Assembly "wrapping": a technique for anti-disassembly

You can see this technique as an improvement on what is called "impossible disassembly" which you can read about in a classic: "Practical Malware Analysis", by Michael Sikorski and Andrew Honig, in chapter 15. I'll be showing some slightly more advanced examples and, of course, the technique itself, not in x86 but in x64 (more room while wrapping the real code).

In order to explain this, I'll be going through the basics till the more advanced stuff, always explaining with code examples:

- Background – Linear sweep and Recursive descent
- "Impossible disassembly" – the technique
- Example in Windows using C
- Advanced version – "jmp short -9"
- Assembly wrapping

Disclaimer: While this technique might be used for malicious purposes, I do not condone it. The only reason I look into such techniques is that 1: they are technically interesting; and 2: as part of a Red Team, you will undoubtedly be developing your own tool arsenal and, as such, end up going deep into reverse engineering (RE) and implementing anti-RE techniques.

Credit: Johannes Kinder, to my knowledge, came up with this originally in a 2010 paper using x86 to explain the concept he called "overlapping instructions" [http://infoscience.epfl.ch/record/167546/files/thesis.pdf]. I only have a new name for it, because I wrote the whole article thinking I came up with it, till this was brought to my attention. However, I'll be going deeper with practical examples, well-known disassemblers' tests, and using x64.

The C code will be compiled with mingw-w64 compiler, and if you are wondering "why aren't you using Visual Studio?", that would be because they (Microsoft) don't support inline assembly for x64 architecture [https://docs.microsoft.com/en-us/cpp/assembler/inline/inline-assembler?view=vs-2019]. Just to be clear, this is not an incapability issue, it's actually a choice they made for better code compiling optimization.

Also, the whole point of this blog is to show a specific anti-disassembly technique in disassemblers. But not only in the purest of forms, such as in tools like objdump or IDA, but also the disassemblers inside debuggers (e.g. x64dbg). So, while I would not intentionally compare apples and oranges, when I do mention debuggers, what I actually mean is the disassembler inside them. Also, while I do use Immunity quite a lot, it doesn't support 64-bit PE files, so it won't be featured here.

Background – Linear sweep and Recursive descent

As a beginner in RE, one tends to assume that the disassembled code shown by disassemblers (e.g. IDA, objdump) or debuggers (e.g. x64dbg, gdb) is definitive. However, that is not true, as proven by the fact that most of these tools allow the researcher to rearrange the code/data analysis.

Bottom line, the linear sweep starts from entry point / start of function and just runs through the opcodes assuming everything is code and all linear (one instruction starts after the other), while the recursive descent is smarter and follows the control flow to better distinguish between code and data. However, there's still linear sweep when doing recursive descent, as it only stops doing the linear sweep when finding control flow instructions such as conditional jumps, absolute jumps, calls, and returns.

As an example, check the following code, with data in between. It's not relevant here to understand all instructions, but rather the fact that there's data ("buffer db …") in between the code:

```
global _start
section .text

_start:
mov rax,1
mov rdi,1
jmp short next

buffer db "Australia",10
len equ $-buffer

next:
mov rsi,buffer
mov rdx,len
syscall
mov rax,60
mov rdi,0
syscall
```

source code in assembly x64

This simply prints out "Australia" in the console. If you want to know more about this code, I've written extensively about shellcoding (even though this is not shellcoding as it has null bytes and absolute references to the buffer memory position) in previous blogs such as https://pentesterslife.blog/2017/11/01/x86_64-tcp-bind-shellcode-with-basic-authentication-on-linux-systems/.

```
vagrant@precise64:/vagrant$ vi z.nasm
vagrant@precise64:/vagrant$ nasm -felf64 z.nasm -o z.o && ld z.o -o z
vagrant@precise64:/vagrant$ ./z
Australia
```

