

Forum:[Hack Forums](#)

Thread: [C Basic hooking tutorial](#)

Prock - September 4, 2017, 6:55 am

I figured my first thread/post should be at least somewhat HQ ?

What is function hooking?:

There are many different kinds of hooking but they all usually follow the same semantics. A hotpatch hook overwrites some few instructions in the code section of the binary and overwrites them with a jump to a user's own function. The execution then returns and continues to the original. This is the one we'll be doing.

There are also VMT,IAT,Segment, and registered callback hooks. The first three just involve overwriting a four byte address with your own in some table in data section or register. The last one is the common style of hooking used in programming, it is when the developer of an application or API allows for users to register their own hooks for events by passing a function pointer.

Get on with it!

Let's write our basic function that we're going to hook.

Code void PrintMessage(char* Message);That is the decleration of the function. We're going to call it with a Message, and it will print that message to the terminal.

Code void PrintMessage(char* Message)

```
{
    puts(Message);
}
```

This will be our C program.

Code #include <stdio.h>

void PrintMessage(char* Message)

```
{
    puts(Message);
}
```

int main()

```
{
    char Buffer[] = "Hello world!";
    PrintMessage(Buffer);
```

```
    fgetc(stdin); //Wait before closing
    return 0;
}
```

If we compile this straight away and run it, our PrintMessage function will become optimized and inline into the main function. For the sake of this tutorial we don't want that. Let's change the function to prevent inlining.

Code __declspec(noinline) void PrintMessage(char* Message)

```
{
    puts(Message);
}
```

If we compile and run our application, it just prints "Hello world!";

Now let's look at the disassembly of the PrintMessage function. This is necessary if we want to

hotpatch the function since we need to know what the original function looks like in memory.

Here is the result:

i.imgur.com/2eIKpSy.png

The base address of the module in file ins 401000 and the base address of the function is module base+0x1000 offset. We got the first thing we needed the address at which to place our hook.

Next we see our first instruction is push ecx which is one byte instruction. Then a call to puts. Since puts takes only one argument from the stack we can assume that main called PrintMessage by passing the buffer into ecx. There's a problem here though which realistically isn't an issue but for the sake of the tutorial it is... push ecx is only one byte while call is five bytes. That's six bytes and we need five bytes for our hotpatch. We can overwrite the total six bytes and restore them later but I'd rather not do that here. Let's go back into our code and add some useless instructions that will give us four bytes + push ecx.

```
Code __declspec(noinline) void PrintMessage(char* Message)
{
    __asm nop
    __asm nop
    __asm nop
    __asm nop
    puts(Message);
}
```

And if we look at the disassembly now:

i.imgur.com/caudQU8.png

PERFECT! We have four bytes of NOP and the one push ecx instruction which we can overwrite and restore later.

Now let's get onto writing our hook. This will remain within our same source file. We'll just be appending to it not making a new project.

First we include the Windows header.

Code #include <windows.h>

Now let's write our Hook function.

```
Code void PrintMessageHook(char* Message)
{
```

}This function will jump back to the original but passing its own message instead.

Let's continue the code from after the call to PrintMessage

```
Code int main()
```

```
{
    char Buffer[] = "Hello world!";
    PrintMessage(Buffer);
```

```
    //Hook code goes here
```

```
    fgetc(stdin);
    return 0;
```

}From the disassembly, we know the base address of the function is Module Base+0x1000.
To get Module Base we can just call GetModuleHandle.

```
Code  PBYTE Base = (PBYTE)GetModuleHandleW(0);
```

Base += 0x1000;Next, since we are patching the code section, we need to change page protection so we don't cause a crash.

```
Code  DWORD OldProtection;
```

```
VirtualProtect(Base, 1, PAGE_EXECUTE_READWRITE, &OldProtection);
```

Overwrite the first byte with a jump (0xE9)

```
Code  *Base = 0xE9;
```

Overwrite the next four bytes with the offset to our hook function. Offset is calculated by destination - source - 5 because the EIP will be five bytes ahead of the current jump instruction.

```
Code  *(DWORD*)(Base + 1) = (DWORD)PrintMessageHook - (DWORD)PrintMessage - 5;
```

Almost there!

Inside our PrintMessageHook we need a way to return to the original function to continue execution. We also need to remember we overwrote that crucial push ecx instruction along with the useless four nop instructions. So we need to restore push ecx and return to the original function+5 to jump past our jump. Or we'd be stuck in an infinite loop. This is called the trampoline:

```
Code __declspec(naked) void PrintMessageTrampoline(char* Message)
```

We declared this naked so we need to write the function in assembly.

```
Code __declspec(naked) void PrintMessageTrampoline(char* Message)
```

```
{
    __asm
    {
        push ecx //original instruction
        mov ecx,PrintMessageAddress
        add ecx,5
        jmp ecx
    }
}
```

}So we first do push ecx because we overwrote that then we move the address of PrintMessage into ecx and add 5 as I explained earlier...but We didn't declare PrintMessageAddress anywhere so we can do that now.

Above PrintMessageTrampoline, place this code

```
Code DWORD PrintMessageAddress = 0;
```

Then inside our main function, after Base+=0x1000; do :

```
Code  PrintMessageAddress = (DWORD)Base;
```

Finally! Let's call our Trampoline from inside our Hook function with our own message instead of "Hello World!"

```
Code void PrintMessageHook(char* Message)
```

```
{
    PrintMessageTrampoline("LOL HOOKED!");
```

}I think we're done! But first let's open this all up in a disassembler/debug to see if everything is correct!

....

.....

And it's not..After all the changes we made to the code it seems the compiler changed up the function that we looked at earlier. So now PrintMessage ends up looking like this:

image.prntscr.com/image/uTkBpHZ2RXGPVeEK12gB1A.png

We have to go back and adjust our trampoline.

The first three bytes are now

```

push ebp
mov ebp,esp
while rest nop
and ecx isn't even part of the function anymore.
Code __declspec(naked) void PrintMessageTrampoline(char* Message)
{
    __asm
    {
        push ebp
        mov ebp,esp
        mov ecx,PrintMessageAddress
        add ecx,5
        jmp ecx
    }
}

```

And now! To prove that the hook worked let's call PrintMessage again after we do all our hook work.

```

Code    char Buffer[] = "Hello world!";
        PrintMessage(Buffer);

//Hook code goes here
PBYTE Base = (PBYTE)GetModuleHandleW(0);
Base += 0x1000;
PrintMessageAddress = (DWORD)Base;
DWORD OldProtection;
VirtualProtect(Base, 1, PAGE_EXECUTE_READWRITE, &OldProtection);
*Base = 0xE9;
*(DWORD*)(Base + 1) = (DWORD)PrintMessageHook - (DWORD)PrintMessage - 5;

```

```

        PrintMessage(Buffer);
And our result :

```

image.prntscr.com/image/lhJES3XFTECisR451INdOg.png

:D!

Thanks for reading! Hope you enjoyed and learnt something!

eoaikyn - September 9, 2017, 10:16 am

Good tutorial! saving this one for later :D