Journal of Computer Virology and Hacking Techniques
https://doi.org/10.1007/s11416-020-00347-x

ORIGINAL PAPER

Exploiting flaws in Windbg: how to escape or fool debuggers from existing flaws

François Plumerault[1] · Baptiste David[1]

Received: 9 August 2019 / Accepted: 2 January 2020

Abstract
In order to perform their goals without being detected, Malware should have a battle of wits with the analyzer. Such a way, they use a large variety of stealth methods to perform their missions. These methods allow to slow or block analysis. Most of the time, these tricks are often operating system or CPU oriented (dll injection, exception handler or API abuse). In addition, they are although focused on the most used analyst tools. These attacks, allow, among other things, to display erroneous information on the analysis tools or to silently detect it so that the malware can change its behavior in case of analysis. Depending of the degree of error of the analyzing tools used, it could become partially or totally ineffective. More than just flowed malware analysts, it is a great drawback in order to find bugs in regular software. In this article, we show how to exploits errors inside debuggers and mainly inside one of the most use: Windbg. This list of errors impacting this Microsoft's tool mainly concerns few flaws in the disassembly engine or in the debug procedure. Some are present in the debugger from years... More directly, we show different ways to block or disturb the normal behaviour of Windbg. Thus, even if these errors are not always critical, they can negatively impact the use of software by any user. For instance, we describe a new way to know if the current process is running under the control of Windbg. This is exactly what malware author are looking for to detect analysis. Due to the complexity of architecture such as x64 and x86, it is hard to design and develop a complete disassembling tool. In fact, no disassembling tool is perfect and most of those we tested have at least one of the flaws which are shown in this article. Among the different flaws, we have int 3 misinterpretation, wrong jump interpretation, partial instruction prefix handling and unsupported instruction. Moreover, nothing prevents these tools to have other kind of errors. Thus, in order to analyze software efficiently, it is necessary to improve the analyzer tools. In this way, we offer different solution to correct the bug we encounter on the different tools.

Keywords Windbg · Exploit · Debugger · Vulnerability · Detection · Mawlare

1 Introduction

From an historical point of view, debuggers are the best friends of developers. It allows them to understand what is happening in their applications and how they can solve remaining bugs. But more than helping to find and fix bugs, they are powerful tools able to analyze efficiently processes. As a logical consequence, malware authors have always tried to escape or directly affect these tools' results.

Detecting or evading debuggers is a sport which is famous by the number of different technics used to achieve such a goal. The technic used can be split in two groups. The first is about the use of dedicated operating system API able to release or neutralize the influence of the debugger. The second is about bugs exploited or code misinterpreted by debuggers. For obvious reasons, it is the second group which is the most complex to manage, since it requires an evolution of the tool or an evolution in the practice of the analysis. But, because it is driven by the technology, debuggers and malware analysts are always more and more efficient to counteract technics used to escape dynamic analysis. It explains why it is relevant to present and fix new technics or bugs exploited from debuggers to detect, escape or subvert analysis.

Our paper is based on Windows operating system, no matter the version used. This choice is explained by the fact that most threats are on this platform. And the most famous – not to say the most used – debuggers on Windows is the one

✉ Baptiste David
bdavid@et.esiea-ouest.fr

1 Paris, France

developed by Microsoft: Windbg. This one is now part of the Windows Driver Kit and a new version, Windbg Preview, is actually developed with nice extensions such as Time Travel Debugger.

In this article, we are going to show four ways to disrupt the functioning of Windbg disassembler (version 10.0.17763.1 for Windbg and 1.0.1904.18001 for Windbg Preview). Section 1 focuses on existing vulnerabilities in debuggers and especially on one already known in Windbg. Section 2 explains how to use this vulnerability in order to turn it into a debugger detector. Section 3 explains how, by using an operand-size override prefix, to fool the Windbg disassembler. Section 5 uses a similar mechanism with the use of the prefix REX to achieve the same goal. Section 6 explains how to use unsupported instructions to mislead the debugger analysis without impacting the execution. Finally, a conclusion section includes all the elements presented.

2 Previous work on debuggers flaws

Debuggers are software used to drive other targeted software. The goal is to allow a step by step execution, stopping execution according to some events and data inspection or modification. It allows human users to understand where a bug can stand in a software or to unsure that everything worked as expected. It exists a lot of debuggers, such as GDB [1], LLDB [2], Microsoft Visual Studio Debugger [3], Radare2 [4], IDA [5] or Windbg [6].

Technically speaking, the debugger has not the ability to execute itself instruction per instruction. This is the aim of an emulator (such as Boch [7] or Unicorn [8]). Instead of, it uses breakpoints which are inserted in the debugged process. Another way to proceed is based on the use of the trap flag which are inserted in the context of one of more debugged thread. One of the soft trick used is based on the trap flag [9] provided by the CPU. Another trick is the use of breakpoint instruction [10]. The last is based on an instruction set that, when encountered by the CPU, gives back the hand to the process referenced as debugger.

If debugging a program can be very useful, it can although be useful to crack software, evade copy protection, digital rights management and more generally all malware activity which prefers to stand below the radar. Such a way, different methods have been developed to counteract debuggers [11–13]. Most of the time, methods used in order to try to detect the present of debuggers (via communication links, dedicated structures in memory) or they are trying to debug themselves since only one debugger can be present at a time or they try to remove the debugger – but it requires to know a debugger is already present.

