
Subject: Function Hooking

Posted by [Neijwiert](#) on Wed, 24 Dec 2014 20:26:37 GMT

[View Forum Message](#) <> [Reply to Message](#)

I've been trying and trying but I just cannot figure it out. I even asked it on StackOverflow but they aren't helping much either.

What I'm trying to achieve is: Call a custom (hook) function and then calling the original function (a trampoline effect, whilst keeping the stack intact between the functions).

I did somehow achieve this when I hooked into the Commands->Find_Object function, I intercepted all calls to it and it worked fine. So I was all happy and satisfied how that worked out and I tried to put everything in a nice class. When I did that it just stopped working. So I was like, well yeah that sometimes happens you mess up some simple thing whilst copying it over. So I moved everything back to the old (messy) code and to my surprise that stopped working aswell...

My new code is actually pretty much a complete mirror and I doubt that the copy pasting went wrong so I'm just gonna put that version here in the hopes somebody of the renegade community can help me to find the problem.

I have the following header file:

Toggle Spoiler

```
#ifndef TIMEMACHINE_INCLUDE__DETOURS_H
#define TIMEMACHINE_INCLUDE__DETOURS_H

#define Detours DetourClass::Get_Instance()

class DetourClass
{
    friend class TimeMachine;

private:
    struct DetourFunction
    {
        DetourFunction(ULONG ID, BYTE *OriginalFunction, BYTE *HookFunction, BYTE
*OriginalFunctionCode)
        {
            this->ID = ID;
            this->OriginalFunction = OriginalFunction;
            this->HookFunction = HookFunction;
            this->OriginalFunctionCode = OriginalFunctionCode;
        }

        ~DetourFunction()
        {
            free(this->OriginalFunctionCode);
        }
    };
};
```

```

    ULONG ID;
    BYTE *OriginalFunction;
    BYTE *HookFunction;
    BYTE *OriginalFunctionCode;
};

public:
    ULONG Install_Detour(BYTE *TargetAddress, BYTE *HookAddress);
    void Remove_Detour(ULONG ID);
    void Remove_All_Detours();
    long Jump_To_Original(ULONG ID);

    static DetourClass *Get_Instance();
protected:
    DetourClass();
    ~DetourClass();

    static DetourClass *Instance;
private:
    int Calculate_Offset_Address(BYTE *StartAddress, BYTE *TargetAddress);

    ULONG _FreeID;
    SimpleDynVecClass<DetourFunction *> _Functions;
};

#endif

```

Which has the following source file:

Toggle Spoiler

```

#include "General.h"
#include "engine.h"
#include "Detours.h"

static const unsigned int JMP32_SZ = 5; // the size of JMP <address>
static const unsigned int NOP = 0x90; // opcode for NOP
static const unsigned int JMP = 0xE9; // opcode for JUMP

DetourClass::DetourClass()
{
    this->_FreeID = 1;
}

DetourClass::~~DetourClass()
{

```

```

Remove_All_Detours();
}

#pragma optimize( "", off )
ULONG DetourClass::Install_Detour(BYTE *TargetAddress, BYTE *HookAddress)
{
    DWORD OldProtect;
    if (!VirtualProtect(TargetAddress, JMP32_SZ, PAGE_EXECUTE_READWRITE, &OldProtect)) //
    Make sure we are allowed to modify that memory area
    {
        return 0;
    }

    BYTE *OriginalCode = (BYTE *)malloc(JMP32_SZ); // Reserve space to store the overwritten
    data of the original part
    memcpy(OriginalCode, TargetAddress, JMP32_SZ); // Copy original part of the function
    memset(TargetAddress, NOP, JMP32_SZ); // Good practice to NOP original part

    // Insert a jump to the hook function in the original code
    int HookJumpOffset = Calculate_Offset_Address(TargetAddress, HookAddress);
    TargetAddress[0] = JMP;
    memcpy(TargetAddress + 1, &HookJumpOffset, 4);
    //////////////////////////////////////

    VirtualProtect(TargetAddress, JMP32_SZ, OldProtect, NULL);

    ULONG NewID = this->_FreeID++;
    this->_Functions.Add(new DetourFunction(NewID, TargetAddress, HookAddress, OriginalCode));

    return NewID;
}
#pragma optimize( "", on )

void DetourClass::Remove_Detour(ULONG ID)
{
    if (ID == 0)
    {
        return;
    }

    for (int x = this->_Functions.Count() - 1; x >= 0; x--)
    {
        DetourFunction *CurFunction = this->_Functions[x];
        if (CurFunction->ID == ID) // Check if the current detour has the same target as the one we're
        trying to remove
        {
            this->_Functions.Delete(x); // Remove that one
        }
    }
}