compilation + execution

The point here is to show objdump and gdb doing a linear sweep, and IDA using the recursive descent to analyze the code, so let's see how each distinguish between code and data. Note that the "len" line will not show up as it will simply be calculated and replaced in the relevant locations – mov rdx,len – by the compiler.

```
vagrant@precise64:/vagrant$ objdump -d -M intel z

z:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:       48 b8 01 00 00 00 00    movabs rax,0x1
  400087:       00 00 00
  40008a:       48 bf 01 00 00 00 00    movabs rdi,0x1
  400091:       00 00 00
  400094:       eb 0a                   jmp    4000a0 <a>

0000000000400096 <buffer>:
  400096:       41 75 73                rex.B jne 40010c <a+0x6c>
  400099:       74 72                   je     40010d <a+0x6d>
  40009b:       61                      (bad)
  40009c:       6c                      ins    BYTE PTR es:[rdi],dx
  40009d:       69 61 0a 48 be 96 00    imul   esp,DWORD PTR [rcx+0xa],0x96be48

00000000004000a0 <a>:
  4000a0:       48 be 96 00 40 00 00    movabs rsi,0x400096
  4000a7:       00 00 00
  4000aa:       48 ba 0a 00 00 00 00    movabs rdx,0xa
  4000b1:       00 00 00
  4000b4:       0f 05                   syscall
  4000b6:       48 b8 3c 00 00 00 00    movabs rax,0x3c
  4000bd:       00 00 00
  4000c0:       48 bf 00 00 00 00 00    movabs rdi,0x0
  4000c7:       00 00 00
  4000ca:       0f 05                   syscall
vagrant@precise64:/vagrant$
```

linear sweep with objdump

```
vagrant@precise64:/vagrant$ gdb -q
(gdb) set disassembly-flavor inte
(gdb) file z
Reading symbols from /vagrant/z...(no debugging symbols found)...done.
(gdb) disassemble _start,+78
Dump of assembler code from 0x400080 to 0x4000ce:
   0x0000000000400080 <_start+0>:       movabs rax,0x1
   0x000000000040008a <_start+10>:      movabs rdi,0x1
   0x0000000000400094 <_start+20>:      jmp    0x4000a0 <a>
   0x0000000000400096 <buffer+0>:       rex.B jne 0x40010c
   0x0000000000400099 <buffer+3>:       je     0x40010d
   0x000000000040009b <buffer+5>:       (bad)
   0x000000000040009c <buffer+6>:       ins    BYTE PTR es:[rdi],dx
   0x000000000040009d <buffer+7>:       imul   esp,DWORD PTR [rcx+0xa],0x96be48
   0x00000000004000a4 <a+4>:    add    BYTE PTR [rax],al
   0x00000000004000a7 <a+7>:    add    BYTE PTR [rax],al
   0x00000000004000a9 <a+9>:    add    BYTE PTR [rax-0x46],cl
   0x00000000004000ac <a+12>:   or     al,BYTE PTR [rax]
   0x00000000004000ae <a+14>:   add    BYTE PTR [rax],al
   0x00000000004000b0 <a+16>:   add    BYTE PTR [rax],al
   0x00000000004000b2 <a+18>:   add    BYTE PTR [rax],al
   0x00000000004000b4 <a+20>:   syscall
   0x00000000004000b6 <a+22>:   movabs rax,0x3c
   0x00000000004000c0 <a+32>:   movabs rdi,0x0
   0x00000000004000ca <a+42>:   syscall
```

linear sweep with gdb

recursive descent with IDA

You can clearly tell that the recursive descent (done by IDA here) is better at telling the difference between code and data.

I’d definitely recommend reading chapter 1 of “IDA Pro book”, 2nd Edition, or chapter 15 on “Practical Malware Analysis”, chapter 15 for more on these algorithms.

“Impossible disassembly” – the technique

By understanding the algorithms previously mentioned, one can imagine there are a few ways to exploit the disassembly process. One such way is called impossible disassembly. The name is a bit unfortunate because it’s not actually impossible, as it is only referenced that way because of the predicament in which the disassembler will find itself: one byte belonging to two instructions.

