SSE4, 3DNow! (w/ extensions), new x86-64 instruction sets, VMX, and AMD's SVM! diStorm was written to decode quickly every instruction as accurately as possible. Robust decoding, while taking special care for valid or unused prefixes, is what makes this disassembler powerful, especially for research. Another benefit that might come in handy is that the module was written as multi-threaded, which means you could disassemble several streams or more simultaneously. For rapidly use, diStorm is compiled for Python and is easily used in C as well. diStorm was originally written under Windows and ported later to Linux and Mac. The source code is portable and platform independent (supports both little and big endianity). It also can be used as a ring0 disassembler (tested as a kernel driver using the DDK under Windows)!

This sounded pretty good to me. Now that we have our disassembler we can start!

The first thing I wanted to know was if it was possible to create bridges without having to relocate jumps. As the reader knows jumps, most of the time, have a relative address as operand and not an absolute one (through registers). This leads to the problem that I can't relocate a jump without having to recalculate its relative address. Also, I wanted to test if this disassembler really worked fine. So, I wrote a little program which creates a log file of all the instructions of all exported functions in a dll which are going to be overwritten by an unconditional jump. Here's the code:

```
#include "stdafx.h"
#include "distorm.h"
#include <stdlib.h>
#include <stdlib.h>
#include <Windows.h>

DWORD RvaToOffset(IMAGE_NT_HEADERS *NT, DWORD Rva);
VOID AddFunctionToLog(FILE *Log, BYTE *FileBuf, DWORD FuncRVA);
VOID GetInstructionString(char *Str, _DecodedInst *Instr);

int _tmain(int argc, _TCHAR* argv[])
{
    if (argc < 2) return 0;

    //
    // Open log file
    //

    FILE *Log = NULL;

    if (_tfopen_s(&Log, argv[2], _T("w")) != 0)
        return 0;

    //
    // Open PE file
    //

    HANDLE hFile = CreateFile(argv[1], GENERIC_READ, FILE_SHARE_READ, NULL,
        OPEN_EXISTING, 0, NULL);

    if (hFile == INVALID_HANDLE_VALUE)
    {
        fclose(Log);
        return 0;
    }

    DWORD FileSize = GetFileSize(hFile, NULL);

    BYTE *FileBuf = new BYTE [FileSize];
```

```

DWORD BRW;

if (FileBuf)
    ReadFile(hFile, FileBuf, FileSize, &BRW, NULL);

CloseHandle(hFile);

IMAGE_DOS_HEADER *pDosHeader = (IMAGE_DOS_HEADER *) FileBuf;
IMAGE_NT_HEADERS *pNtHeaders = (IMAGE_NT_HEADERS *) ((FileBuf != NULL ?
    pDosHeader->e_lfanew : 0) + (ULONG_PTR) FileBuf);

if (!FileBuf || pDosHeader->e_magic != IMAGE_DOS_SIGNATURE ||
    pNtHeaders->Signature != IMAGE_NT_SIGNATURE ||
    pNtHeaders->OptionalHeader.DataDirectory
    [IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress == 0)
{
    fclose(Log);
    if (FileBuf)
        delete FileBuf;
    return 0;
}

//
// Walk through export dir's functions
//

DWORD ET_RVA = pNtHeaders->OptionalHeader.DataDirectory
    [IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;

IMAGE_EXPORT_DIRECTORY *pExportDir = (IMAGE_EXPORT_DIRECTORY *)
    (RvaToOffset(pNtHeaders, ET_RVA) + (ULONG_PTR) FileBuf);

DWORD *pFunctions = (DWORD *) (RvaToOffset(pNtHeaders,
    pExportDir->AddressOfFunctions) + (ULONG_PTR) FileBuf);

for (DWORD x = 0; x < pExportDir->NumberOfFunctions; x++)
{
    if (pFunctions[x] == 0) continue;

    AddFunctionToLog(Log, FileBuf, pFunctions[x]);
}

fclose(Log);
delete FileBuf;

return 0;
}

//
// This function adds to the log the instructions
// at the beginning of each function which are going
// to be overwritten by the hook jump
//

VOID AddFunctionToLog(FILE *Log, BYTE *FileBuf, DWORD FuncRVA)
{
#define MAX_INSTRUCTIONS 100

    IMAGE_NT_HEADERS *pNtHeaders = (IMAGE_NT_HEADERS *)
        ((* (IMAGE_DOS_HEADER *) FileBuf).e_lfanew + (ULONG_PTR) FileBuf);

    _DecodeResult res;
    _DecodedInst decodedInstructions[MAX_INSTRUCTIONS];
    unsigned int decodedInstructionsCount = 0;

#ifdef _M_IX86