```

```

// Try to restore to old situation
DWORD OldProtect;
if (VirtualProtect(CurFunction->OriginalFunction, JMP32_SZ, PAGE_EXECUTE_READWRITE,
&OldProtect))
{
    memcpy(CurFunction->OriginalFunction, CurFunction->OriginalFunctionCode, JMP32_SZ); //
Copy original code back

    VirtualProtect(CurFunction->OriginalFunction, JMP32_SZ, OldProtect, NULL);
}
////////////////////////////////////

delete CurFunction; // Free up resources

return;
}
}
}

void DetourClass::Remove_All_Detours()
{
for (int x = this->_Functions.Count() - 1; x >= 0; x--)
{
    DetourFunction *CurFunction = this->_Functions[x];

    // Try to restore to old situation
    DWORD OldProtect;
    if (VirtualProtect(CurFunction->OriginalFunction, JMP32_SZ, PAGE_EXECUTE_READWRITE,
&OldProtect))
    {
        memcpy(CurFunction->OriginalFunction, CurFunction->OriginalFunctionCode, JMP32_SZ); //
Copy original code back

        VirtualProtect(CurFunction->OriginalFunction, JMP32_SZ, OldProtect, NULL);
    }
    //////////////////////////////////////

    delete CurFunction; // Free up resources
}

this->_Functions.Delete_All();
}

long DetourClass::Jump_To_Original(ULONG ID)
{
if (ID == 0)
{
    return NULL;
}
}

```

```

}

for (int x = this->_Functions.Count() - 1; x >= 0; x--)
{
    DetourFunction *CurFunction = this->_Functions[x];
    if (CurFunction->ID == ID)
    {
        BYTE *ASMCode = (BYTE *)VirtualAlloc(0, JMP32_SZ + JMP32_SZ, MEM_COMMIT,
        PAGE_EXECUTE_READWRITE); // Reserve space for run-time generated asm code

        memcpy(ASMCode, CurFunction->OriginalFunctionCode, JMP32_SZ); // Copy the original code
        to the beginning

        int OriginalJumpOffset = Calculate_Offset_Address(ASMCode + JMP32_SZ,
        CurFunction->OriginalFunction + JMP32_SZ); // Calculate jump offset to original function
        ASMCode[JMP32_SZ] = JMP; // Insert the jump opcode
        memcpy(ASMCode + JMP32_SZ + 1, &OriginalJumpOffset, 4); // Copy the jump address

        long ReturnValue = ((long*)(void))ASMCode(); // Execute the code and get the return value (if
        any)

        VirtualFree(ASMCode, JMP32_SZ + JMP32_SZ, MEM_DECOMMIT); // Free the code

        return ReturnValue;
    }
}

return NULL;
}

DetourClass *DetourClass::Get_Instance()
{
    return DetourClass::Instance;
}

int DetourClass::Calculate_Offset_Address(BYTE *StartAddress, BYTE *TargetAddress)
{
    return (((int)TargetAddress - (int)StartAddress) - JMP32_SZ);
}

DetourClass *DetourClass::Instance = NULL;