In the following image, the disassembly you see above the hex bytes is what the disassembler will show you or “see”, but the instructions below is what it turns out to be (after the jump is made).

```
         jmp -1   | <nonsense code disassembled>
      _ ____ ____|____ _
       |    |    |    |
   ... | eb | ff | c0 | ...
      _|____|____|____|_
            |         |
            | inc eax | <actual "hidden" code>
```

jmp -1 algorithm

The difficulty here (and hence the “impossible”) is a basic assumption in disassembly, which states that one byte is only interpreted in the context of one instruction. This is obviously not true, as shown above where 0xff belongs to two instructions.

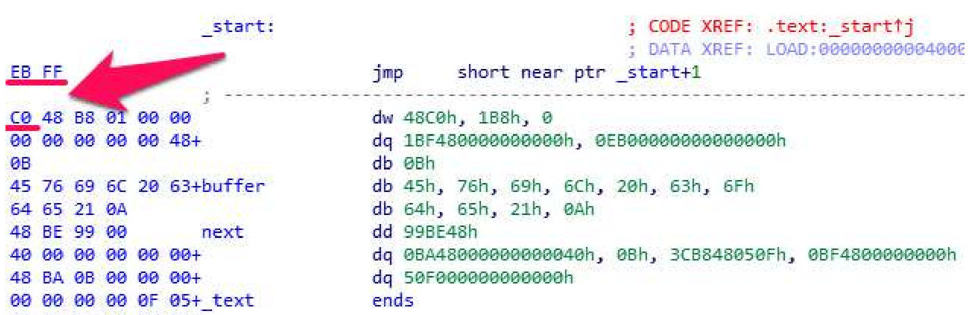
Note that, after the jump, there is an increment to eax. Make sure this does not impact your hidden code! If you were messing with eax and were expecting it to be a specific value, this will change it of course. Also, you can swap the 0xc0 for something else, but you have to keep in mind that this new byte has to be the start of an instruction that will consume some of the next bytes of “real code” and only partially.

So, let’s see it in action! Using the previous assembly example, but this time printing out “Evil code!”, which will be what we’re trying to hide from the disassembler.

Let’s inject the anti-disassembly code right at the beginning of the _start entry point:

source code in assembly x64

After compiled with nasm, you can see the code is hidden, even with recursive descent:



IDA

And this runs just as before:

```
vagrant@precise64:/vagrant$ ./z
Evil code!
vagrant@precise64:/vagrant$
```

execution

Keep in mind that, as I've pointed out before, a reverse engineer can instruct IDA to interpret bytes/opcodes as data (pressing 'D') or as code (pressing 'C') after identifying this technique. So, this won't stop any decent reverse engineer but might slow them down. And you can slow them down even more with the advanced variation of this that we'll look at ahead.

Also, the "inc eax" (ff c0 executed after the jump -1) is irrelevant in this example, given that I set rax to 1 right after.

Example in Windows using C

Why not show a similar example but with assembly, in Windows? Because the concept is exactly the same. The only actual difference is that instead of doing the "db …" (stands for define byte) in the beginning (define byte), you'd be doing a ".byte 0xeb,0xff,0xc0" (different compilers…).

So, for our proof-of-concept, let's use the following, and see if we can hide the "evil" part from disassemblers:

```c
#include<stdio.h>
void main(){
    printf("hello world\n");
    __asm__(".byte 0xeb,0xff,0xc0");
    printf("evil code\n");
}
```