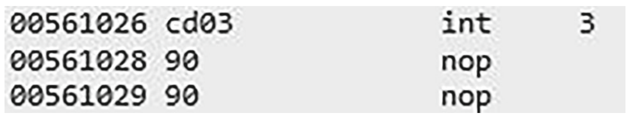
A last trick used is to abuse from bugs in the debugger itself. Indeed, debuggers are software like any others, which means they can be improved. For instance, Ollydbg [14] which is a popular 32-bit user mode debugger had a vulnerability by uncorrectly handling OutputDebugString Windows' API function [15]. A call to this function, by one debugged process from Ollydbg, could have resulted in an unexpected crash [16]. More examples can be found in [17] to explain how to exploit such a debugger's bugs.

One of the bugs used in this paper to exploit Windbg is already partially known [12]. It is based on "int 3h" instruction. This one corresponds to an interruption — the third one — referencing a debug break. When this instruction is encountered, the CPU gives back the control to the debugger process. From the CPU, this instruction can be encoded in two ways, for the same result. Indeed, it could be encoded, from assembly to opcodes executable by CPU, by using either 0xCC or 0xCD 0x03 writing. The reason behind this two encoding stands in the approach used to encode the instruction by the compiler. Indeed, the 0xCC encoding always refers to the third interruption, (debug break). This is the short version of the instruction. However, the longest one, 0xCD 0x03 encoding can refer to any interruption where the id of the interruption is encoded using the second byte of the encoding - 0x03 in our case. Most of the time, only the shortest instruction is used by compilers. Thus, the 0xCC encoding is used a lot more than 0xCD 0x03, except with NASM compiler [18].

More than the waste of a byte to encode the same instruction, the longest version can potentially lead to a bug if it is not taken into account correctly, as suggested in [19,20]. The same way, this behavioral error has already been observed without being explicitly linked to a particular debugger by Peter Ferrie [21]. In this paper, the author has tested a lot of debuggers and emulators without linking this bug to one or more products. More generally, this bug has been used as a "debugger killer". Example is provided in [22] to illustrate how Windbg can be trapped by using these opcodes to make it crash. The same technic is used by malware's packer to avoid analysis under a debugger [23]. But this technic is expected to make crash the application since this one is debugged.

3 INT 3 mishandling exploitation

This is the misinterpretation of the "int 3h" instruction in its long form which can be exploited. From a general point of view, any misinterpretation resulting in a wrong execution by a debugger is a vulnerable flaw which could result to hidden code execution or badly interpreted one. Such a behaviour could lead to debugger detection or the execution of obfuscated code that is difficult to analyze. This is what we are going to show in the two next subsections. The first is about technical details about the flaw in Windbg and the second about the exploitation of that flaw.



```
00561026 cd03          int     3
00561028 90            nop
00561029 90            nop
```

Fig. 1 Interpretation of the debug break instruction before execution

3.1 Technical details

As explained previously, Windbg does not correctly handle the ”int 3h” instruction when this one has been encoded by the compiler in its long form. Before execution, Windbg correctly handles the debug break as the figure bellow illustrates it (Fig. 1).

When the previous code is executed, the next instruction about to be executed by Windbg is the one provided in Fig. 2.

Actually, we are processing step by step, Windbg is incrementing by one the execution pointer (eip in 32 bits – rip in 64 bits) of the debugged thread, as if it would have been the short form version. This is incorrect since it is expected that the debugger increments the current instruction pointer by two as the long form of the debug break instruction would have expected it. The result of this misbehavior is a jump in the middle of the opcodes of the next valid instruction (the one following the debug break) to execute opcode 03 which is an addition (add) instruction. Of course, such behaviour is obviously not about to increase the stability of the program. Most of the time, this behavior will eventually rise an access violation exception, resulting in the crash of the debugged program.

debugger. The problem stands in the fact that 0x03 opcode usually results in an invalid execution flow or in an invalid access memory. The operation expected in such as case is an addition performed with registers or a direct access read in memory, which is not stable. But it is possible to combine this misinterpreted opcode with other opcodes so that the result is still executable and valid.

First, before allowing debugger detection, the code we are writing must be able to survive when there is no debugger. Executing ”int 3h” instruction, when there is nothing registered to handle it, usually results in an application crash. To avoid that, we must ensure that our code which will be executed in an exception handled context. This one is triggered in case of absence of the debugger. Otherwise, this is the debugger itself which is notified first, by default.

In C language, this operation is implemented through a __try/__except block statement [24]. This one can be directly implemented in x86 assembly architecture since exception handling is stored at offset zero in the Thread Information Block (TIB) [25,26] of the current thread. The field responsible of exception handler in the TIB is named Structure Exception Handling (SEH) [27] and it is directly accessible through ”fs:[0]” assembly instruction [28]. To manipulate it, it is just about storing a pointer to a handler function [29] in this structure. This is usually performed via the following instructions, where ”handler” corresponds to a handler function (it corresponds to the ”except” statement bloc in C language).

Listing 1 Implementation of the beginning of exception procedure.

```
assume fs:nothing    ; Avoid segment registers to be considered as an error.
push offset handler  ; Store address of the handler pointer function.
push fs:[0]          ; Store the previous SEH on top of the stack.
mov fs:[0], esp      ; Update the SEH stack.
```

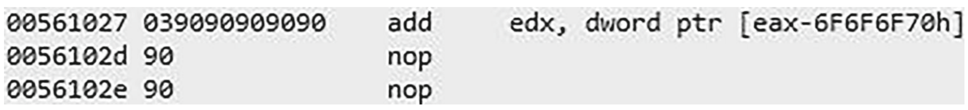
Interestingly, Ferrie worked at Microsoft, in 2008, at a time he published his paper [21]. This one seemed, unfortunately, not to notice that this bug has concerned Windbg. May be it is because this technic is not used efficiently by malware. Indeed, letting an application crash is not the most discreet thing to do. This generally means that people is about to study the reason for the crash and to discover the method used by the malware. The idea is then to propose a method that, exploiting this old and known bug for more than ten years, allows detecting a debugger without crashing.

