MHOOK, AN API HOOKING LIBRARY, V2.2

From: http://codefromthe70s.org/mhook23.aspx

*New version with built-in disassembler and bugfixes. The old, lightweight version is still available **here**. The previous version (2.2), should you need it for some reason, is **here**.*

Mhook is a library for installing API hooks. If you dabble in this area then you'll already know that Microsoft Research's **Detours** pretty much sets the benchmark when it comes to API hooking. Why don't we get a comparison out of the way quickly then?

DETOURS VS. MHOOK

Detours is available for free with a noncommercial license but it only supports the x86 platform. Detours can also be licensed for commercial use which also gives you full x64 support, but you only get to see the licensing conditions after signing an NDA.

Mhook is freely distributed under an MIT license with support for x86 and x64.

Detours shies away from officially supporting the attachment of hooks to a running application. Of course, you are free to do it - but if you end up causing a random crash here or there, you can only blame yourself.

Mhook was meant to be able to set and remove hooks in running applications – after all, that's what you need it for in the real world. It does its best to avoid overwriting code that might be under execution by another thread.

Detours supports transactional hooking and unhooking; that is, setting a bunch of hooks at the same time with an all-or-nothing approach. Hooks will only be set if all of them can be set, otherwise the library will roll back any changes made. Mhook does not do this.

Finally, Mhook is pretty lazy when it comes to managing memory for the trampolines it uses. Detours allocates blocks of memory as needed, and uses the resulting data area to store as many trampolines within as will fit. Mhook, on the other hand, uses one call to VirtualAlloc per hook being set. Every hook needs less than 100 bytes of storage so this is very wasteful, since VirtualAlloc ends up grabbing 64K from the process' virtual address space every time Mhook calls it. (Actual allocated memory will be a single page which is also quite wasteful.) In the end though, this probably does not really matter, unless you are setting a very large number of hooks in an application. Also, this is very easy to fix.

With that out of the way, if you're still here, let's delve into it.

EXAMPLE: HOOKING NTOPENPROCESS

It might be best to start off with a short code example that shows the library in action. The snippet below hooks NtOpenProcess. The code has been tested on x86 and x64 versions of XP and Vista.

```
//==========================================================================
#include "stdafx.h"
#include "mhook.h"

//==========================================================================
// Define _NtOpenProcess so we can dynamically bind to the function
//
typedef ULONG (WINAPI* _NtOpenProcess)(OUT PHANDLE ProcessHandle,
    IN ACCESS_MASK AccessMask, IN PVOID ObjectAttributes,
    IN PCLIENT_ID ClientId );


//==========================================================================
// Get the current (original) address to the function to be hooked
//
_NtOpenProcess TrueNtOpenProcess = (_NtOpenProcess)
    GetProcAddress(GetModuleHandle(L"ntdll"), "NtOpenProcess");

//==========================================================================
// This is the function that will replace NtOpenProcess once the hook
// is in place
//
ULONG    WINAPI MyNtOpenProcess(OUT PHANDLE ProcessHandle,
                    IN ACCESS_MASK AccessMask,
                    IN PVOID ObjectAttributes,
                    IN PCLIENT_ID ClientId )

{
    // do any processing here if needed
    // ...
    // punt the call to the the OS in the end
    return TrueNtOpenProcess(ProcessHandle, AccessMask,
        ObjectAttributes, ClientId);

}

//==========================================================================
// This is how you go about putting the hook in place. When you're done,
```

```
// any calls to NtOpenProcess will be redirected to MyNtOpenProcess.

// If you need to access the original, unmodified API, call
// TrueNtOpenProcess from your code, just like the hook function above does.
//
BOOL WINAPI SetHooksAndDoWork () {

    BOOL bHook = Mhook_SetHook((PVOID*)&TrueNtOpenProcess,
            MyNtOpenProcess));

    // Minimalist error handling
    if (!bHook) return FALSE;

    // ... any calls to NtOpenProcess within this process are now

    // rerouted to MyNtOpenProcess.

    // For example, this call will end up in our hook function since
    // kernel32!OpenProcess just calls ntdll!NtOpenProcess internally
    HANDLE hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE,

    GetCurrentProcessId());

    // ...

    // This call will bypass the hook:
    // (parameter initialization omitted for brevity)
    TrueNtOpenProcess(&hProc, &accessMask, &objAttrs, &clientId);

    // ...

    // Remove the hook when we're done.
    return Mhook_Unhook((PVOID*)&TrueNtOpenProcess);

}

//=========================================================================
```

OVERVIEW

As you can see the library is pretty easy to use. It gets only slightly more complicated under the hood.