source code in C

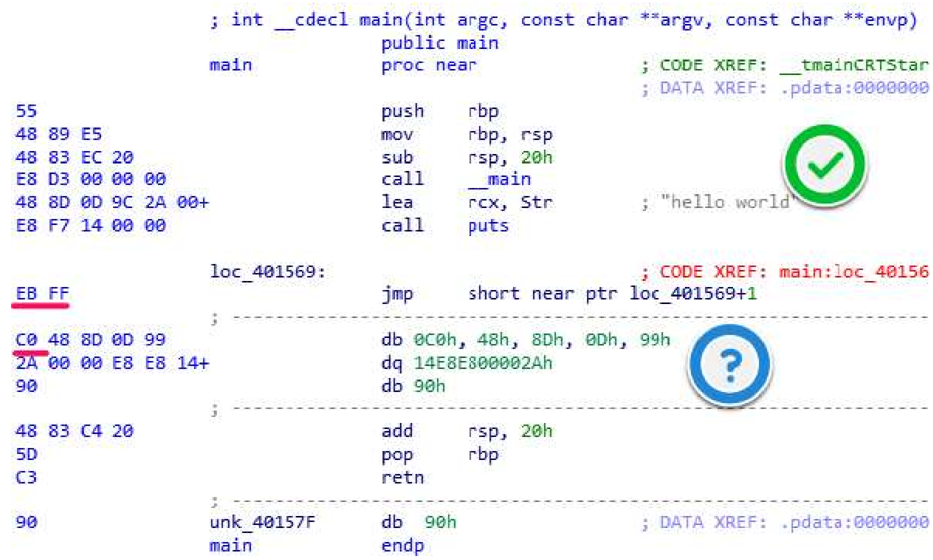
Which then executes into the following, where you can see the "evil" code being executed:

```
Windows Command Processor

C:\Users\alima\Desktop>x86_64-w64-mingw32-gcc.exe -m64 -masm=intel c:\Users\alima\Desktop\test.c -o test.exe

C:\Users\alima\Desktop>test.exe
hello world
evil code

C:\Users\alima\Desktop>
```

compilation + execution

But when looking at it, using different tools, they can’t disassembly it right, at first – again, you can do this manually in both the following tools, but it requires extra work in your RE.

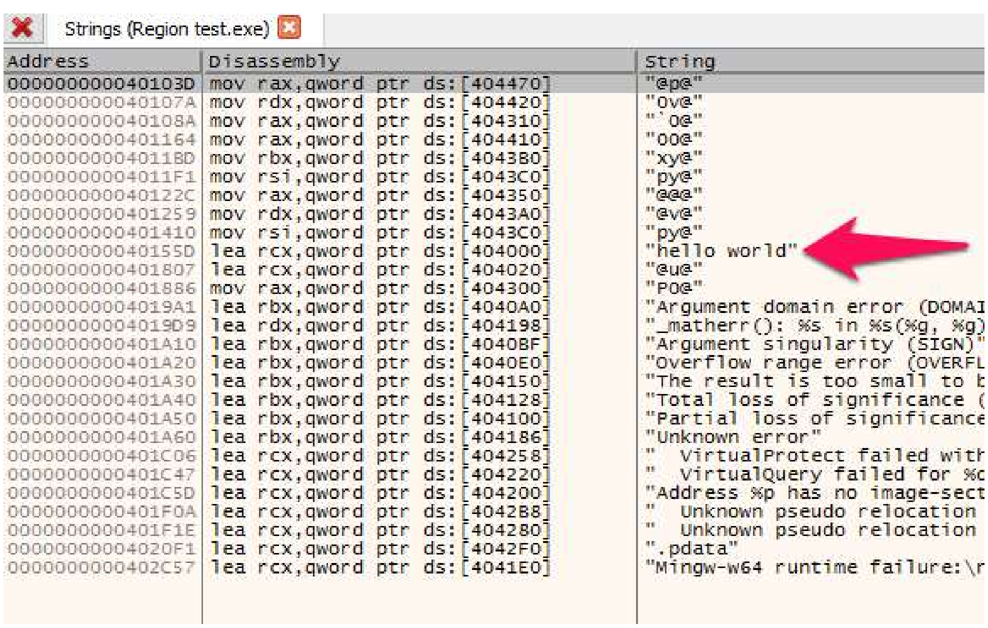
So, let’s look at IDA:



IDA

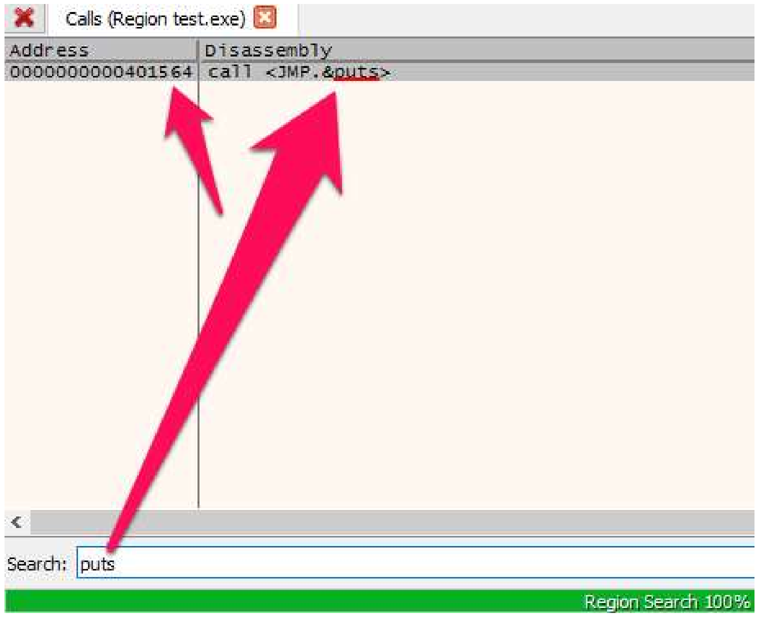
The first “printf” (“puts”) is clearly shown but not the second “evil” code. So, the added 3 bytes do exactly what they are supposed to do.

In x64dbg, it compromises its analysis as well. A common task in RE is to search for string references and intermodular calls, and in this case, the “evil code” doesn’t show up and, in looking for intermodular calls, it only shows one occurrence of the “puts” function:


Strings (Region test.exe)