3.2 Technical exploitation

The main issue with the use of ”int 3h” Windbg’s bug is the resulting crash when exploiting under the control of the

The procedure in x64 would be a bit different since the exception handler is no more stored in the TIB directly, as it was in the x86 architectue, but it is mapped from the MZ-PE executable file. This can be done by the use of Vectored Exception Handler API [30] with the same logic. A dedicated list of functions is registered in order to be used if an exception occurs. This one is able to handle correctly the ”int 3h” instruction when there is no debugger.

Once the handler exception is registered, we have to deal with the debugger’s misinterpretation. The main problem is that the opcodes used must produce a valid result both in the normal case and in the case where they are misinterpreted. The easiest way to achieve this result is to try to reach one of two unconditional jumps. The first in the case when the process is running under a debugger, the second when there is no debugger. All the code between the ”int 3h” instruc-



```
00561027 039090909090    add     edx, dword ptr [eax-6F6F6F70h]
0056102d 90              nop
0056102e 90              nop
```

Fig. 2 Interpretation of the debug break instruction after execution

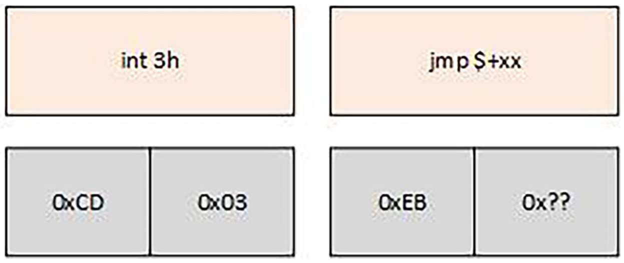


Fig. 3 Illustration of the correct disassembling of ”int 3h”

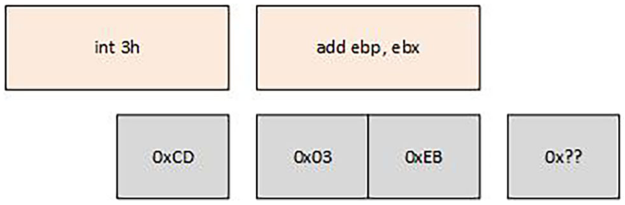


Fig. 4 Illustration of the incorrect disassembling of ”int 3h” by Windbg

tion and the two jumps is designed to avoid or cancel bad consequences of a misinterpretation while keeping normal execution flow.

On way to encode unconditional jump operation is through 0xEB opcode, followed by a single byte indicated the relative offset to jump. This offset value is interpreted as a signed number which results in a jump of −128 to +127 bytes in memory. This is far enough for our code which is just about to jump to a dedicated location returning zero or one, depending of the presence or absence of a debugger.

The most efficient way to build the code is to stick the unconditional jump after the debug break instruction. In this way, the code we have, when there is no debugging, is handled by the exception handler function which gives back the hand to the jump. The Fig. 3 illustrates this procedure.

When Windbg is present, the misinterpretation of the debug break occurs, resulting in a shift inside the instructions flows. These ones are now interpreted as shown in the Fig. 4.

To cancel the ”add ebp, ebx”, it is enough to compute instruction ”sub ebp, ebx”. This last one is encoded with 0x2B 0xEB opcodes. Then, it comes the final unconditional jump to return zero or one, depending if this function must be able to detect the presence or the absence of a debugger. This construction allows the function to cancel consequences of misinterpretation (corruption of ebp register) while keeping the final jump to a location where we can handle if a debugger is present. The final result looks like the Fig. 5.

The case of unconditional jump offsets remains to be resolved. In the case where a debugger is present, the last jump can point to any relevant location for our function. Whatever is the offset, it will not change anything is can of correct or misinterpretation. In the case where there is no debugger and it is our exception handler function which resolves correctly the debug break, offset of the jump is fixed at +0x2B (offset of 43 bytes) as represented in the Fig. 6.

The fixed offset of the unconditional jump in the case where there is no debugger forces us to write relevant code at that point. Instead of using nop instructions as a junk code, we propose to optimize the code so that this one is as optimized as possible. Actually, we are going to write the exception handler function right after the debug break part. It means the jump at a fixed offset will be inside the opcodes of the exception handler function.

Without loss of generality, we decided to build the present code to detect if the program is under the influence of Windbg debugger. More directly, it means the code returns one if a debugger is attached and zero otherwise. Since the exception handler function returns zero (under the defined value EXCEPTION_EXECUTE_HANDLER) to continue execution [29], it makes sense to reuse the last opcodes of the exception handler procedure to return zero when there is no debugger attached.

3.3 Illustration with operational code

In the case where we are executed under Windbg control, the unconditional jump is linked to a dedicated part supposing to return one. Then, it reroutes the execution flow to the original epilogue of the code. Note that there is no need to realign the stack after the exception handler is set up. Indeed, the function’s epilogue reset the stack alignment thanks to ”mov esp, ebp” instruction. Finally, taking into account everything, the code of debugger detection is the one presented in Fig. 2.