When hooking a Windows API function you determine the location of the function, change the

page protection so the memory can be written to, modify the function so it jumps to your own code rather than doing its own thing (other processes that have this DLL loaded will be unaffected due to the OS's copy-on-write mechanism) – and you're done.

CODE INJECTION

Mhook will not take care of code injection for you, and assumes that your code is already running in the target process. This is fairly easy to achieve. If you need more information on this topic, **this is a good place to start**.
Normally you'd do this from a DLL that you inject into the target process via any of the well-established methods: the AppInitDLLs registry key, using CreateRemoteThread, or even SetWindowsHookEx . When the DLL is loaded, you set your API hook in DLL_PROCESS_ATTACH, and remove it in DLL_PROCESS_DETACH.

Of course you will need to do a bit of housekeeping, the most important aspect of which is ensuring that the original API will be available if and when you need it: for one, your replacement function will probably have to be able to access the original API.

So how exactly do we tie up the loose ends?

When a hook is set, the library allocates a chunk of memory that will contain the trampolines. (No, this isn't a term Microsoft came up with: see **http://en.wikipedia.org/wiki/Trampoline_(computers)**.) The trampolines will need to be within +/- 2GB of the target API because the hooked version of the function will begin with a 32-bit relative jump once the hook is in place. A 32-bit relative jump takes up 5 bytes on both x86 and x64. This is the smallest JMP instruction useful for our purposes, and it's important to keep these JMPs small because space is sometimes at a premium in target APIs.

Fortunately the allocation process is easy, thanks to VirtualQuery and VirtualAlloc. You start enumerating memory blocks with VirtualQuery at (TargetFunctionAddr – 2GB), and loop until you find a free block. Once a free block has been found, you try to allocate some memory for your trampolines. If you succeed you're done with this step – if you fail you simply keep trying until a block is successfully allocated, or your base address goes beyond (TargetFunctionAddr + 2GB), at which point you must give up.

ABSOLUTE JMP SIZES

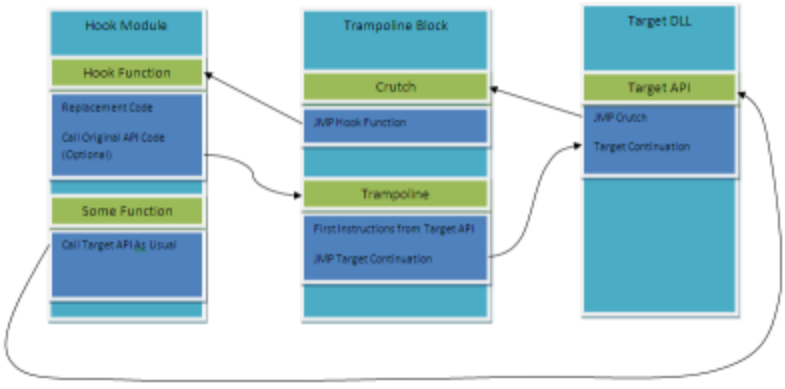
An absolute JMP on x86 requires 9 bytes: one for the E9 opcode, four for the absolute address of the memory location that stores the target address, and four for the aforementioned memory location with the target address itself. The same instruction requires 13 bytes on x64: one byte for the E9 opcode, four for a relative offset pointing to the memory location of the absolute address, and 8 bytes at the aforementioned location that point at the target location itself. You could do this with shorter instructions if you didn't care about preserving register contents, but

that's not an option in our case.
Once we have a suitable chunk of memory, we create two trampolines in it. The first one will contain the first few machine-code instructions from the target API (these will be overwritten by the JMP instruction in the original location so we must duplicate them somewhere) followed by a jump to the rest of the API back in its original location. Calling this trampoline is functionally identical to calling the original API before we hooked it – the address of the trampoline can therefore be used as a secret entry point for the original, unmodified functionality of the API. The other trampoline is just a crutch: the JMP instruction at the beginning of the hooked API will point here. The crutch will contain another JMP instruction that points at our replacement function. The reason for not JMPing directly from the original API to the replacement function is that we cannot be certain that the replacement function (which will most likely just reside in our DLL as a C/C++ routine) is within +/- 2GB of the original function – but the crutch is guaranteed to be there. We therefore insert a 5-byte relative JMP instruction in the original function and have room for an absolute JMP in the trampoline, thus eliminating the problem of our replacement code potentially being further than 2GB from the target API.

