## Windows may have no backdoors but its documentation has some!

**Baptiste DAVID (C + V)$^{O}$ Lab**

When I was young, my grandmother always told me that without the right tools, we could not succeed... This is perfectly true. To be honest with you, I'm not a great handyman. And when I want to drive a nail into a wall, in general, I read the documentation which advices to use a hammer and hit on the nail. It's what I do with more or less success… With a hammer, it is easy and fast to reach on the goal. But what happens if the documentation advises you to use a Stradivarius violin to do so?!!

To illustrate the point, let's talk about memory protection of a process. Why such a topic? Usually, since a long time, almost everything has been already written about. Yes, it's pretty right, when we where at school, we learn everything about shared memory, who can write and who can read… If you have spent few hours in security courses, you have seen the same case with writing and execution[1]. But the story of memory protection, in truth, doesn't end there…

To sum things up, we saw read xor write (R^W) and write xor executable (W^X) when we were novice programmers [1] [2], but it still misses to us read xor executable (R^X). And this is where our adventure begins… Once upon a time…

What does mean R^X? It just means that the op-codes you run are not readable. The main objective behind that, is about to make a sort of big black box in which he could write our secret instructions away from prying eyes. In short, the op-codes of your programs (and by extension their features) that you could write would be "confidential in memory". Formally, it's an interesting feature to protect your code from any oafish competitors…

To do so, no choice, we need to interact with the system here because it's the OS which manage the memory in the computer... Before all, we just have to solve one question: Windows or Linux? We threw a coin in the air, stack, and let's choose windows (the coin is simulated by Windows random API).

On msdn, two major functions are available for us: VirtualAlloc[Ex] [3] and VirtualProtect[Ex] [4]. The first is used to allocate a buffer with the rights we want, the second to change the rights of an allocated buffer. The main argument that interests us is the fourth to the first function and the third for the second function. This argument is called Memory Protection Constants [5]. Specifically, it allows you to choose what type of rights you want to give to a memory section (ie: +rwx). No need to read very far to find our solution, the first is the good.

---

[1] The last point of writing and execution is about to prevent the case of polymorphic code in memory when we execute and write op-codes in the same buffer (note that this type of protection is useless because it is possible to do this kind of thing sequentially).

| Constant/value | Description |
| --- | --- |
| **PAGE_EXECUTE**<br>0x10 | Enables execute access to the committed region of pages. An attempt to read from or write to the committed region results in an access violation.<br>This flag is not supported by the **CreateFileMapping** function. |

Without being myself a native English speaker, I understand that it means when the memory is set to this mode, it is only executable. Wonderful, it's what we want! A small and smart implementation should quickly bring us what we want. I'm old school and I did it in x86 assembly (it can also be done in C easily). I will explain it progressively to make this as clear as possible.

```
;~ Allocate a buffer to store our own op-codes.
invoke VirtualAlloc, NULL, 15, MEM_COMMIT, PAGE_READWRITE
mov dwMemOp, eax
.if (eax == 0)
    invoke MessageBox, NULL, addr msgError, addr msgTitle, MB_OK
    xor eax, eax
    ret
.endif
```

The first thing to do is to allocate a buffer which will be able to receive our secret op-codes. Nothing complicated here, we just call the function we saw earlier. The rights for the buffer are *a priori* read and write (not executable at all).

Then we write the secrets op-codes as expected.

```
lea ebx, OpCodes            ;~ The adress of the buffer in the .data
                            ;~ section where the op-codes are stored.
mov ecx, dwSizeOpcodes      ;~ Number of op-codes to write.

;~ Fill the buffer allocated with the op-codes.
@@:
    movzx edx, BYTE ptr [ebx]
    mov BYTE ptr [eax], dl
    inc eax
    inc ebx
loop @B
```

Of course, if my op-codes are stored in clear text in my data section (as it's the case here), there is no secret, but we can well imagine that they come from an encrypted file or any other source. In addition, the goal here is to make a proof of concept, nothing more. The op-codes written here do not do anything special; it's just a function which returns 1. I say it because we will reuse this fact latter. The op-codes are as follows:

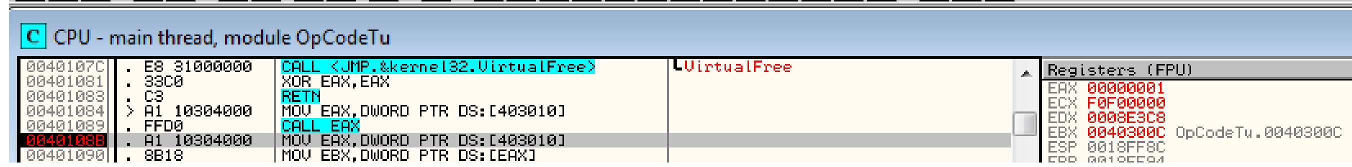
```
.data
OpCodes         BYTE        055h                            ;~ push ebp
                BYTE        08bh, 0ech                      ;~ mov ebp, esp
                BYTE        0b8h, 001h, 00h, 00h, 00h       ;~ mov eax, 1
                BYTE        08bh, 0e5h                      ;~ mov esp, ebp
                BYTE        05dh                            ;~ pop ebp
                BYTE        0c3h                            ;~ ret
```

Well, now that everything has been done, we just need to change the rights of our buffer to make it only executable and run it. It is relatively simple to do with the VirtualProtect function.

```
;~ Change the rights to PAGE_EXECUTE for the buffer with the op-codes
;~ (normally they are not supposed to be readable).
invoke VirtualProtect, eax, 15, PAGE_EXECUTE, addr OldProtect
.if (eax == 0)
    invoke MessageBox, NULL, addr msgErrorBis, addr msgTitle, MB_OK
    invoke VirtualFree, dwMemOp, 15, MEM_DECOMMIT
    xor eax, eax
    ret
.endif

mov eax, dwMemOp    ;~ In order to call, we move the adress of the buffer into eax.
call eax            ;~ Let's make the call (let's check if it's really executable).
```

The call to the buffer worked well and the value returned by the function is 1 (proof that the function has been well executed, that’s why we explained our op-code function before).




As we have started to check if everything worked well, we can verify that our buffer is not writable. It's very simple; we just need to try to replace the op-codes in the buffer by some others without affecting the rights of the buffer.

```
;~ Let's try to write on it...
mov DWORD ptr [eax], 90909090h ;~ Normaly, impossible due to the restrictions.
```

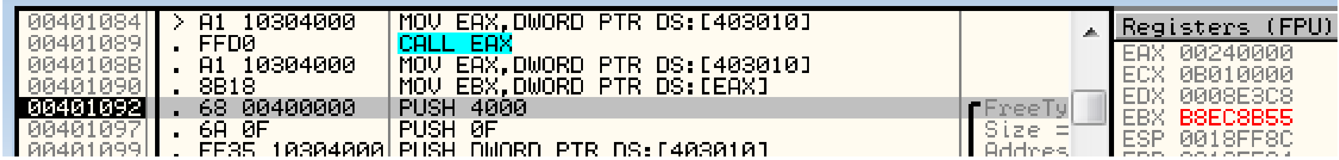
The result expected is – of course – the one we have: *Access violation when writing*… Normal, it’s written on the msdn that it’s supposed to result in an access violation with the PAGE_EXECUTE memory constant protection.

```
00401092 Access violation when writing to [003E0000]
00401092 Access violation when writing to [003E0000]
00401092 Access violation when writing to [003E0000]
00401090 . 8B18 MOV EBX,DWORD PTR DS:[EAX]
00401092 . C700 90909090 MOV DWORD PTR DS:[EAX],90909090
00401098 . 68 00400000 PUSH 4000 FreeType = MEM_DECOMMIT
```

We could take for granted that the buffer is not readable. Nevertheless, to be definitely sure, we checked this option. The code to do so is obvious; we just need to try to store in a register the op-codes stored in the buffer (without changing the rights of the buffer again). Here, we try to store the first op-codes of the buffer in ebx.

```
;~ Let's try to read...
mov eax, dwMemOp        ;~ We store the adress of the buffer into eax.
mov ebx, [eax]          ;~ Should be an access violation!
```

*Close tour eyes, you will get a surprise*… And what a surprise we’ve got!