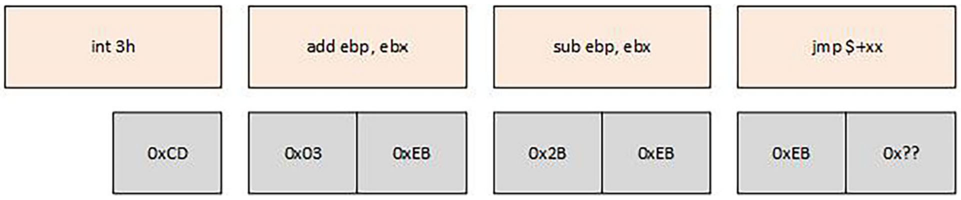


Fig. 5 Correction to cancel side effect of the misinterpretation from Windbg

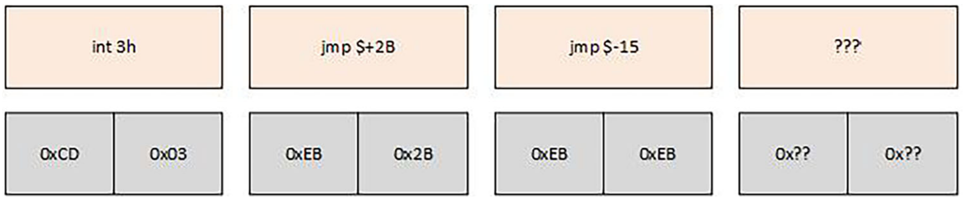


Fig. 6 View of the assembly code executed where there is no Windbg

Listing 2 Final version of the Windbg's detection shellcode.

```
main proc

     push ebp
     mov  ebp, esp
     assume fs:nothing

        ; Register the SEH.
        push offset__handler
        push fs:[0]
        mov  fs:[0], esp

        ; In case of misinterpretation.
        ; [add ebp, ebx]  [sub ebp, ebx]  [jmp $+0E]
        db 0CDh, 003d, 0EBh,  02Bh, 0EBh,  0EBh, 00Eh
        ;  [int 3h]    [jmp      $+2B]
        ; In case of correct interpretation.

        ; Restore original SEH.
        pop fs:[0]
        add esp, 4

     mov esp, ebp
     pop ebp
     ret

     ; Return one if there is a debugger.
     or eax, 1
     jmp $-7   ; Go upper to the epilogue of the main function.

__handler:
    push ebp
    mov  ebp, esp
    mov  eax, [ebp+08h]             ; Retrieve the first parameter.
    mov  edx, dword ptr [eax,+0B8h] ; Get access to the value of instruction pointer.
    add  edx, 2                     ; Add two to jump correctly over "int 3h".
    mov  dword ptr [eax+0B8h], edx  ; Store the new value of rip.
    xor  eax, eax                   ; Return zero.
    mov  esp, ebp
    pop  ebp
    ret

main endp
```

It results that the execution flows depends on the present or the absence of Windbg debugger. Since assembly programming can be complex to understand, we propose to illustrate the execution flow of the program with a the Fig. 7. This one represents the different jumps the code uses to achieve the Windbg detection goal.

The first case is where there is no debugger. In such a case, the "int 3h" instruction is handled by the __handler function registered just before and displayed in blue boxes. The role of the handler [31,32] is to retrieve, form its first parameter (stored at [ebp+08h]) a pointer to an EXCEPTION_RECORD structure [33] where it will be able to change the instruction pointer where the exception handler must give back the hand to the main code. In the case where there is no debugger, the execution still continue to restore the original version of the SEH and finish the function, with the return value (stored in eax) equals to zero.

The main right arrow symbolizes when there is Windbg debugger. In such a case, misinterpretation forces to go a little further on a bloc of code responsible to return one (or eax, 1) and reuses the epilogue of the main's function by a jump with a negative offset (jmp $-7, the small left array) to get access to mov esp, ebp instruction. Such a way, the function returns one when there is a debugger.

Finally, if there is a debugger able to handle correctly the long form of "int 3h" instruction, this one will execute next instruction which is an unconditional jump with a positive offset (jmp $+2b). Compiled with MASM compiler, this last jump in our codes goes on the epilogue of the handler function, the same than the main function. Such a way, it does not change the logic of instructions present in the main functions and it returns zero.

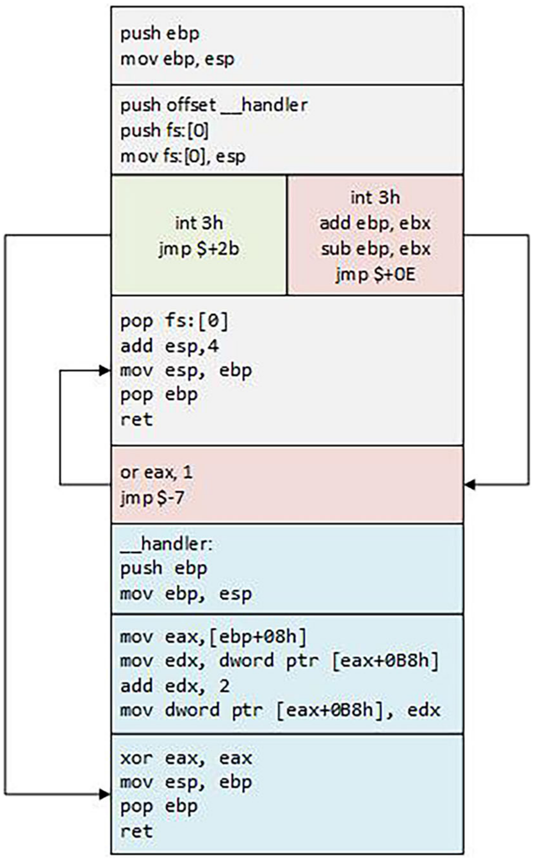

```
push ebp
mov ebp, esp

push offset __handler
push fs:[0]
mov fs:[0], esp

int 3h
jmp $+2b

int 3h
add ebp, ebx
sub ebp, ebx
jmp $+0E

pop fs:[0]
add esp,4
mov esp, ebp
pop ebp
ret

or eax, 1
jmp $-7

__handler:
push ebp
mov ebp, esp

mov eax,[ebp+08h]
mov edx, dword ptr [eax+0B8h]
add edx, 2
mov dword ptr [eax+0B8h], edx

xor eax, eax
mov esp, ebp
pop ebp
ret
```

Fig. 7 Graphical view of the Windbg's detection procedure

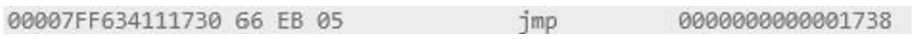
4 Wrong jump interpretation

To work properly, debuggers need a disassembler. Generally, the compilation procedure aims to move from a code, written for example in C, to a single compiled code. The compiled version of the code then depends on the style chosen by the compiler used. The disassembling procedure, which starts from the compiled code to retrieve a human-readable form, is more complex. Indeed, it is necessary to take into account various backwards compatibility issues of the assembler language or the architecture of the CPU manufacturers. There are also the different optimizations, compilers style procedures and various freedoms taken or imposed by CPU manufacturers. In short, it is notoriously complex to write a truly perfect disassembler.