```

In my plugin source file I have these calls to the DetourClass:
Toggle Spoiler

```

ULONG ObjectHookID = 0;

```

```

ULONG FooHookID = 0;
int Foo()
{
    Console_Output("Normal Foo\n");

    return 5;
}

int Foo_Hook()
{
    Console_Output("Hook Foo\n");

    return (int)Detours->Jump_To_Original(FooHookID);
}

GameObject *Find_Object_Hook(int obj_id)
{
    Console_Output("Finding object with id: %d\n", obj_id);

    return (GameObject *)Detours->Jump_To_Original(ObjectHookID);
}

TimeMachine::TimeMachine()
{
    DetourClass::Instance = new DetourClass();

    RegisterEvent(EVENT_LOAD_LEVEL_HOOK, this);

    //ObjectHookID = Detours->Install_Detour(&Commands->Find_Object, &Find_Object_Hook);
    FooHookID = Detours->Install_Detour((BYTE *)&Foo, (BYTE *)&Foo_Hook);
    if (FooHookID == 0)
    {
        Console_Output("Install failed\n");
    }
}

TimeMachine::~TimeMachine()
{
    delete DetourClass::Instance;

    UnregisterEvent(EVENT_LOAD_LEVEL_HOOK, this);

    Console_Output(__FUNCTION__ "\n");
}

void TimeMachine::OnLoadLevel()
{
    Console_Output("%d\n", Foo());
}

```

}

Where TimeMachine is my plugin class (So the constructor gets called when SSGM loads the library). I have checked if the memory is actually changed after my function calls and it is indeed changed to the correct variables. But as soon as I call Foo() it just executes it as if nothing changed. The reason that I have it in OnLoadLevel is because in one of my earlier tests it started working when I moved it to OnLoadLevel (so outside the constructor). The reason why I'm not trying to hook Find_Object right now is because when internal engine calls go to Find_Object when the game starts it crashes the fds instantly (The hooking in the constructor goes without any problems).

If somebody comes up with a solution or pushes me in the right direction that would be greatly appreciated!

Subject: Re: Function Hooking
Posted by [jonwil](#) on Wed, 24 Dec 2014 21:42:45 GMT
[View Forum Message](#) <> [Reply to Message](#)

If you want to hook Commands->Find_Object, just read the address out of that variable (the "original" Find_Object) then replace it with the address of your new function.
Your new funxtion would then call the stock function through the pointer you saved ealier.
p

Subject: Re: Function Hooking
Posted by [Neijwiert](#) on Thu, 25 Dec 2014 00:41:32 GMT
[View Forum Message](#) <> [Reply to Message](#)

jonwil wrote on Wed, 24 December 2014 14:42If you want to hook Commands->Find_Object, just read the address out of that variable (the "original" Find_Object) then replace it with the address of your new function.
Your new funxtion would then call the stock function through the pointer you saved ealier.
p

That would result in an infinite loop? For example:

```
typedef GameObject (**FindObjectPointer)(int);
FindObjectPointer OriginalFindObject;
GameObject *FindObject_Test(int obj_id)
{
    Console_Output("Finding object with id: %d\n", obj_id);

    return (* OriginalFindObject)(obj_id);
}
```

```
}  
  
TimeMachine::TimeMachine()  
{  
    OriginalFindObject = &Commands->Find_Object;  
    *&Commands->Find_Object = &Find_Object_Test;  
}
```

The OriginalFindObject would point right back to the hooked one. Im trying to catch all calls to the original method and then do some stuff. I'm just using Find_Object as an example, the actual command I'm going to target is Start_Timer.

When I compile and run this I get an infinite loop.

NOTE: I'm also trying to catch calls to the method outside of my DLL. So there's no other way than memory hooking it with a jump? Or am I just thinking to difficult right now?

