Old D♌g - New Tricks

aka

-Finding Flaws in OllyDbg v2.01-

Author: Kyriakos Economou

www.anti-reversing.com

Date: 10d/12m/2013

“Any fool can know. The point is to understand.”
— Albert Einstein

Aknowledgements

This paper is dedicated to my family, my friends, and to Irene.

Table of Contents

1. Introduction

Bridging knowledge with understanding is what everyone should pursue. This is what motivated me to write this paper. Being a lover of anti-reversing tricks and a reverse engineer in constant will to learn as much as possible, makes me realize that sometimes we ignore important details in things that we consider more superficial than they really are.

In malware analysis, for example, it happens very often that we need to deal with various tricks implemented in order to make static and/or dynamic analysis of binaries much harder. Same and similar techniques are also implemented into commercial software protections, often called as packers, and into various DRMs. Who is learning from who is a difficult thing to establish. I would say that both sides teach each other. Malware authors make use of tricks that software protections implement and at the same time anti-piracy oriented code implements tricks that a malware author might have first designed, which later became public when a virus sample was caught and analysed.

However, along with the anti-reversing tricks being 'invented' through time we also need to consider the new analysis tools that arise, and even more importantly the newer versions of tools that we know that are/have been constatly targetted by the anti-reversing tricks that we often need to deal with. This brings me to the target tool of choice for this paper which is **Olly Debugger v2.01**.

Olly Debugger is with no doubt one of the best reverse engineering tools ever existed (Thank you Oleh!), and my tool of choice for analysing and debugging Windows 32-bit applications either during malware analysis or for bug hunting during exploit development. It is also one of the most supported tools by the community with more than 200 plugins available for it, and all of this for free. The new version of Olly contains some significant changes regarding to the internals of the debugger engine itself. We will talk about these in detail later, but for now it is very important to mention that these changes do/will affect the behaviour of several implementations of anti-reversing tricks in existing/future applications. Changes in behaviour of such tricks actually means that they can either become useless or generaly missbehave during dynamic analysis.

You might ask yourself (or myself) at this point why do we need to care about this? The answer is quite simple, even if obvious might be not, and that is **evolution of anti-reversing tricks**. In a few words, we should expect to see changes to existing tricks in order to become OllyDbg v2.01 compatible, or incompatibe depending on the point of view. It's like we force a natural selection that promotes several genes mutations to make living organisms (or viruses) stronger in order to survive against their enemies or other harming agents.
Olly is just an example, a case study to help us go two steps forward while malware authors are still thinking about their next single step, and a chance to prove that new/updated tools don't necessarily mean less problems. In a few words, this paper is based on research that I performed over Olly v2.01 in order to estimate or predict future changes in existing anti-reversing tricks or the creation of new ones based on potentially new characteristics or features of this new version of Olly Debugger.

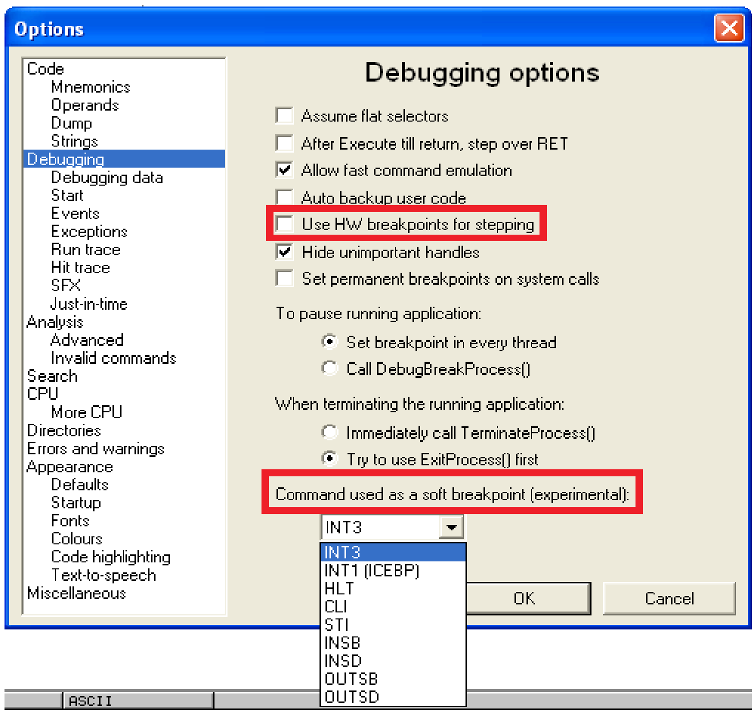
2. New Debugging Features in Olly v2.01

The new features of Olly v2.01 are not limited to those described in this paper. We will only focus, as already mentioned, to those that can affect the behaviour of existing anti-reversing tricks and

trigger as a result their evolution in the near future.
However, we will also discuss about behavioral changes of other tricks that are more generic and don't target just Olly, but in general any Ring3 debugger that operates using the same programming and functional concepts.