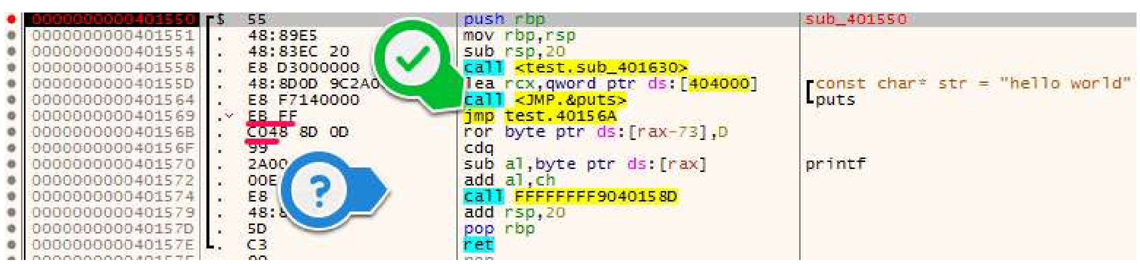
Address	Disassembly	String
000000000040103D	mov rax,qword ptr ds:[404470]	"@p@"
000000000040107A	mov rdx,qword ptr ds:[404420]	"Ov@"
000000000040108A	mov rax,qword ptr ds:[404310]	"`O@"
0000000000401164	mov rax,qword ptr ds:[404410]	"OO@"
00000000004011BD	mov rbx,qword ptr ds:[4043B0]	"xy@"
00000000004011F1	mov rsi,qword ptr ds:[4043C0]	"py@"
000000000040122C	mov rax,qword ptr ds:[404350]	"@@@"
0000000000401259	mov rdx,qword ptr ds:[4043A0]	"@v@"
0000000000401410	mov rsi,qword ptr ds:[4043C0]	"py@"
000000000040155D	lea rcx,qword ptr ds:[404000]	"hello world"
0000000000401807	lea rcx,qword ptr ds:[404020]	"@u@"
0000000000401886	mov rax,qword ptr ds:[404300]	"PO@"
00000000004019A1	lea rbx,qword ptr ds:[4040A0]	"Argument domain error (DOMAI
00000000004019D9	lea rdx,qword ptr ds:[404198]	"_matherr(): %s in %s(%g, %g)
0000000000401A10	lea rbx,qword ptr ds:[4040BF]	"Argument singularity (SIGN)"
0000000000401A20	lea rbx,qword ptr ds:[4040E0]	"Overflow range error (OVERFL
0000000000401A30	lea rbx,qword ptr ds:[404150]	"The result is too small to b
0000000000401A40	lea rbx,qword ptr ds:[404128]	"Total loss of significance (
0000000000401A50	lea rbx,qword ptr ds:[404100]	"Partial loss of significance
0000000000401A60	lea rbx,qword ptr ds:[404186]	"Unknown error"
0000000000401C06	lea rcx,qword ptr ds:[404258]	" VirtualProtect failed with
0000000000401C47	lea rcx,qword ptr ds:[404220]	" VirtualQuery failed for %c
0000000000401C5D	lea rcx,qword ptr ds:[404200]	"Address %p has no image-sect
0000000000401F0A	lea rcx,qword ptr ds:[4042B8]	" Unknown pseudo relocation
0000000000401F1E	lea rcx,qword ptr ds:[404280]	" Unknown pseudo relocation
00000000004020F1	lea rcx,qword ptr ds:[4042F0]	".pdata"
0000000000402C57	lea rcx,qword ptr ds:[4041E0]	"Mingw-w64 runtime failure:\r

x64dbg: only “hello world” shown



x64dbg: only one “puts” shown

And, of course, the disassembled code:



x64dbg: disassembled code

Advanced version: “jmp short -9”

Now, just in case you’re wondering if one couldn’t simply write down a script (e.g. IDAPython) and patch that “eb ff c0” sequence by removing them, think again. While that would solve this specific problem, it wouldn’t solve the million variations that you can come up with. Also, after the CPU does the jump, it’ll execute “inc eax”, and to avoid complexity, we neglected that instruction. But one could write code right after, that would depend on that increment, or validate eax to a specific/expected value. So, you can see there’s no universal solution here. Moreover, what if we don’t just jump back one byte (jmp short -1 | eb ff) but, instead, jump 9 bytes back?

Let’s see such an example:

```
      mov rax,0xffff08ebffffffff                 |  xor rax,rax | jmp -9  | <nonsense code disassembled>
    _ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____|____ ____ ____|____ ____|____ _
     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
... | 48 | b8 | ff | ff | ff | ff | eb | 08 | ff | ff | 48 | 31 | c0 | eb | f7 | e8 | ...
    _|____|____|____|____|____|____|____|____|____|____|____|____|____|____|____|____|_
                                    |         |                                  |
                                    | jmp +8  |                                  | <actual "hidden" code>
```

jmp -9 algorithm

To understand this image, read the code as disassembled above the hex code, and then after the "jmp" backwards (-9), read the code below the hex code, which will be hidden from the disassembler.

What you'll see in the disassembler is:

```
mov rax,0x…
xor rax,rax
jmp -9
call …
```

disassembled code

The jump (right before the last instruction call) will go backwards and land the CPU (rip) right in the "middle" of the data I first put into rax, in the eb 08 part to be more precise, which is another jmp but ahead, right after the e8. The e8 is important here because it's the start of a call instruction and will consume the next bytes as a memory address (as function/code memory location) and will, therefore, hide the instructions that those bytes actually represent. And that's why the "jmp" ahead (eb 08) lands right after the e8 (fake call).

