

Forum:[Hack Forums](#)

Thread: [Detecting API hooking \(Run-Time byte patching - C++\)](#)

AmericanExpress - June 26, 2017, 12:24 pm

Hello! ?

I've been planning to write a tutorial on different methods of detecting present hooks and decided to kick-start it with some basic detection methods of run-time byte patching. Due to the amount of different methods of hooking (it can be done differently all the time, even for function patching in memory) the code within this thread may not necessarily always work therefore it isn't 100% reliable, it depends how the hook was made.

The detection method I will demonstrate within this thread should work if the hook has been performed either by:

- altering the first 5 bytes of the function with a JMP instruction (for 32-bit processes).
- using the MOV RAX and then JMP RAX method (for 64-bit processes).

For testing purposes I used the hook code I posted in the Coding area the other day, the thread can be found here: <https://hackforums.net/showthread.php?tid=5651733>

Before I get into the tutorial I also want to note that the focus right now is for NTAPI functions exported by ntdll.dll. I also want to note that the function prologues are not always identical depending on what OS version you are on and patch updates, although the process method within the prologue will pretty much be the same for the newer versions of Windows. Just not identical. For example, the system call IDs, and on X86-X64 it has to pass through Wow64. I recommend you study hooking and learn how it all works before moving on to hooking identification since you might get confused.

To start off, we will look at the 32-bit function prologues for 3 NTAPI functions: NtTerminateProcess; NtOpenProcess; and NtSuspendProcess.

NtTerminateProcess:

```
Code B8 2C 00 07 00 ; mov     eax,7002Ch
BA F0 2C 74 77 ; mov     edx,77742CF0h
FF D2          ; call     edx
C2 08 00          ; ret     8
```

NtOpenProcess:

```
Code B8 26 00 00 00 ; mov     eax,26h
BA F0 2C 74 77 ; mov     edx,77742CF0h
FF D2          ; call     edx
C2 10 00          ; ret     10h
```

NtSuspendProcess:

```
Code B8 A8 01 03 00 ; mov     eax,301A8h
BA F0 2C 74 77 ; mov     edx,77742CF0h
FF D2          ; call     edx
C2 04 00          ; ret     4
```

As we can see from the above, the process for 32-bit processes is usually more or less the same.

The function ID for the system call is moved into the EAX register, then the address for passing through Wow64 is moved into the EDX register, then the function will call edx so code execution moves through Wow64 (since that is where the address in the register is leading to) and then we return 4. Wow64 layer will handle the system call, so the value within EAX will end up in the kernel so the kernel knows which function was being called.

On a 32-bit OS version of Windows it will be different, however since I am on a 64-bit OS all my 32-bit processes will pass through Wow64. If you are interesting in checking how it happens on a 32-bit OS version then use a debugger and go to the address of an NTAPI function.

On older versions of Windows it does actually pass through layers like KiFastSystemCall. If you're on Windows 10 that doesn't happen though, neither on Windows 8 I don't think. Now let's look at the x64 process function prologues.

NtTerminateProcess:

```
Code 4C 8B D1      mov     r10,rcx
B8 2C 00 00 00    mov     eax,2Ch
F6 04 25 08 03 FE 7F 01 test    byte ptr [7FFE0308h],1
75 03            jne     00007FFA91166655
0F 05            syscall
C3              ret
```

NtOpenProcess:

```
Code 4C 8B D1      mov     r10,rcx
B8 26 00 00 00    mov     eax,26h
F6 04 25 08 03 FE 7F 01 test    byte ptr [7FFE0308h],1
75 03            jne     00007FFA91166595
0F 05            syscall
C3              ret
```

NtSuspendProcess:

```
Code 4C 8B D1      mov     r10,rcx
B8 A8 01 00 00    mov     eax,1A8h
F6 04 25 08 03 FE 7F 01 test    byte ptr [7FFE0308h],1
75 03            jne     00007FFA911695C5
0F 05            syscall
C3              ret
```

As we can see from the above function prologues for the 64-bit NTAPI functions, all of them have some things in common although they are not 100% identical. The main similarities would be that all of them start with MOV R10, RCX (0x4C, 0x8B, 0xD1); afterwards they move the function ID into the EAX register like we are used to. The system call is performed at the end of each function prologue and then it uses ret to return.

If we go back to the X86-X64 function prologues, we can see that all of them have similarities too... They all start with MOV EAX, ... <addr>!

This means that if a hook on our function (via run-time byte patching) has overwritten the first few instructions to place a JMP instruction to another address or made use of the MOV RAX and then JMP RAX method, we can identify these two methods just by comparing the first bytes at the address of our NTAPI function. I will now show you C++ code on how to do this.