The two main changes, that someone can visibly spot through the debugger configuration, are the following:

- **Software breakpoints that use INT1, HLT, CLI, STI INSB, INSD, OUTSB, OUTSD instead of INT3.**
- **Use HW breakpoints for instruction stepping.**



Debugging options

- ☐ Assume flat selectors
- ☐ After Execute till return, step over RET
- ☑ Allow fast command emulation
- ☐ Auto backup user code
- ☐ Use HW breakpoints for stepping
- ☑ Hide unimportant handles
- ☐ Set permanent breakpoints on system calls

To pause running application:
[✓ Set breakpoint in every thread]
Call DebugBreakProcess()

When terminating the running application:
Immediately call TerminateProcess()
[✓ Try to use ExitProcess() first]

Command used as a soft breakpoint (experimental):
INT3
- INT3
- INT1 (ICEBP)
- HLT
- CLI
- STI
- INSB
- INSD
- OUTSB
- OUTSD

Figure 1. New Dbg Features

These functional additions to the debugging engine of Olly are not just an updated buffet to sattisfy your need of having more options, even if you don't really care what internally this means. In the next part of this paper we will examine these updates starting with software breakpoints in terms of functionality against older and potentially new tricks that target exactly this feature of any debugger.

3. Software Breakpoints -Not Just INT3-

The most common and 'natural' way, in terms of the x86 CPU architecture, to set a software breakpoint is via the INT3 software interrupt. When this instruction is executed, a specific exception will be raised and the OS will stop execution of that process while the process debugging that process will be notified about that specific event. This will trigger the debugger's exception handler which will decide if it will either handle the exception appropriately or pass it to the

debuggee to be processed by it.
The fact that a debugger might handle an exception occured or pass it to the debuggee, it is of great importance in terms of anti-reversing, and we will see later the reasons behind this concept (see: 3.1.3). Going back to Olly v2.01 we are now able to use more instructions instead of the well known INT3 (Figure 1). This means that when we choose one of the other instructions to break the execution like we have been doing till now with INT3 (0xCC), we implicitly change the behaviour of the debugger. In a few words we say **'when a software breakpoint exception occurs, ignore it aka pass the exception to the application'**.
So, how does Olly now know when to stop when using the other instructions for the same purpose? If you look closely to those instructions you will realize that have something in common, well except from INT1 (0xF1), and that is that these are privileged instructions, or in other words instructions that are only allowed to be executed from Ring0, which again means that if we attempt to execute one of those in Ring3 then a **privileged instruction exception** will be raised.
Since Olly is only used for debugging applications this makes sense. Putting the facts together, what the debugger will now know is that **'if a privileged instruction exception occured and the instruction is the one chosen for setting software breakpoints, and it is actually set by us, then stop there and wait for user interaction, otherwise show exception to the user.**
Another very important common characteristic of those instructions, including INT1, is that they all are single-byte instructions, just like the INT3.
On the other hand, the outcast of this new feature, the INT1, works in a different way raising another type of exception which is a **single step exception** type that the debugger will also either process or choose to pass the exception to the application. Generally, a debugger will process by default single step and software breakpoint exceptions, unless it is instructed by the user to perform otherwise.
The debugger itself can enable instruction single step execution. This is normally achieved by setting the trap flag in the EFLAGS register which will cause the execution of an INT1 by the CPU upon the execution of the next instruction, but remember there is also a new feature for that (Figure 1). We will be talking with more detail about single stepping over different types of instructions later during this paper (see: 3.1.4).
What is interesting to mention before proceeding, is that **Olly will update the saved breakpoints with the new instructions if you choose to change the settings and restart the debugee**. You can try this by yourself by setting a few breakpoints using a single type of the available instructions (INT3 or INT1 etc...) and then change the setting and restart the debugee. Of course this change is transparent to you but you can verify this by directly assembling an instruction into the code that reads back to a register the first byte of an instruction where you have set a breakpoint.

3.1 Fun with Software Breakpoints

In this section we will examine the new method of setting sw breakpoints, against anti-reversing tricks that attempt to detect the presence of them on runtime, as well as a debugger detection method via INT3 exceptions.