Subject: Re: Function Hooking
Posted by [jonwil](#) on Thu, 25 Dec 2014 01:29:05 GMT
[View Forum Message](#) <> [Reply to Message](#)

do this:
OriginalFindObject = Commands->Find_Object;
Commands->Find_Object = Find_Object_Test;

then later do
return OriginalFindObject(bj_id);

Subject: Re: Function Hooking
Posted by [Neijwiert](#) on Thu, 25 Dec 2014 12:37:22 GMT
[View Forum Message](#) <> [Reply to Message](#)

I must be completely retarded...
Talking about taking the hard route...
Well thanks, it works. But I'm still kind of curious as to why my first approach doesn't work. It isn't even affecting the execution of the original function, yet I'm overwriting memory.

Subject: Re: Function Hooking
Posted by [iRANian](#) on Sun, 28 Dec 2014 14:05:21 GMT

What you can also do is place a JMP at the very start of the original function to your own hook. Then when you want to call the original function you re-create the first 5 bytes you overwrote in assembly then just jmp 5 bytes into the original function.

```
function:
push ebp ; byte 1
push edi ; byte 2
push esi ; byte 3
push ebx ; byte 4
push ecx ; byte 5
push edx ; byte 6
```

Then after jumping hooking:

```
function:
jmp <hookfunc> ; byte 1-5
push edx ; byte 6
```

```
void HookFunc()
{
    blabla
}
```

```
void _declspec(naked)Call original func()
{
    _asm
    {
        push ebp ; byte 1
        push edi ; byte 2
        push esi ; byte 3
        push ebx ; byte 4
        push ecx ; byte 5
        jmp to byte 6; where 'push edx' is located
    }
}
```

Subject: Re: Function Hooking
Posted by [Neijwiert](#) on Sun, 28 Dec 2014 17:58:22 GMT
[View Forum Message](#) <> [Reply to Message](#)

iRANian wrote on Sun, 28 December 2014 07:05What you can also do is place a JMP at the very start of the original function to your own hook. Then when you want to call the original function you re-create the first 5 bytes you overwrote in assembly then just jmp 5 bytes into the original

function.

```
function:
push ebp ; byte 1
push edi ; byte 2
push esi ; byte 3
push ebx ; byte 4
push ecx ; byte 5
push edx ; byte 6
```

Then after jumping hooking:

```
function:
jmp <hookfunc> ; byte 1-5
push edx ; byte 6
```

```
void HookFunc()
{
    blabla
}
```

```
void _declspec(naked)Call original func()
{
    _asm
    {
        push ebp ; byte 1
        push edi ; byte 2
        push esi ; byte 3
        push ebx ; byte 4
        push ecx ; byte 5
        jmp to byte 6; where 'push edx' is located
    }
}
```

That was exactly what I was trying to achieve in my first attempt. Yet it somehow didn't jump to the new function, If you toggle the spoilers in the first post you can see how I tried it.

Subject: Re: Function Hooking
Posted by [iRANian](#) on Tue, 30 Dec 2014 23:39:06 GMT
[View Forum Message](#) <> [Reply to Message](#)

Yeah I saw that, but you're doing some complicated memory copying code.

Subject: Re: Function Hooking
Posted by [Neijwiert](#) on Wed, 31 Dec 2014 01:35:15 GMT
[View Forum Message](#) <> [Reply to Message](#)

iRANian wrote on Tue, 30 December 2014 16:39Yeah I saw that, but you're doing some complicated memory copying code.

How else can you execute the first 5 bytes then again if you don't store that anywhere?

Subject: Re: Function Hooking
Posted by [iRANian](#) on Sun, 11 Jan 2015 15:18:47 GMT
[View Forum Message](#) <> [Reply to Message](#)

By using a nakedspec function with asm

```
void _declspec(naked) blabla()
{
    _asm
    {
        // epilogue (first 5 bytes or so)
        push esp
        push ebp
        push edx
        push ebx
        push ecx

        jmp FunctionAddress+5 //or just the direct adress
    }
}
```