```

```

    _DecodeType dt = Decode32Bits;

#define JUMP_SIZE 5

#else ifdef _M_AMD64

    _DecodeType dt = Decode64Bits;

#define JUMP_SIZE 13 // worst case scenario

#endif

_OffsetType offset = 0;

res = distorm_decode(offset, // offset for buffer, e.g. 0x00400000
    (const BYTE *) &FileBuf[RvaToOffset(pNtHeaders, FuncRVA)],
    50, // function size (code size to disasm)
    dt, // x86 or x64?
    decodedInstructions, // decoded instr
    MAX_INSTRUCTIONS, // array size
    &decodedInstructionsCount // how many instr were disassembled?
);

if (res == DECRES_INPUTERR)
    return;

DWORD InstrSize = 0;

for (UINT x = 0; x < decodedInstructionsCount; x++)
{
    if (InstrSize >= JUMP_SIZE)
        break;

    InstrSize += decodedInstructions[x].size;

    char Instr[100];
    GetInstructionString(Instr, &decodedInstructions[x]);

    fprintf(Log, "%s\n", Instr);
}

fprintf(Log, "\n\n\n");
}

VOID GetInstructionString(char *Str, _DecodedInst *Instr)
{
    wsprintfA(Str, "%s %s", Instr->mnemonic.p, Instr->operands.p);
    _strlwr_s(Str, 100);
}

DWORD RvaToOffset(IMAGE_NT_HEADERS *NT, DWORD Rva)
{
    DWORD Offset = Rva, Limit;
    IMAGE_SECTION_HEADER *Img;
    WORD i;

    Img = IMAGE_FIRST_SECTION(NT);

    if (Rva < Img->PointerToRawData)
        return Rva;

    for (i = 0; i < NT->FileHeader.NumberOfSections; i++)
    {
        if (Img[i].SizeOfRawData)
            Limit = Img[i].SizeOfRawData;
        else
            Limit = Img[i].Misc.VirtualSize;
    }
}

```

```

    if (Rva >= Img[i].VirtualAddress &&
        Rva < (Img[i].VirtualAddress + Limit))
    {
        if (Img[i].PointerToRawData != 0)
        {
            Offset -= Img[i].VirtualAddress;
            Offset += Img[i].PointerToRawData;
        }

        return Offset;
    }
}

return NULL;
}

```

The command line syntax is: pefile logfile (e.g.: disasmtest ntdll.dll ntdll.log). As you can see, I only took 5 bytes for x86 hooks, but for x64 I took 13, which is really a lot. It's possible to use 5 bytes jumps, but it's necessary to check that there's less than 2GB between the original function and our code and between the bridge and the original function. In addition, for every new hook I would've to recreate the bridge. So, I decided to implement only the 13 bytes absolute jump, which translated in instructions would be:

```

mov r15, 0000056000010000
jmp r15

```

It's not a great optimization, but you can easily change this behaviour from the hook engine's code. As for the results of the little program above, I created logs for the ntdll.dll and advapi32.dll both in x86 and x64. Here, for instance, is a small part of the ntdll.dll x86 log:

```

mov edi, edi
push ebp
mov ebp, esp

```

```

mov edi, edi
push ebp
mov ebp, esp

```

```

push 0x18
push 0x7c92de18

```

```

mov edi, edi
push ebp
mov ebp, esp

```

```

push 0x14
push 0x7c931a28

```

This is of course pretty encouraging, but let's see the results for the x64 platform.

```

sub rsp, 0x48
mov rax, [rsp+0x78]
mov byte [rsp+0x30], 0x0