3.1.1 Search for 0xCC!

There are 2 main ways of detecting sw bpreakpoints on runtime. The first method is easily implemented by directly looking for 0xCC bytes, usually among the first 5 bytes of a function where we meet the function prologue. Since 0xCC can also be part of constant makes sense that we don't want in general to search through an entire function in order to avoid false positives.

Function Prologue:	push ebp
	mov ebp,esp
	sub esp, const

Table 1. Function Prologue

This method will not work anymore in case you use one of the other instructions instead of INT3. In a few words, if you choose to use this feature of Olly, you are basically immune to direct sw breakpoints detection.

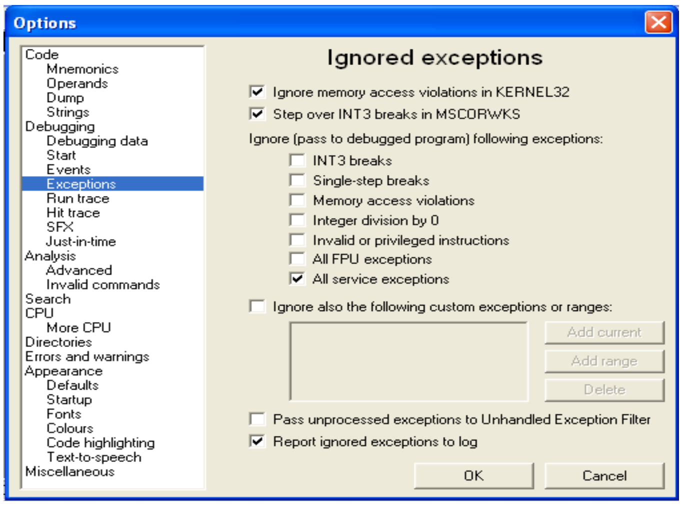
3.1.2 Do some Math!

This method is more generic and even if it is used to reveal code modifications on runtime, it can also be used for sw breakpoints detection. This can be easily implemented either by performing an addition, XOR or any other suitable operation between a constant and a byte at a specific memory address and then checking if the result is the expected one, or by using a checksum algorithm over a block of code. Since no matter how you implement sw breakpoints, by using INT3, INT1, HLT, etc... you are actually modifying the code in memory, your breakpoints will be detected.
In a few words, I am expecting to see more of this type of breakpoint detection method in the future, as it is immune to the new feature of Olly Dbg.

3.1.3 INT3 Exception!

This method aims to detect the presence of a debugger by exploiting the fact that software breakpoint exceptions are by default processed by the debugger, if a process is being debugged of course, otherwise the exception should be handled by the application itself. The iplementation is quite simple. We execute an INT3 instruction and then we check if our exception handler was triggered. If it didn't it means that something else handled the exception. I wonder what that would be... ;o)

So, let's see how Olly v2.01 behaves against this trick based on the sw bp configuration. Exceptions related settings are set to the default.



Ignored exceptions

[✓] Ignore memory access violations in KERNEL32
[✓] Step over INT3 breaks in MSCORWKS
Ignore (pass to debugged program) following exceptions:
[ ] INT3 breaks
[ ] Single-step breaks
[ ] Memory access violations
[ ] Integer division by 0
[ ] Invalid or privileged instructions
[ ] All FPU exceptions
[✓] All service exceptions
[ ] Ignore also the following custom exceptions or ranges:
[ ] Pass unprocessed exceptions to Unhandled Exception Filter
[✓] Report ignored exceptions to log

Figure 2. Default Exceptions Settings

3.1.3.1 Case 1: Exceptions (Default) / SW BP (INT1)

Using the default exceptions processing settings and INT1 for sw breakpoints the debugger behaves as expected when executing an INT3 instruction. It will stop and give the chance to the user to pass the exception to the application if required.

3.1.3.2 Case 2: Exceptions (Default) / SW BP (HLT, CLI, STI etc...)

Using the default exceptions processing settings and the new instructions for setting sw breakpoints the debugger behaves as expected when executing an INT3 instruction. It will stop and give the chance to the user to pass the exception to the application if required.