The code works even if it's not supposed to work! Indeed, there is no access violation, at the opposite of what we had with the attempt to write into the buffer. Here, the op-codes are all stored in ebx, which proves that the protection has completely failed[2]. The main question is about to know why we can read the data stored in the buffer without any error.

A first answer, I supposed, was because I am on 64-bit architecture and because I wrote a code in assembly x86, the 32-bit emulation of Wow64 could have a problem. Okay, why not, I redid the code in x64.

[2] it seems that the op-codes are stored upside down, it is normal, the memory is managed in little endian.

```
mov r9, 04h                    ;~ PAGE_READWRITE
mov r8, 00001000h              ;~ MEM_COMMIT
mov rdx, 20                    ;~ Size of the buffer allocated.
mov rcx, 0                     ;~ No special preferences.
call VirtualAlloc              ;~ Allocate the memory for the buffer (in +rw).
mov qwMemOp, rax

or rax, rax                    ;~ .if (rax == 0)
je __main_end                  ;~      If error, go to end.
                               ;~ .endif

lea rbx, OpCodes               ;~ The address of the buffer in the data section where the op-codes are stored.
mov rcx, qwSizeOpcodes         ;~ Counter for the loop to know how much op-codes to write in the buffer.

@@:
    mov dl, BYTE ptr [rbx]     ;~ Get the op-code to write.
    mov BYTE ptr [rax], dl     ;~ Write it in the buffer allocated previously.
    inc rax
    inc rbx
loop @B                        ;~ Write all the op-codes.

mov rcx, qwMemOp               ;~ The base address of the buffer allocated with the op-codes.

lea r9, OldProtect             ;~ Offset to get the old value for protection.
mov r8, 10h                    ;~ PAGE_EXECUTE
mov rdx, 20                    ;~ Size of the buffer allocated.
call VirtualProtect            ;~ Make the buffer executable (and only executable, which means (+x)).

or rax, rax                    ;~ .if (rax == 0)
je __main_free_memory          ;~      If error, go to free memory and then end.
                               ;~ .endif

mov rax, qwMemOp               ;~ Ready to call the buffer of op-codes.
call rax                       ;~ Let's make the call (let's check if it's really executable).

;~ Let's try to read...
mov rax, qwMemOp               ;~ Load in rax register the current address of the buffer.
mov rbx, [rax]                 ;~ Should be an access violation, but it's not, why ?!!

; mov DWORD ptr [rax], 0       ;~ Impossible due to the restriction, ok, it's normal.

__main_free_memory:
mov r8, 4000h                  ; MEM_DECOMMIT
mov rdx, 20                    ;~ Size of the buffer allocated.
mov rcx, qwMemOp               ;~ The address of the buffer to desallocate.
call VirtualFree               ;~ Free the buffer allocated at the beginning.

__main_end:
call ExitProcess               ;~ End of process.
```

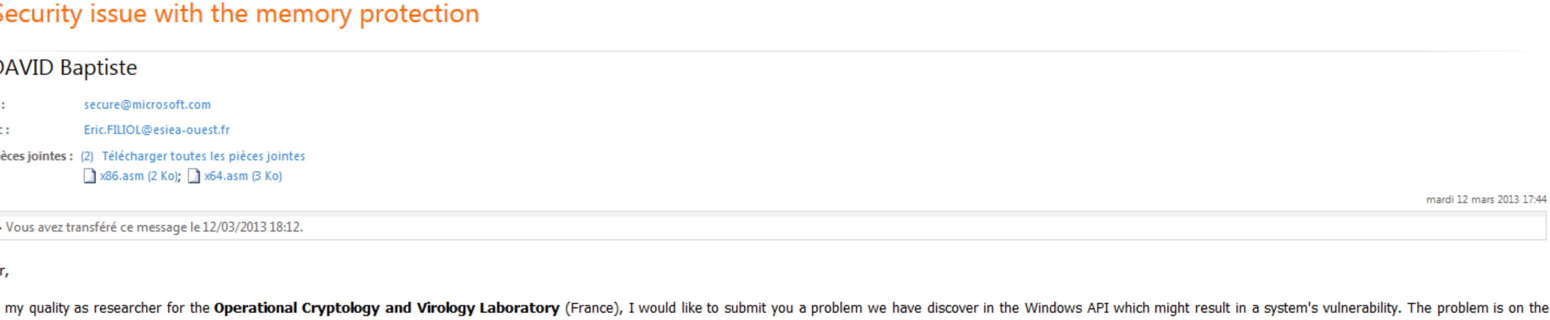
Without suspense, the result is exactly the same than the x86 code. We cannot write but we can read and execute... The problem is not in our implementation, but - for us and at that point - it comes from Windows. Two possibilities: The first is about the OS itself which is bugged and what we have found is a confidential issue for all the codes which use this feature proposed by Windows all over the world. The second possibility is about an *"error"* which is narrated in the msdn (and we could wonder about how the documentation on the msdn is made)…