PUTTING IT TOGETHER

Once the trampolines have been set up with the proper instructions, we simply overwrite the target function's first five bytes with a JMP to our replacement function (well, a JMP to the trampoline that jumps to the replacement function to be precise) and we're done: the API is hooked. This step requires extra care however: aside from using VirtualProtect to enable writing to the target page (PAGE_EXECUTE_READWRITE) and then resetting the page protection to its original value we must also FlushInstructionCache after we're done. In order to protect other threads in the application properly and ensure that we're not overwriting a memory location where another thread is currently executing we also suspend all other threads before placing our JMP into the target function, and resume them afterwards. We go even further and compare each suspended thread's instruction pointer against the target API. If the IP in the range where the JMP function will be placed (basically, the first 5 bytes of the API) then we resume the thread, let the IP advance, and suspend the thread again.

The following diagram is meant to illustrate a hooked function, the replacement function, the trampolines, and their relationship (the arrows indicate transfer of control):



Mhook in action

(click to enlarge)

USING THE LIBRARY

Mhook is quite simple to use. There are two functions you'll need to call:
BOOL Mhook_SetHook (PVOID *ppSystemFunction, PVOID pHookFunction);

Mhook_SetHook will set a hook in a specified API.
PVOID* ppSystemFunction

The first parameter to Mhook_SetHook is used for both input and output: it needs to point to a variable that stores the address of the function to be hooked. Upon successful return of the function this variable will contain the address of the trampoline that provides access to the original functionality of the hooked API.
PVOID pHookFunction

The second parameter to Mhook_SetHook is a pointer to the replacement function.

Return value: nonzero if successful, zero if an error occurrs.
BOOL Mhook_Unhook (PVOID *ppHookedFunction);

Mhook_Unhook will remove a previously set hook.
PVOID* ppHookedFunction

The one and only parameter to Mhook_Unhook is used for both input and output. It needs to point to the same location the first parameter of Mhook_SetHook pointed to. When the function returns with success, it will also overwrite the contents of the memory location addressed by this parameter, and instead of pointing at the trampoline, the memory will once again point to the original API.

Return value: nonzero if successful, zero if an error occurrs.
Take a look at the sample code at the beginning of this article to see these calls in action.

There is no hard limit on the number of hooks that can be used simultaneously within a single process, other than the fact that each hook will reserve 64 kilobytes of VM space.
SUMMARY

If you can live with the limitations outlined at the beginning of this article then Mhook can be a very useful tool.

Mhook is distributed under the MIT license. As usual, no warranties are implied or expressly granted.

You can download the source code from here: **mhook-2.3.zip**

ACKNOWLEDGEMENTS

Mhook contains a disassembler that is a stripped-down version of the **excellent tDisasm package** by Matt Conover. Thank you Matt! tDisasm comes with a BSD-style license and re-releasig a derivative of it under the MIT license has been confirmed to be OK by its author.

Alexandr Filenkov submitted bugfixes in Sept-2007. Michael Syrovatsky submitted fixes for IP-relative addressing in Jun-2008. Andrey Kubyshev submitted a bugfix in Jul-2011.
CHANGELOG

- 2012-01-15: Version 2.3 with a bugfix that allows hooking more API functions
- 2008-06-27: Version 2.2 with support for instructions using IP-relative addressing
- 2007-10-15: Version 2.1 with fixes
- 2007-07-08: Version 2.0 with built-in disassembler
- 2007-06-24: Original Release

FUTURE IMPROVEMENTS

Mhook is far from perfect. The following things should be addressed in the future:

- Implement a memory allocator so one call to VirtualAlloc can service multiple hooks

- Improve the thread-suspension code so it can deal with threads that are spawned during the execution of the thread-suspension process itself
- Improve error handling so meaningful failure codes can be retrieved by GetLastError
- For the truly paranoid: deal with possible conflicts with other hooking libraries (what if Mhook_SetHook is called on a function that is currently hooked with Detours, etc)
- Add support for IA64 (Itanium)