3.1.4 Single-Step Exception!

This is another debugger detection method that has been widely used. It relies on the fact that a debugger should handle itself any single step exceptions because this is necessary in order to enable one of the most fundamental features of a debugger. That is the ability to execute instructions one by one.
However, **single stepping should not be confused with stepping-over any type of instructions**. Stepping over a **CALL** instruction is not the same as stepping-over a MOV instruction. Same concept applies for instructions with **REP** prefixes which in simple words imply the number of times that instruction will be executed through the ECX register.
So, to make things clear, when we step over a CALL instruction the debugger will place a breakpoint at the instruction just after that CALL instruction. What happens in the background is that the debugger will resume the execution of that thread and it will only stop again, passing the control back to the user, when the execution returns back from that routine and its subroutines. Same concept applies when, for example, we step over a **REP MOVSB** instruction. The debugger will place a breakpoint at the next instruction waiting for the execution to arrive there. All these things happen in the background, so I hope you are not too disappointed but yes **the debugger sets breakpoints (hw or sw) without asking you**. This can complicate things when someone writes code aware of these implicit changes applied by the debugger, and uses them as a debugger detection mechanism. **To summarize the above, just keep in mind that instruction stepping does not just involve single-step exceptions.**
Going back to the main point of this section, if our application forces a single step exception by executing an INT1 instruction or by setting the Trap Flag to 1, but its exception handler is not triggered, then we know what is going on.
In Olly v2.01 we have the option of using INT1 instructions for setting sw breakpoints. This means that the debugger should treat single step exceptions as if they were breakpoint execeptions, but things might be a little bit more complicated than that and for this reason we need to perform some tests.

3.1.4.1 Case 1: Exceptions (Default) / SW BP (INT3)

During this test I triggered a single step exception using two ways. The first way was by executing directly an INT1 (0xF1) instruction, while in the second one I enabled the Trap Flag to generate a single step exception upon the execution of the next instruction.

Olly behaved as expected, correctly recognizing the exception and giving to me the chance to pass it to the application.

3.1.4.2 Case 2: Exceptions (Default) / SW BP (INT1)

During this test I managed to retrieve some nice findings. Initially, everything seemed to work as expected while running the application and Olly had the same behaviour as above, except from when I was stepping over PUSH instructions and particularly PUSH REG instructions where I had previously set a sw breakpoint (remember that we now use INT1 instead of INT3). I will talk in more detail in the next section (see: 3.1.5) about this behaviour.

3.1.5 A Single Stepping Bug

As mentioned above, there is obviously a bug that is triggered when we use INT1 for setting sw breakpoints over PUSH REG instructions (but might affect also others). Olly keeps executing the same instruction over and over either when you press F8 or F9. Olly just won't advance EIP as it should. When using the other new options for setting sw breakpoints Olly works as expected. The problem here is just functional and I don't really see any way someone could exploit this on purpose as an anti-reversing trick since setting INT1s as sw breakpoints is clearly the reverser's choice and not something that can be forced through the code that we debug. As a final thing to add to this fact, it is important to mention that the bug persists even if we choose to use hw breakpoints for stepping (Figure1).
To test this by yourself, set an INT1 breakpoint at a PUSH EBP instruction that you will commonly find at the beginning of a function (just be sure that the execution will reach that instruction) and just run until you break there. Then, just try to advance with F9 or F8.

3.1.6 Changing the type of SW Breakpoint During Debugging

During these tests I wanted to check how the new version of Olly will handle different types of sw breakpoints, and the different exception types triggered by them (breakpoint, privileged instruction, single step), all set during the same debugging session.
So, while I was testing this I noticed a strange behaviour. Olly would either update all the breakpoints to the same type when hit one of them, which could be considered normal, but in some cases the breakpoint was completely removed and even so that happened, the instruction was still marked as having an active breakpoint on it. The breakpoints list would also show that instruction, but in reality there was no breakpoint (of any type) set there anymore. These findings might not be very consistent since I did not spend much time on this, but it's up to the reader to dig more into it.
In any case, in terms of anti-reversing, this is also not something that can be used against the reverser or the debugger, since there is no good reason why someone would do something like that while debugging an application.

4. Hardware Breakpoints -Not As Usual-

This type of breakpoints is another powerful feature of the x86 CPU architecture. These breakpoints are more flexible and less invasive than then software ones. Apart from using them to stop the execution at a specific instruction, hardware breakpoints can also be used to monitor read/write access to a specific address. Furthermore, hardware breakpoints make use of a specific type of registers, the Debug Registers and in fact they don't force any code modifications in memory which makes them quite suitable for breaking in the middle of self-modifying code. The main disadvantages of this type of hardware breakpoints is that we cannot set more than four per thread. This makes sense since each thread has its own set of registers in order to keep track of the CPU state for each one of them. This state is saved in a structure called CONTEXT every time the OS needs to perform context swithing and other internal operations.
Going back to the hardware breakpoints, the addresses of these four breakpoints (per thread) are stored in the Debug Registers DR0 to DR3.
The new version of Olly offers a new feature, which allows us to use hardware breakpoints for singl e stepping instead of just using single step exceptions as most debuggers do. So in this section I am going to test this new feature in different scenarios in order to see if there are any artifacts that someone could exploit through his code in order to either detect Olly v2.01 or force some type of missbehaviour.