If the first possibility is not from our responsibility but the second can quickly have a solution. It suffices to continue the reading of the msdn about Memory Protection Constants. The constant just after the one we tested is:

| Constant/value | Description |
|---|---|
| **PAGE_EXECUTE**<br>0x10 | Enables execute access to the committed region of pages. An attempt to read from or write to the committed region results in an access violation.<br>This flag is not supported by the **CreateFileMapping** function. |
| **PAGE_EXECUTE_READ**<br>0x20 | Enables execute or read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation.<br>**Windows Server 2003 and Windows XP:** This attribute is not supported by the **CreateFileMapping** function until Windows XP with SP2 and Windows Server 2003 with SP1. |

Today's question is: What is the difference between PAGE_EXECUTE and PAGE_EXECUTE_READ? If we understood that PAGE_EXECUTE_READ gave to the memory read-only and execute rights, but it seems that PAGE_EXECUTE is identical... So, if it's confusion on the msdn - we could believe it - but why does Microsoft have two different constants which share the same behavior?

For an answer, we sent an email with a report similar to this one to the security support of Microsoft. This one is displayed as fallow.



Répondre Répondre à tous Transférer

Security issue with the memory protection

DAVID Baptiste

À : secure@microsoft.com
Cc : Eric.FILIOL@esiea-ouest.fr
Pièces jointes : (2) Télécharger toutes les pièces jointes
x86.asm (2 Ko); x64.asm (3 Ko)

mardi 12 mars 2013 17:44

- Vous avez transféré ce message le 12/03/2013 18:12.

Sir,

In my quality as researcher for the **Operational Cryptology and Virology Laboratory** (France), I would like to submit you a problem we have discover in the Windows API which might result in a system's vulnerability. The problem is on the versions of Windows 7 and Windows 8 (x64). These are the versions we tested, but it is possible that others versions are affected.

The problem is about the rights to the process memory. When we allocate memory, we can choose the memory protection (with VirtualAlloc it's the third argument). At that point, there is no problem and we chose to allocate the buffer with the rights of reading and writing (PAGE_READWRITE). Then, we can write, in the buffer allocated, some opcodes as data. Now we can change the rights of the buffer so that it becomes only executable (no longer readable or writable). We do that with the function VirtualProtect and the memory protection constant used is **PAGE_EXECUTE**.

This constant is clearly defined in http://msdn.microsoft.com/en-us/library/windows/desktop/aa366786(v=vs.85).aspx and it is written about: "Enables execute access to the committed region of pages. **An attempt to read from or write to the committed region results in an access violation**. This flag is not supported by the **CreateFileMapping** function."

After testing, we find that the opcodes are executable and not writable, as expected. However, they are perfectly readable. In agreement with the quote made just above from the msdn, it should not be the case. Technically, an application that uses this feature has its opcodes readable, even if this one didn't want that. This issue compromises the confidentiality of the opcodes from a program.

Our tests have been done on Windows 7 Pro SP1 (x64) and Windows 8 Pro (x64).

You will find, attached to this email, tow codes in assembler which are commented to prove what we explained. One code is written in assembler Intel x86, the other in x64. The x86 has been compiled with Microsoft (R) Macro Assembler Version 6.14.8444 and the x64 with Microsoft (R) Macro Assembler (x64) Version 11.00.51106.1. Both show the same results (we can read the *executable only* opcodes). If you can not compile the codes in assembler, we can provide to you the executables.