```

```

mov [rsp+0x10], rbx
mov [rsp+0x18], rbp
mov [rsp+0x20], rsi

```

```

push rsi

```

```

push r14
push r15
sub rsp, 0x480

mov rax, rsp
mov [rax+0x8], rbx
mov [rax+0x10], rsi
mov [rax+0x18], r12

sub rsp, 0x38
mov [rsp+0x20], r8
mov r9d, edx
mov r8, rcx

mov rax, rsp
mov [rax+0x8], rsi
mov [rax+0x10], rdi
mov [rax+0x18], r12

mov [rsp+0x10], rbx
mov [rsp+0x18], rsi
push rdi
push r12

sub rsp, 0x68
mov rax, r9
mov r9d, [rsp+0xb0]

```

But what about the functions which just call a syscall after moving a number into a register like NtCreateProcess, NtOpenKey etc.? These functions have very few instructions and our 13 bytes jump will overwrite more code than the one of the function itself. But that doesn't seem to be a problem, since as we can see from the disassembler these functions have a 16-bytes alignment. So, we won't overwrite other functions code anyway.

```

.text:0000000078ED1260
.text:0000000078ED1260
.text:0000000078ED1260 ZwOpenThread
.text:0000000078ED1260
.text:0000000078ED1263
.text:0000000078ED1268
.text:0000000078ED126A
.text:0000000078ED126A ZwOpenThread
.text:0000000078ED126A
.text:0000000078ED126A ; -----
.text:0000000078ED126B align 10h

public ZwOpenThread
proc near
mov     r10, rcx           ; NtOpenThread
mov     eax, 0FCh
syscall
retn
endp

```

Here's the code of the hook engine:

```

#include "stdafx.h"
#include <stdlib.h>
#include "distorm.h"

// 10000 hooks should be enough
#define MAX_HOOKS 10000

typedef struct _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];

UINT NumberOfHooks = 0;

BYTE *pBridgeBuffer = NULL; // Here are going to be stored all the bridges

UINT CurrentBridgeBufferSize = 0; // This number is incremented as
                                   // the bridge buffer is growing

#ifdef _M_IX86

#define JUMP_SIZE      5

#else ifdef _M_AMD64

#define JUMP_SIZE      13    // Worst case scenario
                             // only the case when there's more than
                             // 2GB from the api to the bridge

#endif

#define SMALL_JUMP      5

#define KB_SIZE          1024
#define MB_SIZE          (1024 * KB_SIZE)
#define GB_SIZE          (1024 * MB_SIZE)

BOOL WINAPI DllMain(HMODULE hModule, DWORD dwReason, LPVOID lpReserved)
{
    switch (dwReason)
    {
    case DLL_PROCESS_ATTACH:
    {
        pBridgeBuffer = (BYTE *) malloc(MAX_HOOKS * (JUMP_SIZE * 3));

        if (pBridgeBuffer)
        {
            DWORD dwOldProtect = 0;

            VirtualProtect(pBridgeBuffer, MAX_HOOKS * (JUMP_SIZE * 3),
                PAGE_EXECUTE_READWRITE, &dwOldProtect);
        }
    }

    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }

    return TRUE;
}

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;
}

#ifdef _M_AMD64

//
// This function evaluates if a 64bit jump is necessary
//

BOOL Is64bitJumpNeeded(ULONG_PTR PosA, ULONG_PTR PosB)
{
    return TRUE; // Always return TRUE

    ULONG_PTR res = max(PosA, PosB) - min(PosA, PosB);

    return res < (GB_SIZE * 2) ? FALSE : TRUE;
}

#endif

//
// This function writes unconditional jumps
// both for x86 and x64
//

VOID WriteJump(VOID *pAddress, ULONG_PTR JumpTo)
{
    DWORD dwOldProtect = 0;

    VirtualProtect(pAddress, JUMP_SIZE, PAGE_EXECUTE_READWRITE, &dwOldProtect);

    BYTE *pCur = (BYTE *) pAddress;

#ifdef _M_IX86

    *pCur = 0xE9; // jmp

    DWORD RelAddr = (DWORD) (JumpTo - (ULONG_PTR) pAddress) - JUMP_SIZE;

    memcpy(++pCur, &RelAddr, sizeof (DWORD));

#else ifdef _M_AMD64

    if (Is64bitJumpNeeded((ULONG_PTR) pAddress, JumpTo) == FALSE)
    {
        // TODO: Write small jump if a < 2GB jmp is enough
    }
    else
    {
        *pCur = 0x49; // mov r15, ...
        *(++pCur) = 0xBF;
        memcpy(++pCur, &JumpTo, sizeof (ULONG_PTR));
        pCur += sizeof (ULONG_PTR);
        *pCur = 0x41; // jmp r15
        *(++pCur) = 0xFF;
        *(++pCur) = 0xE7;
    }

#endif

    DWORD dwBuf = 0; // nessary othewrise the function fails

    VirtualProtect(pAddress, JUMP_SIZE, dwOldProtect, &dwBuf);