4.1 Fun with Hardware Breakpoints

While testing this new feature I encoutered a few interesting things that could be potentially used to create new methods to target Olly v2.01. In fact, I was kind of surprised by the new findings and I spent some time playing with them. So let's take a look.

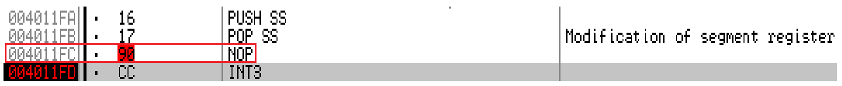
4.1.1 PUSH SS / POP SS!

Some people reading this paper will immediately recognize the trick involving these instructions, but for the sake of knowledge I will give some more detail for the people that don't know anything about it.
Basically, when stepping over a POP SS instruction you can notice in the previous version of Olly (v1.10) that the following instruction would be explicitly executed and the debugger would stop execution at the instruction after. So, if the instruction after the POP SS is a PUSHFD instruction then we could save the state of Eflags register while the Trap Flag (TF) was still enabled (set to 1) for the single step purposes. Normally the TF is disabled after the execution of each instruction while stepping, but the fact of executing the next instruction while the TF was still set it would reveal the presence of a debugger, since the TF should be disabled (set to 0).

```
Example:  push ss
          pop ss
          pushfd
          pop eax
          and eax, 0x100 ; check TF
          jnz _dbgDetected
```

Table 2. Check Trap Flag

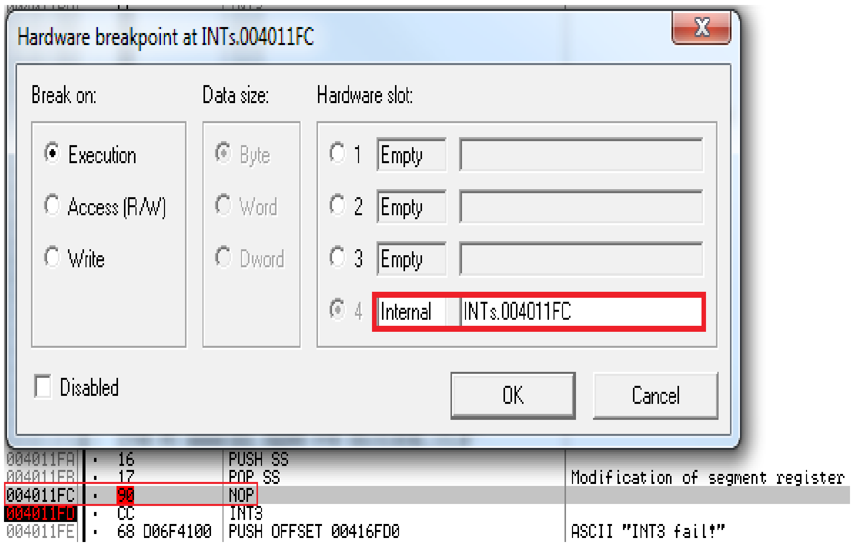
This trick is not effective anymore in Olly v2.01 so in case you are debugging code that used this trick to detect the debugger you can sleep safe, well not literally though.
However, when this trick in used while you have enabled hw breakpoints for stepping over instructions (see Figure 1), then things get a little bit weird or let's just say quite interesting.
In fact, **Olly will place an 'Internal HW Breakpoint' at the address immediately after the POP SS instruction**. This is most probably done to keep control while disabling the artifacts of this well known trick, but this time instead of leaving temporarily the TF enabled, it will forget behind an internal hw breakpoint. This means that again some artifacts are left behind, and this time permanently as the hw breakpoint will remain there until we restart the process.



```
004011FA  .  16         PUSH SS
004011FB  .  17         POP SS                    Modification of segment register
004011FC  .  90         NOP
004011FD  .  CC         INT3
```

Figure 3. Internal HW BP #1

The hw breakpoint on the NOP instruction was set by Olly.
In addition, since this is an internal hw breakpoint set by the debugging engine for its own purposes, we are not allowed to remove it.