It may be us that make a mistake and misunderstand the msdn. But in this case, what is the difference between PAGE_EXECUTE and PAGE_EXECUTE_READ?

As this problem has been found after a process of research from us, we inform you that we will publish this result on our blog (cvo-lab.blogspot.com) in 2 or 3 weeks. We assume that this time is sufficient for you to solve the problem. We do not know if this issue will receive a CVE number, in which case, thank you communicate it to us.

Waiting for your reply,

Best regards,

DAVID Baptiste.

Just over 24 hours later, we had received an answer from Microsoft.

Microsoft Security Response Center [secure@microsoft.co... Actions

*En réponse au message de DAVID Baptiste, mar. 17:44*

À : DAVID Baptiste

Cc : FILIOL Eric; Microsoft Security Response Center [secure@microsoft.com]

mercredi 13 mars 2013 23:01

Hello David,

Thank you for your report. Intel x86 and x64 does not support an execute only(non-readable) page. We are looking to update the MSDN documentation to clarify this limitation.

Best Regards,
Nate

Before anything else, I want to thank Microsoft for their prompt response and the interest they have shown for our problem. However, the answer is not very satisfactory. The fault does not appear to come from Microsoft, but from Intel architecture processors. What does that mean? Simply that Microsoft (great friend with Intel by the way) has implemented features that do not exist on the Intel's processors. Here, there are two possibilities. Either Microsoft takes us for a ride or they've made a huge mistake in their documentation.

To try to see the true from the false, I implemented an equivalent code on Unix to test if what *works* on Windows *works* on Linux too. The implementation is freely inspired from [9].

```
int main (int argc, char *argv[], char *envp[]){
    int fd;
    struct sigaction sa;
    unsigned char value = 0;

    /* Install segv_handler as the handler for SIGSEGV. */
    memset (&sa, 0, sizeof (sa));
    sa.sa_handler = &segv_handler;
    sigaction (SIGSEGV, &sa, NULL);

    /* Allocate one page of memory by mapping /dev/zero. Map the memory
    as write-only, initially. */
    alloc_size = getpagesize ();
    fd = open ("/dev/zero", O_RDONLY);
    memory = mmap (NULL, alloc_size, PROT_WRITE, MAP_PRIVATE, fd, 0);
    close (fd);

    /* Write to the page to obtain a private copy. */
    memory[0] = 0xff;

    /* Make the memory unwritable. */
    mprotect (memory, alloc_size, PROT_EXEC);

    value = memory[0]; /* Try to read the value. */

    printf("The value is : 0x%02x.\n", value); /* It works ! */

    /* All done; unmap the memory. */
    printf ("all done\n");
    munmap (memory, alloc_size);
    return 0;
}
```

The result is similar to the one observed under Windows. But unlike Windows, Linux does not claim that it is possible to have executable-only rights on the processes memory [6]. This experience is therefore in the direction of the response of Microsoft and it helps to confirm that the memory protection is dependent to the processor [7].

In truth, the processor has a registry NX that allows it to manage the memory protection [10]. In the same way that OpenBSD was a pioneer in this field with its W^X feature [11], it is only possible for a processor to prevent the access of a piece of memory only with write or (exclusive or) execute rights (of course, both are possible). But, there is nothing about the only executable rights. It's possible to have no rights on a piece of memory, but it's not useful in our case because we couldn't execute any more our code…

In conclusion, in the absence of alternatives, it is not possible to guarantee the confidentiality of op-codes executed in memory, even if Windows has promised it in its documentation. It is almost as if we promised you a magic hammer which sings when you hit nails. Except that, in truth, you have a rusty hammer which bends nails. It is not just about that the tool is not good. But it is rather as if we promise to a child a candy without giving it to him. It's really frustrating when you are this child…

At the end, one may ask about "why" such an error is in the msdn. By falsely suggest that this feature can protect your codes, it is a good way to spy on your best features that you want to protect. Based on the fact that, only those who have something to hide, hide it, others do not... Here, we do not allow you to hide anything. In truth, we highlight, for those who know what to watch, what you want to hide. It's worth!