But there is more, since assembly language is a complex mix of different norms. Historically, Intel was leader in the 32-bits architecture. In the old days, when a new architecture was released by Intel, AMD had no choice but to follow it in order to keep its market share. However, when a 64-bit architecture rose on the market, Intel and AMD both proposed their own norms. The fist was the IA-64 architecture, with Itanium CPU, developed by Intel, a totally new architecture which broke with the x86 one. The second was the AMD64 architecture developed by AMD, which mainly consist of adding 64-bit operations in existing x86 architecture. At the end, the success of the IA-64 architecture was mitigated, which resulted, for the first time, that Intel followed the architecture developed by AMD (AMD64) and under the name Intel 64. A more neutral name for the AMD64 and Intel 64 architecture has been finally used in the industry to identify this architecture: x64. This architecture is very well documented by both Intel and AMD. However, there are still few differences between AMD and Intel to know in order to disassemble code correctly. Therein lies the rub.

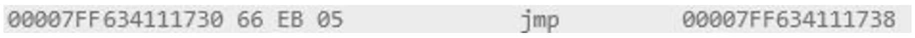
4.1 Technical description

Some of the 64-bit instructions have a strange behavior. These behaviors are described as unpredictable in the Intel documentation [34]. However, it does not mean that these behaviors are really unpredictable. In fact, it just means that



```
00007FF634111730 66 EB 05          jmp         0000000000001738
```

Fig. 8 Correct interpretation under AMD CPU but misinterpretation on Intel CPU due to the prefix used



```
00007FF634111730 66 EB 05          jmp         00007FF634111738
```

Fig. 9 What will be executed on an Intel CPU with the provided opcodes

the x64 instruction set reference does not provide a precise behavior for these instructions.

One of this undocumented behavior is the use of an operand-size override prefix (encoded with 0x66 value at compilation time) on a relative jump. This use of the operand-size override prefix is not a common practice because there is no real operational use for it. Usually, to obtain the destination of the relative jump, we use the following formula:

$$\text{instruction\_pointer} = \text{address\_of\_jump} + \text{size\_of\_jump} + \text{sign\_extended\_displacement}$$

To illustrate with a real example, let's suppose we have a jump at address 0x100000, supposed to jump 5 bytes after. This one is encoded on 3 bytes (0x66 for the prefix, 0xEB for the jump and one byte to encode the offset of 5 bytes). The final result jumps at address 0x100008, since it is the sum of the size of the instruction, plus the offset inducted.

$$0x0100008 = 0x100000 + 3 + 0x05$$

The logic which is described above is always true on Intel processor. Nevertheless, on AMD processor the use of operand-size override prefix creates a totally new logic. Indeed, when a relative jump gets an operand-size override prefix on AMD processor, the formula is different, including now a final operation to only keep the last 2 bytes. The following equation illustrates the procedure to compute the new address where to jump. Note that 0x`FFFF` represents a cast to only keeps the last two bytes of the resulting operation.

$$\text{instruction\_pointer} = (\text{address\_of\_jump} + \text{size\_of\_jump} + \text{sign\_extended\_displacement}) \ \& \ 0x\texttt{FFFF}$$

Now, with a jump which has the same conditions (base address 0x100000 and an inducted offset of 5 bytes) as the one we use before, it will result in a jump at address 0x0008. Details of the computation are provided as follows.

$$0x0000008 = (0x100000 + 3 + 0x05) \ \& \ 0x\texttt{FFFF}$$

4.2 Technical exploitation

Windbg is able to partially handle this kind of behavior. For instance, we will use a relative jump with an operand-size override prefix and an 8-bits displacement (sign extended to 64-bits) with the same characteristic as the precedent example. Such a jump will be encoded ”0x66 0xEB 0x05” and Windbg will give the following interpretation (Fig. 8).

We can see that the result is exact if we assume we are on an AMD processor. Moreover, when we execute it on an AMD processor we will jump at address 0x0000000000001738. The issue stands in the fact that Windbg interprets this jump the same way on Intel processor. However, on Intel processor, it is the first logic presented which is used for jump calculation. Indeed, when we execute the ”0x66 0xEB 0x05” jump on an Intel processor, we will jump at address 00007FF634111738 instead of address 0x0000000000001738. Thus, the correct interpretation that is expected from Windows, when running on Intel processor, is the following (Fig. 9).

This behavior also concerns any conditional jumps (Jcc instructions) although and it can be easily checked by executing them. It is not a big deal since it does not change execution of the process, but it could lead to misunderstand the assembly code analyzed by the debugger. Even if this error will not impact the functioning of the debugged processes, it could confuse the user of Windbg and negatively affecting the analysis. This is one of the few tricks about CPU differences we can find online on Wikipedia [35].