Hardware breakpoint at INTs.004011FC

Break on:	Data size:	Hardware slot:
[✓ Execution]	[✓ Byte]	1 Empty
Access (R/W)	Word	2 Empty
Write	Dword	3 Empty
		[✓ 4] Internal INTs.004011FC

Disabled

```
004011FA  .  16            PUSH SS
004011FB  .  17            POP SS                    Modification of segment register
004011FC  .  90            NOP
004011FD  .  CC            INT3
004011FE  .  68 D06F4100   PUSH OFFSET 00416FD0      ASCII "INT3 fail!"
```

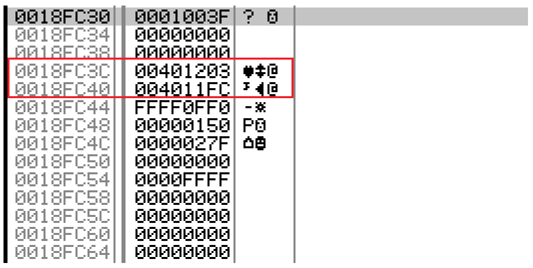
Figure 4. Internal HW BP #2

By this time, you might have been asking yourself what is that INT3 instruction doing there after the NOP instruction. Well, patience is a virtue, or at least that's what I 've heard.

4.1.2 INTs & Exceptions!

At this point I will put things together connecting some facts back to the previous case, and I will explain what that INT3 instruction is doing over there.
So, in the previous test we saw that Olly will take the initiative to set an internal hw breakpoint when stepping over a POP SS instruction wheh we use hw breakpoint for stepping. However, we didn't see if that hw breakpoint is really active in the context of that specific thread which is the most artifact in terms of anti-reversing.

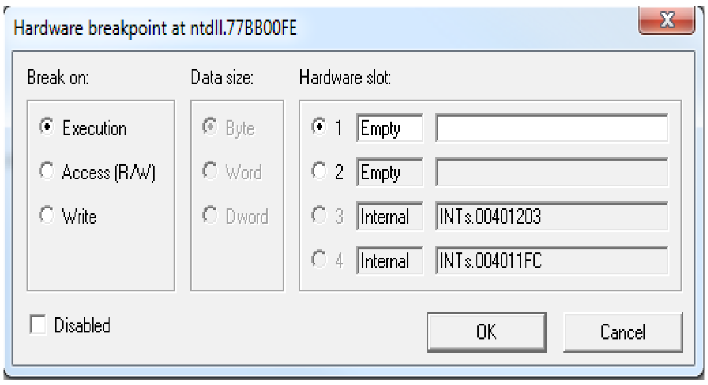
That brings us to the INT3, which I put there in order to generate an exception and check the values of the Debug Registers (DR0-DR3) through a saved context structure in the stack which keeps all the information we need about the state of the CPU for that specific thread when the exception occured.
I set a breakpoint at the **KiUserExceptionDispatcher** API and then I located the context structure in the stack.



```
0018FC30  0001003F  ? 0
0018FC34  00000000
0018FC38  00000000
0018FC3C  00401203  ♥‡@
0018FC40  004011FC  ¹◄@
0018FC44  FFFF0FF0  -☼
0018FC48  00000150  P0
0018FC4C  0000027F  ⌂☻
0018FC50  00000000
0018FC54  0000FFFF
0018FC58  00000000
0018FC5C  00000000
0018FC60  00000000
0018FC64  00000000
```

Figure 5. Thread Context

The two dwords marked in Figure 5, indicate the values of DR2 and DR3 registers. So, we see that there are now two hw breakpoints enabled, both set by Olly.



Hardware breakpoint at ntdll.77BB00FE

Break on: [✓ Execution] Access (R/W) Write

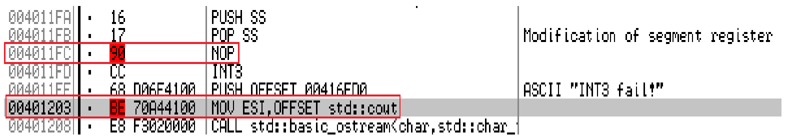
Data size: [✓ Byte] Word Dword

Hardware slot:

[✓ 1]	Empty	
2	Empty	
3	Internal	INTs.00401203
4	Internal	INTs.004011FC

Figure 6. Internal HW BP #3

If we follow in disassembler the address of the second hw breakpoint we will see that it is set after the INT3 instruction. To make things more clear, Olly will also set a hw breakpoint either after an INT instruction or after the instruction where it stopped when an exception occured.



```
004011FA  .  16           PUSH SS
004011FB  .  17           POP SS                              Modification of segment register
004011FC  .  90           NOP
004011FD  .  CC           INT3
004011FE  .  68 D06F4100  PUSH OFFSET 00416FD0                ASCII "INT3 fail!"
00401203  .  BE 70A44100  MOV ESI,OFFSET std::cout
00401208  .  E8 F3020000  CALL std::basic_ostream<char,std::char_
```

Figure 7. Internal HW BP #4

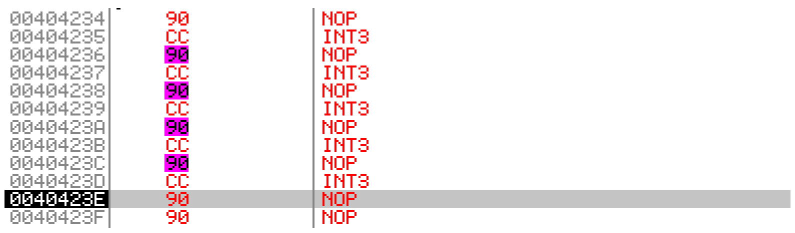
Clearly, permently enabling hw breakpoints without removing them can be used for debugger detection. As we saw this can be triggered by either executing an INT instruction or by generating an exception. Both methods can be easily implemented. An exception to this rule are software interupts that don't generate exceptions (INT 2A in 32-bit Windows). In that case Olly will set a hw breakpoint but it will remove it once the EIP points to the next instruction.

4.1.3 Adios HW Breakpoints!

Till this point we demonstrated that it is possible to trick Olly and force it to set hw breakpoints and then forget about them. So what would you say to fill all hw breakpoints slots with internal ones?

4.1.3.1 Case 1: Stepping over INT3 instructions

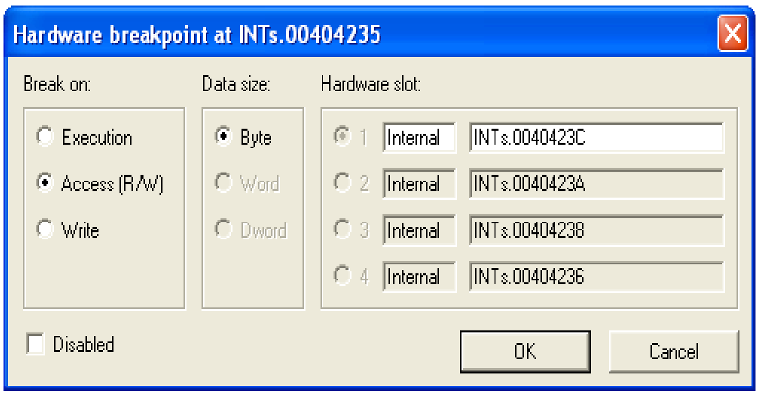
One small step for a debugger, one giant leap for hw breakpoints. For this example I just patched an application on the fly just to make a quick proof of concept. Then, I just stepped over the INT3 instructions and voilà!



```
00404234     90     NOP
00404235     CC     INT3
00404236     90     NOP
00404237     CC     INT3
00404238     90     NOP
00404239     CC     INT3
0040423A     90     NOP
0040423B     CC     INT3
0040423C     90     NOP
0040423D     CC     INT3
0040423E     90     NOP
0040423F     90     NOP
```

Figure 8. Internal HW BP #5

You probably noticed that the hw breakpoints are now indicated in purple.
That's only because I patched the code on the fly and the modified instructions are shown in red. Olly will just make this user-friendly color change for our own pleasure. Keep in mind that you can achieve the same by forcing the reverser to step over instructions that would force an exception.
Let's also have a look at the hw breakpoints table that appears when we try to set a new one.