```

```

}

//
// This function creates a bridge of the original function
//

VOID *CreateBridge(ULONG_PTR Function, const UINT JumpSize = JUMP_SIZE)
{
    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)
        dt, // 50 instr should be _quite_ enough
        decodedInstructions, // x86 or x64?
        MAX_INSTRUCTIONS, // decoded instr
        &decodedInstructionsCount, // array size
        ); // 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);
        }
        */

#ifdef _M_IX86

```

```

        CurrentBridgeBufferSize += JUMP_SIZE;

#else ifdef _M_AMD64

        if (Is64bitJumpNeeded((ULONG_PTR)
&pBridgeBuffer[CurrentBridgeBufferSize], Dest))
            CurrentBridgeBufferSize += JUMP_SIZE;
        else
            CurrentBridgeBufferSize += SMALL_JUMP;

#endif

    }
    else
    {
        /*
        memcpy(&pBridgeBuffer[CurrentBridgeBufferSize],
            (VOID *) pCurInstr, decodedInstructions[x].size);

        CurrentBridgeBufferSize += decodedInstructions[x].size;
        //}

        InstrSize += decodedInstructions[x].size;
    }

    WriteJump(&pBridgeBuffer[CurrentBridgeBufferSize], Function + InstrSize);
    CurrentBridgeBufferSize += JUMP_SIZE;

    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;

#ifdef _M_IX86

        VOID *pBridge = CreateBridge(OriginalFunction);

#else ifdef _M_AMD64

        UINT JumpSize = Is64bitJumpNeeded(OriginalFunction, NewFunction)
            == TRUE ? JUMP_SIZE : SMALL_JUMP;
        VOID *pBridge = CreateBridge(OriginalFunction, JumpSize);

#endif

        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;
}

```

As you can see I implemented it as a dll, which brings us to the next paragraph.

Using the code

Using the code is very simple. Basically, the dll only exports 3 functions: one to hook, another to unhook and the last to get the address of the bridge of the hooked function. Of course, we need to retrieve the address of the bridge, otherwise we can't call the original code of the hooked function.

Let's see a little code sample which works both on x86 and x64:

```
#include "stdafx.h"
#include "NtHookEngine_Test.h"

BOOL (__cdecl *HookFunction)(ULONG_PTR OriginalFunction, ULONG_PTR NewFunction);
VOID (__cdecl *UnhookFunction)(ULONG_PTR Function);
ULONG_PTR (__cdecl *GetOriginalFunction)(ULONG_PTR Hook);

int WINAPI MyMessageBoxW(HWND hWnd, LPCWSTR lpText, LPCWSTR lpCaption,
                        UINT uType, WORD wLanguageId, DWORD dwMilliseconds);

int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPTSTR lpCmdLine, int nCmdShow)
{
    //
    // Retrive hook functions
    //

    HMODULE hHookEngineDll = LoadLibrary(_T("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 0;

    //
    // Hook MessageBoxTimeoutW
    //

    HookFunction((ULONG_PTR) GetProcAddress(LoadLibrary(_T("User32.dll")),
        "MessageBoxTimeoutW"),
        (ULONG_PTR) &MyMessageBoxW);

    MessageBox(0, _T("Hi, this is a message box!"), _T("This is the title."),
        MB_ICONINFORMATION);

    //
    // Unhook MessageBoxTimeoutW
    //

    UnhookFunction((ULONG_PTR) GetProcAddress(LoadLibrary(_T("User32.dll")),
        "MessageBoxTimeoutW"));

    MessageBox(0, _T("Hi, this is a message box!"), _T("This is the title."),
        MB_ICONINFORMATION);

    return 0;
}

int WINAPI MyMessageBoxW(HWND hWnd, LPCWSTR lpText, LPCWSTR lpCaption, UINT
uType,
                        WORD wLanguageId, DWORD dwMilliseconds)
{
    int (WINAPI *pMessageBoxW)(HWND hWnd, LPCWSTR lpText,
        LPCWSTR lpCaption, UINT uType, WORD wLanguageId,
        DWORD dwMilliseconds);

    pMessageBoxW = (int (WINAPI *) (HWND, LPCWSTR, LPCWSTR, UINT, WORD, DWORD))
        GetOriginalFunction((ULONG_PTR) MyMessageBoxW);
```

```

return pMessageBoxW(hWnd, lpText, L"Hooked MessageBox",
    uType, wLanguageId, dwMilliseconds);
}