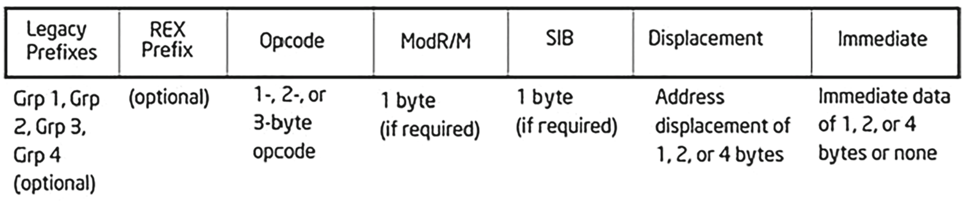
5 Partial instruction prefix handling

5.1 Technical detail

All AMD64 and Intel64 instructions have a structured form which is described in the documentation of the manufacturer: the Intel documentation [36]. An instruction is described as represented in the Fig. 10 below. First, before the opcode itself, we find prefixes. These ones are used as extended information provided to an instruction. Usually, the main purpose is to custom or repeat a specific instruction. In the area of prefixes, we can split them in two different types.

The firsts are legacy prefixes used in x86 architecture to compute specific actions on instructions. One type is lock prefix which is used on certain read-modify-write instructions in order to prevent simultaneous access to the memory. A second one is the repeat prefix that causes string handling instructions to be repeated. This one is usually driven by the content of the rcx (ecx when using 32-bits address

Fig. 10 Illustration of the assembly semantic by Intel



Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

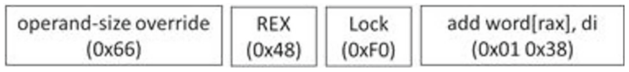


Fig. 11 Illustration of undocumented use of REX prefix in assembly semantic

size) register to evaluate the number of times an instruction must be repeated. The same way, we can talk about branch taken/not taken prefixes which give clues to the CPU to lessen the impact of branch misprediction. Another one is the operand-size override prefix, which is normally used to switch between an instruction from 32-bits to 16-bits operand. Finally, there is the address-size override prefix, which can be used in 64-bit mode to use 32-bit addressing memory.

The second type of prefixes is REX prefix. This last one has a particular encoding. In fact, unlike the legacy prefix, this one is encoded using values from 0x40 to 0x4F range. Its lower nibble allows encoding several properties that have two main purposes. The first is to allow 64-bits operand size on some instructions which usually use 32-bits operand size. This is a smart way to extend existing opcodes from x86 to be usable easily for x64. The second use is to access newly added registers in 64-bit mode (r8 to r15, xmm8 to xmm15, ymm8 to ymm15, cr8 to cr15 and dr8 to dr15). In fact, the REX prefix could modify the initial behavior of the ModR/M byte and SIB byte [37] to make them access new registers. But the REX prefix, despite being part of the x64 instruction semantic and defined in the documentation, is still perfectly optional.

Then, it comes the remaining of the instruction encoding with the opcode itself which provides the real meaning of the operation. Finally, the last bytes are about memory or register involved in the operation and how they are involved.

Even though the position of the REX prefix is defined, in this figure extracted from Intel documentation [34], between the legacy prefix and the opcode, it is not always encoded in such a way. Indeed, the REX prefix property (64-bit instruction and access to new registers) will only be taken into account when it is right before the opcode. However, there could be several REX prefixes mixed with legacy prefixes, as long as the total instruction size does not exceed the instruction maximum size of 15 bytes [38]. One example of undocumented use of the REX prefix can be seen in Fig. 11.

In this example, we can see that there is a legacy prefix (lock) between the REX prefix and the opcode (direct addition of a value stored in memory where its address is referenced via rax register). This lock prefix is used for two main purposes. Firstly, the access to the memory is protected during the operation. Secondly, the impact of the REX prefix is now disable and we can consider the 0x48 byte as irrelevant (sort of ”nop” prefix, in a way). However, the operand-size override prefix is still valid and it impacts the size of the operand. Thanks to the operand-size override prefix, the operation is using WORD instead of DWORD operand size.

5.2 Technical exploitation

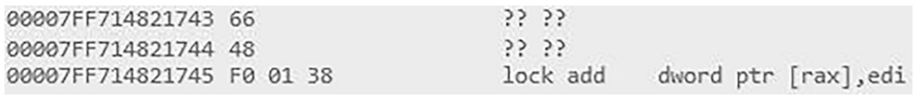
Nevertheless, Windbg does not handle this kind of use of the REX prefix correctly. Thus, for the preceding example encoded ”0x66 0x48 0xF0 0x01 0x38”, Windbg is going to interpret it baldy (Fig. 12).

The correct answer would be about to accept the REX prefix inside the instruction but to ignore its property and therefore the operand-size override prefix. Thus, the correct interpretation of the instruction would be the following assembly code (Fig. 13).

This, behavior — which also concerns GDB [1] debugger — allows shellcode to be potentially unreadable by Windbg when they are analyzed. In fact, it would be very easy to abuse the use of useless prefix in order to disturb the use of Windbg and GDB or any vulnerable debugger. Indeed, we can easily make Windbg guess wrong more than half of the opcodes bytes used in a single instruction. Thus, it would be very troublesome to analyze assembly code for human readers while preserving the correct behavior of the code executed by the CPU.

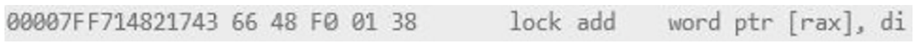
6 Unsupported instruction

Despite the fact that most of the AMD64 instructions are well documented, some instructions are not. These instructions are not documented for two reasons. The first reason is that they could be used internally by Intel or AMD for their own purposes. The second reason stands in the fact that they could create new instructions in the future. In order to keep these instructions work on former processor, they assign them at nop (no operation).



```
00007FF714821743 66             ?? ??
00007FF714821744 48             ?? ??
00007FF714821745 F0 01 38       lock add    dword ptr [rax],edi
```

Fig. 12 Misinterpretation of code by Windbg due to REX prefix



```
00007FF714821743 66 48 F0 01 38    lock add    word ptr [rax], di
```

Fig. 13 The code provided in Fig. 12 should be interpreted like this one

In consequence, it could be complex to handle every single instruction. In that respect, there is not a lot of tools that handle all instructions. Indeed, radar2, GDB, x64gdb as well as Windbg [1,4,6,39] do not manage all instructions. For instance, there is a NOP encoded 0F 19 /r[1] that Windbg does not handle correctly. This one, encoded with any register selected, should be observed in a debugger as a regular nop (Fig. 14).