Hardware breakpoint at INTs.00404235

Break on: Execution [✓ Access (R/W)] Write

Data size: [✓ Byte] Word Dword

Hardware slot:

[✓ 1]	Internal	INTs.0040423C
2	Internal	INTs.0040423A
3	Internal	INTs.00404238
4	Internal	INTs.00404236

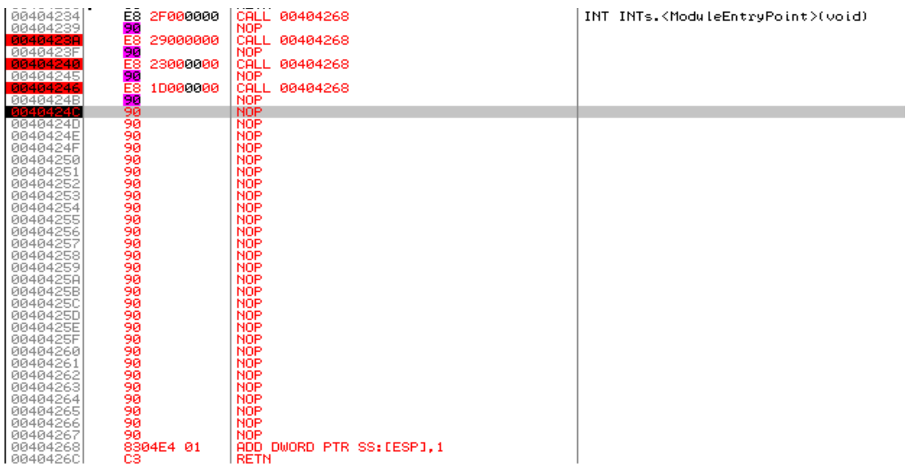
☐ Disabled

Figure 9. No HW Breakpoints

As you can see, Olly has filled the debug registers DR0-DR3 with its own breakpoints, which means that we cannot set our own during this debugging session, since as already mentioned, we are not allowed to remove internal hw breakpoints set by Olly itself.

4.1.3.2 Case2: Fake Return Addresses

As mentioned before, when we stop over a CALL instruction the debugger will implicitly set a breakpoint (usually a software one) at the instruction just after that CALL instruction in order to keep control on the execution. In case we use hw breakpoints for stepping, Olly will set an internal hw breakpoint at the return address which will then remove once the execution returns back from that call. So what if we change the return address from inside the CALL, will Olly remove that internal hw BP or will just forget about it? Again, I patched on the fly an application in order to create proof of concept.



```
00404234    E8 2F000000    CALL 00404268                    INT INTs.<ModuleEntryPoint>(void)
00404239    90             NOP
0040423A    E8 29000000    CALL 00404268
0040423F    90             NOP
00404240    E8 23000000    CALL 00404268
00404245    90             NOP
00404246    E8 1D000000    CALL 00404268
0040424B    90             NOP
0040424C    90             NOP
0040424D    90             NOP
0040424E    90             NOP
0040424F    90             NOP
00404250    90             NOP
00404251    90             NOP
00404252    90             NOP
00404253    90             NOP
00404254    90             NOP
00404255    90             NOP
00404256    90             NOP
00404257    90             NOP
00404258    90             NOP
00404259    90             NOP
0040425A    90             NOP
0040425B    90             NOP
0040425C    90             NOP
0040425D    90             NOP
0040425E    90             NOP
0040425F    90             NOP
00404260    90             NOP
00404261    90             NOP
00404262    90             NOP
00404263    90             NOP
00404264    90             NOP
00404265    90             NOP
00404266    90             NOP
00404267    90             NOP
00404268    8304E4 01      ADD DWORD PTR SS:[ESP],1
0040426C    C3             RETN
```

Figure 10. Internal HW BP #6

As you can notice, I put on purpose some software breakpoints on the real return addresses so that I wouldn't lose control over the execution. This was important in order to step over all the CALL instructions. In a real case scenario CALL instructions to functions that changes the return address (in this case EIP is only advanced by 1) should be set into multiple locations. So, again we managed to hijack all available slots for hw breakpoints.

5. Conclusion

As binary analysis tools evolve we need to be ready to face new challenges that will constantly attempt to find bugs and artifacts in these tools in order to make them less effective or trick them in any possible way. Olly debugger is without doubt a great tool, and by no means this paper is meant to underestimate or undervalue this great contribution to the community and/or its author Oleh Yushuk.
However, we need to stay alert and some times take a leap of faith before trouble finds us. Some times it's better to research about a future non existent problem rather than just for solutions in the existing ones. In fact through this research, I managed to discover an internal bug in Olly, related to the new feature regarding sw breakpoints.
Finally, I demonstrated some flaws in the new instructions stepping feature that makes use of hardware breakpoints instead of the common technique through the TF of EFLAGs register, which can be used against Olly itself either for detecting it or by eliminating the possibility from the analyst of using hardware breakpoints.
When I started writing this paper, everything was just an idea and a vague persuation of myself that

this indeed does worth the time and effort. In the end, as every creature seeks a ray of light, I really hope that together, through this paper, we also found one.