```

In this sample I'm hooking the API "MessageBoxTimeoutW". I tried to hook MessageBoxW and that worked fine on x86, then I tried on x64 and the code generated an exception. So, I disassembled the MessageBoxW function on x64:

```

.text:0000000078CE6880 ; int __stdcall MessageBoxW(HWND hWnd,LPCWSTR lpText,LPCWSTR lpCap
.text:0000000078CE6880 public MessageBoxW
.text:0000000078CE6880 MessageBoxW      ; CODE XREF: __ClientNoMem
.text:0000000078CE6880
.text:0000000078CE6880 var_18      = word ptr -18h
.text:0000000078CE6880 var_10      = dword ptr -10h
.text:0000000078CE6880
.text:0000000078CE6880 sub        rsp, 38h
.text:0000000078CE6884 cmp        cs:gfEMIEEnable, 0
.text:0000000078CE6888 jz         short loc_78CE68BC
.text:0000000078CE688D mov        rax, gs:30h
.text:0000000078CE6896 mov        r10, [rax+48h]
.text:0000000078CE689A xor        eax, eax
.text:0000000078CE689C lock cmpxchg cs:gdwEMIThreadID, r10
.text:0000000078CE68A5 mov        r10, cs:gpReturnAddr
.text:0000000078CE68AC mov        eax, 1
.text:0000000078CE68B1 cmovz     r10, rax
.text:0000000078CE68B5 mov        cs:gpReturnAddr, r10
.text:0000000078CE68BC
.text:0000000078CE68BC loc_78CE68BC:      ; CODE XREF: MessageBoxW+B
.text:0000000078CE68BC or         [rsp+38h+var_10], 0FFFFFFFFh
.text:0000000078CE68C1 and         [rsp+38h+var_18], 0
.text:0000000078CE68C7 call        MessageBoxTimeoutW
.text:0000000078CE68CC add        rsp, 38h
.text:0000000078CE68D0 retn
.text:0000000078CE68D0 MessageBoxW      endp

```

Unfortunately, as you can notice, the first instructions of this API include a jz which is going to be overwritten by our unconditional jump. And since we don't relocate jumps in our bridge, we can't hook this function. So, I had to hook the function MessageBoxTimeoutW, which is called inside MessageBoxW and has no jumps at the beginning.

In the code example I first hook the function and call it, then I unhook it and call it again. So, the output will be:



That's all. Of course, this code works only if MessageBoxTimeoutW is available. I'm not completely sure about when it was first introduced, since it's an undocumented API. I guess it has been introduced with XP, so chances are that this particular hook won't work on Windows 2000.

Conclusions

As it's possible to see from the previous example, the hook engine isn't perfect, but it can easily be

improved. I don't develop it further because I don't need a more powerful one (right now, I mean). I just needed an x86/x64 hook engine with no license restrictions. I wrote this engine and the article in just one day, it really wasn't much work. Most of the work in such a hook engine is writing the disassembler, which I didn't do. So, in my opinion, it doesn't make much sense paying for a hook engine. The only thing which I really can't provide in this engine is support for Itanium. That's because I don't have a disassembler for this platform. But I would rather write one myself than buying a hook engine. I might actually add an Itanium disassembler in the future, who knows...

I hope you can find this code useful.

Daniel Pistelli