In fact, Windbg is totally lost when it met this instruction above and it describes it as something entirely different, as we can see below (Fig. 15).

Moreover, Winbdg, like IDA, radar2, GDB and x64dbg, only partially checks the `cpuid` instruction of the processor before disassembling and debugging. It means, they do not take into account the exact features supported by the CPU of the machine they are executed from. Thus, when Windbg faces Intel MPX instructions [40], it would always describe them as if they were supported on the current CPU, even if the CPU does not support them. It means that the debugger tries to interpret them even if such instructions behave as NOP or invalid instruction when they are not supported. This is a disadvantage since the debugger is describing a reality that is quite different from the one perceived by the program currently debugged on the machine. The debugger is interpreting its own reality, not the one of the current CPU. The Fig. 16 shows an instance of such Intel MPX instruction not supported on our CPU (AMD Ryzen 7 1800X).

This reality is also true for the IDA software [5], which decompiles everything it can. It should be noted, however, that IDA is not just only a debugger and that its role is first and foremost to disassemble. Thus, having the ability to interpret instructions that the CPU of the host machine cannot execute is not a bad thing in itself, even if IDA could highlight in a better way the fact that some instructions are not supported.

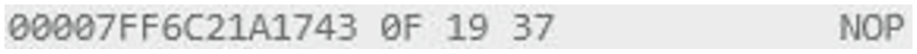
A good remediation would be to take into account the information from the current CPU to determine on which architecture the debugged program is running. This is true for debuggers who are required to execute code on the machine on which the targeted program is running (the case of the network debug aims to identify the remote hardware in the same way). The case of static analysis tools such as IDA is more complex. Indeed, they may have to evaluate binaries that are not supported by the CPU architecture of the current machine.

7 Conclusion

This paper aims to present bugs in Windbg debugger and to explain how to exploit existing generic vulnerabilities in debuggers to detect Windbg. More than the result about vulnerability exploitation, the method presented and used here is sufficient to allow that some kind of bugs or misinterpretations in a debugger to be exploited by a malware. The goal for malware is to avoid detection when it knows it is currently executed in an analyzed environment. Otherwise, it is possible to complicate the analysis task by human through the help of debugger tools by the use of misunderstand specificities between different architectures of processors. Debuggers, since they allow to find bugs in software or to analyze malware, should be reliable, efficient and accurate. Otherwise, the trust between the analyst and the tool could be broken, complicating much more the work of analysis.

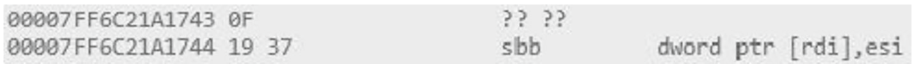
Our study has been mainly focused on Windbg since it is one of the most famous debugger and one of the most largely used by malware analysts. All the tests reported here have been driven on third party debuggers with the results provided in the table below. The main conclusion is that there is no debugger definitely perfect and that all could improve their software in order to provide a much more accurate result. Note that, we have contacted Microsoft the 13/06/2019 and the 23/07/2019 to inform them about troubles in Windbg. After acknowledging receipt of emails and forwarding them to the appropriate person, we had no further news. Of course, bugs are always present and they are still exploitable. The disclosure time elapsed from a long time, this is why we publish them.

[1] http://ref.x86asm.net/coder32.html#x0F19.



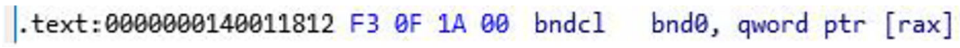
```
00007FF6C21A1743 0F 19 37                NOP
```

Fig. 14 Unsupported instruction should be interpreted as a nop



```
00007FF6C21A1743 0F                    ?? ??
00007FF6C21A1744 19 37                 sbb      dword ptr [rdi],esi
```

Fig. 15 Unsupported instruction is not correctly interpreted by Windbg which tries to provide a meaning



```
.text:0000000140011812 F3 0F 1A 00  bndcl   bnd0, qword ptr [rax]
```

Fig. 16 Instance of unsupported CPU instruction interpreted as nop

	Windbg	IDA	Radare2	GDB	x64dbg
Manage Rex	No	Yes	Yes	No	Yes
Manage int 3	No	Yes	Yes	Yes	Yes
Manage AMD specific instruction	Yes	Yes	No	No	Yes
Manage CPUID	No	No	No	No	No
Manage undocumented instruction	Partially	Yes	Yes	Partially	Yes

More than Microsoft, all the possible bugs exploitation reported in this paper have been submitted organizations responsible to develop debuggers. The main recommendation is to fix disassembling issues and to use cpuid instruction check so that the debugger knows exactly on which architecture it is running. Such a way, it could be able to calibrate efficiently the methods it uses to perform the disassembly operations. In addition, a better implementation of debug break management for Windbg is strongly recommended so that it is not exploitable by malware.

Finally, this study could be continued by the test of new instructions provided by last generations of CPU on the market in addition to other old ones, kept for backward compatibility purposes. Checking the differences between what the debugger expects and the reality of process execution is always a good way to find tricks to detect or evade debuggers.