If we had wrong mindset, we could think that this story may result in a vey interesting backdoor. A vey interesting because it is useful and easy to refute if someone find it (claim that it's an unfortunate omission or confusion is so easy). In addition, the correction of this error, in the way that Microsoft envisages it, is completely mad. We take an eraser and we erase the documentation, even if this could be an advantageous feature for software engineer… We just have to continue to sleep, to dream and to believe in fairy tales.

Note that all the paper has been perform without the use of reverse engineering. Every thing which has been described here has been studied indirectly by the interaction between the system and the features implemented in our codes. It is amazing what we can find when we just try to do what is written in the documentation...

Bonus: Does-it work in ring0? Just to help you to make your own opinion, consider ZwAllocateVirtualMemory [8] and check what you can do with the last argument *ULONG Protect*. The protection rights seem to be the same. The question is simple, does it work better? Of course… And nothing is provided in the remarks or elsewhere in page [8] to clarify the possible limits of the use. Surprising isn't it?

ZwQueryFullAttributesFile
ZwQueryInformationFile
ZwQueryInformationToken
ZwQueryKey
ZwQueryObject
ZwQueryQuotaInformationFile
ZwQuerySecurityObject
ZwQuerySymbolicLinkObject
ZwQueryValueKey
ZwQueryVolumeInformationFile
ZwReadFile
ZwSetEaFile
ZwSetEvent
ZwSetInformationFile
ZwSetInformationThread
ZwSetInformationToken
ZwSetQuotaInformationFile
ZwSetSecurityObject

*Protect* [in]

Bitmask containing page protection flags that specify the protection desired for the committed region of pages. The possible values are:

| Flag | Meaning |
|---|---|
| PAGE_NOACCESS | No access to the committed region of pages is allowed. An attempt to read, write, or execute the committed region results in an access violation exception, called a general protection (GP) fault. |
| PAGE_READONLY | Read-only and execute access to the committed region of pages is allowed. An attempt to write the committed region results in an access violation. |
| PAGE_READWRITE | Read, write, and execute access to the committed region of pages is allowed. If write access to the underlying section is allowed, then a single copy of the pages is shared. Otherwise the pages are shared read only/copy on write. |
| PAGE_EXECUTE | Execute access to the committed region of pages is allowed. An attempt to read or write to the committed region results in an access violation. |
| PAGE_EXECUTE_READ | Execute and read access to the committed region of pages are allowed. An attempt to write to the committed region results in an access violation. |

DAVID Baptiste.

Ps: Thanks to Sébastien and Clarisse for their help.

**References:**

[1] Memory protection, http://en.wikipedia.org/wiki/Memory_protection, Wikipedia, 13/03/2013

[2] Executable space protection, http://en.wikipedia.org/wiki/Executable_space_protection, Wikipedia, 02/02/2013

[3] VirtualAlloc function (Windows), http://msdn.microsoft.com/en-us/library/windows/desktop/aa366887(v=vs.85).aspx, MSDN, 26/10/2012

[4] VirtualProtect function (Windows), http://msdn.microsoft.com/en-us/library/windows/desktop/aa366898(v=vs.85).aspx, MSDN, 26/10/2012

[5] Memory Protection Constants (Windows), http://msdn.microsoft.com/en-us/library/windows/desktop/aa366786(v=vs.8).aspx, MSDN, 20/10/2012

[6] mprotect - set protection on a region of memory, http://man7.org/linux/man-pages/man2/mprotect.2.html, man pages, 14/08/2012

[7] Memory protection on an OS, http://stackoverflow.com/questions/15045375/memory-protection-on-an-os, StackOverflow, 23/02/2013

[8] ZwAllocateVirtualMemory routine, http://msdn.microsoft.com/en-us/library/windows/hardware/ff566416(v=vs.85).aspx, MSDN, 3/12/2013

[9] 8.9 mprotect: Setting Memory Permissions, http://www.informit.com/articles/article.aspx?p=23618&seqNum=10, informit, 12/11/2001

[10] NX bit, http://en.wikipedia.org/wiki/NX_bit, Wikipedia, 02/03/2013

[11] W^X, http://en.wikipedia.org/wiki/W^EX, Wikipedia, 07/03/2013