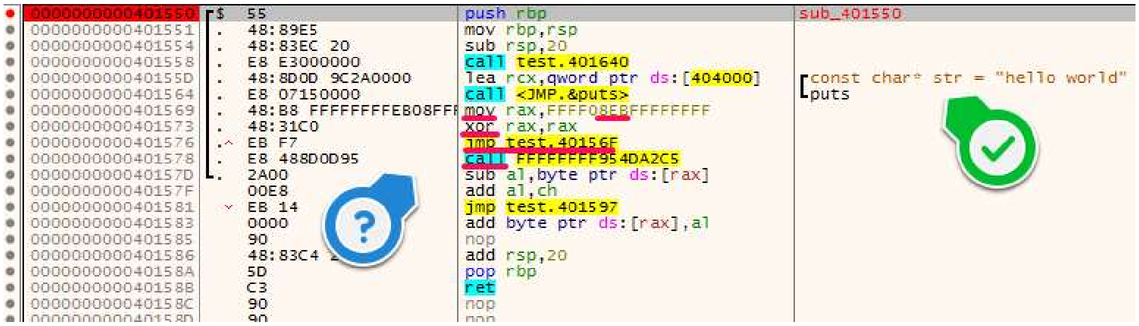
Let's see it working then:

```c
#include<stdio.h>
void main(){
    printf("advanced hello world\n");
    __asm__(".byte 0x48, 0xb8, 0xff, 0xff, 0xff, 0xff, 0xeb, 0x08, 0xff, 0xff, 0x48, 0x31, 0xc0, 0xeb, 0xf7, 0xe8");
    printf("advanced evil code\n");
}
```

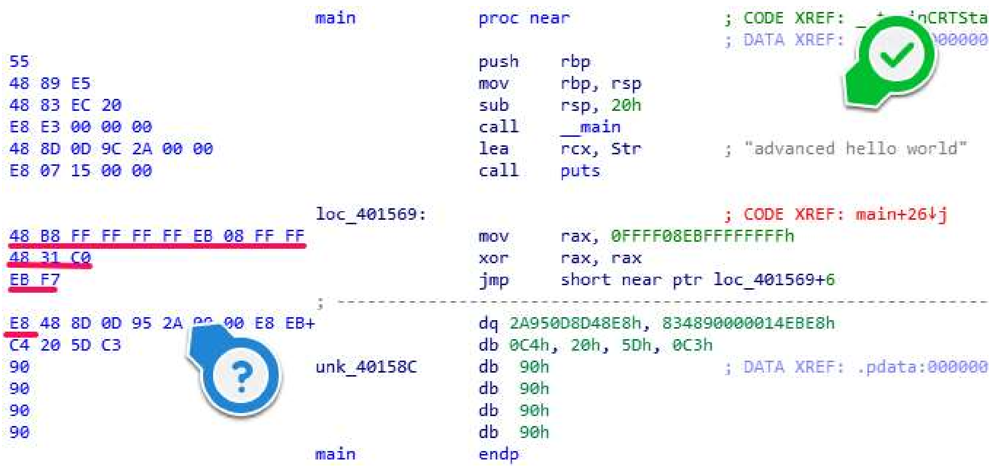
C code

```
Windows Command Processor
C:\Users\alima\Desktop>x86_64-w64-mingw32-gcc.exe -m64 -masm=intel c:\Users\alima\Desktop\test.c -o test.exe

C:\Users\alima\Desktop>test.exe
advanced hello world
advanced evil code

C:\Users\alima\Desktop>
```

compiled + executed



x64dbg



IDA

The other interesting thing is that, because of the nature of this code, the ff's are quite irrelevant. So you can replace them with any other bytes and it would still work. Ideally, if you have different variations on the same executable, it would make it harder to automate detection and elimination, through scripting, of these byte sequences in the code.