References

1. GDB community, GDB: The GNU Project Debugger, GDB (2019). www.gnu.org/software/gdb
2. LLVM community, The LLVM Compiler Infrastructure, LLVM Foundation (2019). https://llvm.org
3. Microsoft, First look at the Visual Studio Debugger, MSDN (2019). https://docs.microsoft.com/fr-fr/visualstudio/debugger/debugger-feature-tour?view=vs-2019
4. Radar community, Radar2, Radar Community (2019). https://github.com/radare/radare2/
5. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. No Starch Press, San Francisco (2011)
6. Microsoft, Download the Windows Driver Kit (WDK), MSDN (2018). https://docs.microsoft.com/fr-fr/windows-hardware/drivers/download-the-wdk
7. Bochs community, Bochs x86 PC emulator, Bochs, 16 July 2019 (2019). https://sourceforge.net/projects/bochs/
8. Quynh, N.A., Vu, D.H.: Unicorn The ultimate CPU emulator, Unicorn (2017). https://www.unicorn-engine.org
9. Intel, Intel 64 and IA-32 Architectures Software Developer's Manual, Intel documentation, vol. 1, pp. 3–17 (2019)
10. Intel, Intel 64 and IA-32 Architectures Software Developer's Manual, Intel documentation, vol. 1, pp. 6–13 (2019)
11. Kulchytskyy, O., Kukoba, A.: Anti Debugging Protection Techniques With Examples, Apriorit (2019). https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software
12. Ferrie, P.: The "Ultimate" Anti-Debugging Reference, Ferrie (2011)
13. Afianian, A.: Malware Dynamic Analysis Evasion Techniques: A Survey, Arxiv, November 2018, arXiv:1811.01190
14. Yuschuk, O.: OllyDbg, OllyDbg (2014). http://www.ollydbg.de
15. Microsoft, OutputDebugStringA function, MSDN (2018). https://docs.microsoft.com/fr-fr/windows/win32/api/debugapi/nf-debugapi-outputdebugstringa
16. Securityfocus, OllyDbg Debugger Messages Format String Vulnerability, securityfocus (2004). https://www.securityfocus.com/bid/10742
17. Ferrie P.: Anti-unpacker tricks—part twelve, Microsoft (2010). https://www.virusbulletin.com/virusbulletin/2010/09/anti-unpacker-tricks-part-twelve
18. NASM community, NASM, The NASM development team (2018). https://www.nasm.us
19. Tully, J.: An Anti-Reverse Engineering Guide, code project (2008). https://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide#BpInt3
20. Yang Reiley, Data Breakpoints, Microsoft (2011). https://blogs.msdn.microsoft.com/reiley/2011/07/21/data-breakpoints/
21. Ferrie P.: ANTI-UNPACKER TRICKS, Microsoft (2005). http://pferrie.host22.com/papers/unpackers.pdf
22. Zhang, Y.K.: Software Debugging. Publishing House of Electronics Industry, Beijing (2008)
23. Bowes, R.: In-depth malware: Unpacking the "lcmw" Trojan, SkullSecurity (2014). https://blog.skullsecurity.org/2014/in-depth-malware-unpacking-the-lcmw-trojan
24. Microsoft, Try-except Statement, MSDN (2018). https://docs.microsoft.com/en-us/cpp/cpp/try-except-statement?view=vs-2019

25. Microsoft, Thread Environment Block (Debugging Notes), MSDN (2018). https://docs.microsoft.com/en-us/windows/win32/debug/thread-environment-block--debugging-notes-
26. Microsoft, TEB structure, MSDN (2018). https://docs.microsoft.com/fr-fr/windows/win32/api/winternl/ns-winternl-teb?redirectedfrom=MSDN
27. Microsoft, Structured Exception Handling, MSDN (2018). https://docs.microsoft.com/en-us/windows/win32/debug/structured-exception-handling
28. Microsoft, Using an Exception Handler, MSDN (2018). https://docs.microsoft.com/en-us/windows/win32/debug/using-an-exception-handler
29. Microsoft, Using a Vectored Exception Handler, MSDN (2018). https://docs.microsoft.com/en-us/windows/win32/debug/using-a-vectored-exception-handler
30. Microsoft, Vectored Exception Handling, MSDN (2018). https://docs.microsoft.com/en-us/windows/win32/debug/vectored-exception-handling
31. Czumak, M.: Windows Exploit Development—Part 6: SEH Exploits, Security Sift (2014). https://www.securitysift.com/windows-exploit-development-part-6-seh-exploits/
32. Swiat at Security Research & Defense, Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP, Microsoft (2009). https://msrc-blog.microsoft.com/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/
33. Microsoft, EXCEPTION_RECORD structure, MSDN (2018). https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_record
34. Intel, Intel 64 and IA-32 Architectures Software Developer's Manual, Intel documentation, vol. 2, pp. 2–1 (2019)
35. Wikipedia, x86-64, Wikipedia foundation (2019). https://en.wikipedia.org/wiki/X86-64#Differences_between_AMD64_and_Intel_64
36. Intel, Intel 64 and IA-32 Architectures Software Developer's Manual, Intel documentation, vol. 2, pp. 2–8 (2019)
37. William Swanson, Understanding Intel Instruction Sizes, Swanson Technologies (2003). https://www.swansontec.com/sintel.html
38. Intel, Intel 64 and IA-32 Architectures Software Developer's Manual, Intel documentation, vol. 2, pp. 2–20 (2019)
39. x64dbg community, x64dbg debugger (2013). https://x64dbg.com/
40. Oleksenko, O., et al.: Intel MPX explained: a cross-layer analysis of the intel MPX system stack. In: Proceedings of the ACM on Measurement and Analysis of Computing Systems (2018). https://intel-mpx.github.io/code/submission.pdf