I will start by getting the addresses of the target functions I wish to scan:

```
Code LPVOID lpTarget1 = (LPVOID)GetProcAddress(LoadLibraryA("ntdll.dll"),
"NtTerminateProcess");
```

```
LPVOID lpTarget2 = (LPVOID)GetProcAddress(LoadLibraryA("ntdll.dll"), "NtOpenProcess");
LPVOID lpTarget3 = (LPVOID)GetProcAddress(LoadLibraryA("ntdll.dll"), "NtSuspendProcess");
// you can always put it in an array of LPVOID but it is down to you really
```

Now we have the addresses of the function/s, we can now start the byte comparison. I have written a function in C++ which takes in two parameters (the first being the function address, and the second being a bool to represent whether the process we are scanning is 32-bit or 64-bit... It doesn't matter if it is an X86-X64 process, if the process is 64-bit then you pass TRUE and if not you use FALSE for the second parameter). I will post the function below and then explain how it works:

```
Code BOOL detect_hook(LPVOID lpFunctionAddress, bool b64)
```

```
{
    if (!b64) // x86
    {
        if (*(BYTE*)lpFunctionAddress != 0xB8)
        {
            return TRUE;
        }
    }
    else // x64
    {
        if (*(BYTE*)lpFunctionAddress != 0x4C && (*(BYTE*)lpFunctionAddress + 1) != 0x8B &&
            (*(BYTE*)lpFunctionAddress + 2) != 0xD1)
        {
            return TRUE;
        }
    }
    return FALSE;
}
```

If the function is told that the program is not running as x64 compiled then it will compare the first byte of the function to check if it is the same as MOV EAX (which is 0xB8 in byte representation). However, since we know that all (if not then most, the ones that will perform the system call at least or pass through Wow64, since some NTAPI functions don't even do that they just redirect to other stubs, an example would be RtlCreateUserThread which lands at NtCreateThreadEx I believe) will start with 0xB8 it means it will be quite accurate. If the function has been hooked with a normal run-time byte patch through placing a JMP instruction it will potentially look like the following:

```
Code E9 00 29 3B 89    jmp      NtTerminateProcessCallback (0AE1195h)
BA F0 2C 74 77    mov     edx,77742CF0h
FF D2            call    edx
C2 08 00          ret     8
```

As we can see from the above function prologue which has been hot patched, the first instruction is 0xE9 (JMP) instead of 0xB8 (MOV EAX).

Going back to what I was speaking about earlier, if the function is given TRUE as the second parameter then it will assume the process is 64-bit compiled and will then attempt to detect if the start of the function prologue is correct for x64. As we know from earlier, the similarities would be MOV R10, RCX therefore we scan the first few bytes to detect if that exists or not, and if not we return TRUE to represent the function prologue is not correct.

Lets take a look at an x64 process which has been hooked using the MOV RAX, addr and then JMP RAX method:

```
Code 48 B8 C6 12 A3 C6 F7 7F 00 00 mov     rax,7FF7C6A312C6h
FF E0            jmp     rax
03 FE            add     edi,esi
```

```
7F 01      jg      00007FFA91166651
75 03      jne     00007FFA91166655
0F 05      syscall
C3         ret
```

As we can see the start is 0x48 0xB8 (MOV RAX) instead of MOV R10, RCX... So we report with returning TRUE to signify the function prologue has been modified.

Like I have already said, it depends on the circumstances because this detection will not always be accurate or work. For example, some NTAPI functions do not follow the same similarities... Plus there are different hooking techniques, even for run-time byte patching. However this is just the start to a basic identification method for run-time byte patching for beginners who would like to get into learning about the different API hooking methods with some practical code.

In a future tutorial I will demonstrate a bit on repairing the detected hooks. Apologies for this being basic and short, but hopefully will still be found educational to some!

Good luck! ?

Don Genovese - June 26, 2017, 12:30 pm

Very nice thread mate, great content. Thanks for the information. You post a lot of HQ shit, keep it up ?

AmericanExpress - June 26, 2017, 12:33 pm

(Jun 26, 2017 12:30 PM)Wolf™ Wrote: [\[CITE\]](#) Very nice thread mate, great content. Thanks for the information. You post a lot of HQ shit, keep it up ?Thank you! I'm trying to contribute to help teach people who are stuck/interested, we all have to start from somewhere! ?

HFDone - June 27, 2017, 12:21 am

you can check dword (4C8BD1B8) for the 64 bit

AmericanExpress - June 27, 2017, 5:27 pm

(Jun 27, 2017 12:21 AM)HFDone Wrote: [\[CITE\]](#) you can check dword (4C8BD1B8) for the 64 bitThat is a good idea IMO. ?

AmericanExpress - June 28, 2017, 10:21 am

(Jun 28, 2017 09:06 AM)Friedrich Nietzsche Wrote: [\[CITE\]](#) What if the hook use other methods for redirecting the call such as:

Code mov eax, hookaddr
jmp eaxor other methods? Your method of checking is not reliable, and while it is a good start it could be greatly improved. A possible method would be to compare it with the file on disk (since it is rare to see people hooking NtReadFile to prevent hooking detection, I personally have never encountered it) or even better just map your own copy of ntdll and call the functions from there (which would bypass all Ring3 hooks).You could always check for the JMP EAX which shouldn't be there since EAX should contain the system call ID for the calling function.

My method was not supposed to be reliable for a copy and paste, I just wanted to make an educational starter thread to it all. I'd like people who read it to use the info within the thread to get a better understanding about it, and then use their own skills/initiative to do something even better themselves. As opposed to a really good hook scanner and repair ready for copy paste similar to what I've done in other threads like the x86/x64 hook code post.

By the way, your idea about reading ntdll.dll and using that is a good idea and I was considering posting a thread on it the other day. I probably still will, but first I'll do a post about direct NTAPI system calls first. However, if the process is X86-X64 the best thing to do would be Heavens Gate to perform direct x64 system calls since Wow64 can be intercepted. ?

Thanks for reading the thread and posting and I do agree with everything you've said. ?