Assembly wrapping

So the previous code has something interesting about it: it jumps back into a large instruction (mov rax,…) and executes code that is inside the value you're putting into rax. The ff's are quite irrelevant but, what if they weren't? What if I could build a "skeleton" code where I could then place hidden code instead of the ff's? This is what I came up with:

vagrant@precise64: /vagrant

```
global _start
section .text
_start:
mov rax,0x04ebffffffffffff
jmp short -9
mov rax,0x02ebffffffffffff
mov rax,0x02ebffffffffffff
mov rax,0x02ebffffffffffff
mov rax,0x02ebffffffffffff
mov rax,0x02ebffffffffffff
mov rax,0x02ebffffffffffff
mov rax,0x02ebffffffffffff
mov rax,0x02ebffffffffffff
mov rax,0x02ebffffffffffff
mov rax,0x02ebffffffffffff
mov rax,0x02ebffffffffffff
~
~
```

skeleton.nasm

The hidden code execution will be triggered by the jmp instruction. The values are reversed (little-endian), but what's happening here is a jump back to the first byte of the 8-byte value placed in rax, which will be (in the correct order) "ff ff ff ff ff ff eb 04". Now, all ff's will be replaced with my real/hidden code which will be executed, and then the "eb 04" is simply a jump ahead into the start of the next 8-byte value placed into the next "mov rax,…", which will again execute the code that will be placed there until it reaches the "eb 02" which, again, jumps ahead into the next hidden instruction.

Before writing up code to be hidden inside this skeleton, we must acknowledge some limitations:

- Given the fact that I chose the "mov rax,…" (a 10-byte instruction) as my "skeleton", none of the hidden code's instructions must be longer than 6 bytes. This is because the bytes/opcodes on a single instruction must be placed right next to each other when being read by the CPU, and I only have 6 available, given that "mov rax,…" is made of 2 bytes that identify the instruction, and 8 bytes to put in the register. I still need to take on 2 of these 8 bytes for the "jmp short 2 / eb 02", so I'm left with 6 bytes to play with. However, I can still join instructions together, as long as the total number of bytes doesn't exceed the number 6. And you'll also notice that I sometimes have to "pad" (encryption term) the instructions when they're shorter than 6 bytes with NOPs (0x90), otherwise, the compiler nasm will have null bytes appended to the higher end of the 8-byte value.

- Given the previous point, you can now understand why I can't write this in C, as you'll definitely have the C compiler throw assembly code with instructions longer than 6 bytes at you. So I need full control on writing the assembly, which forces me to write it myself.

I'll mention the advantages after the example as you'll understand my point better.

So the code we wish to hide is the following:

vagrant@precise64: /vagrant

```
global _start
section .text
_start:
push 1
pop rax
push rax
pop rdi
mov esi,0x0a6c6976
rol rsi,8
xor rsi,0x45
push rsi
push rsp
pop rsi
push 5
pop rdx
syscall
push 60
pop rax
xor rdi,rdi
syscall
~
```

code.nasm

This is not as simple or straight-forward as the previous assembly codes I've shown, because this is more like actual shellcode, while still just printing something – "Evil\n" – out on the command line. I chose to do this, this way because I want to:

- have as short instructions as possible to fit the most inside those 6 bytes, in a single "mov rax,…".
- have no null bytes, again to save on space.
- need position-independent code as absolute memory positions will change once placed inside the skeleton code.

These are all characteristics of shellcode, which I've written extensively about in my previous blog [https://pentesterslife.blog/] so I won't delve into the details of the code, but suffice to say it simply prints out "Evil\n":

vagrant@precise64: /vagrant

```
vagrant@precise64:/vagrant$ vi code.nasm
vagrant@precise64:/vagrant$ nasm -felf64 code.nasm -o code.o && ld code.o -o code
vagrant@precise64:/vagrant$ ./code
Evil
vagrant@precise64:/vagrant$
```

compilation + execution of hidden code to produce opcodes

The executable has the following opcode:

```
6a 01                   push   0x1
58                      pop    rax
50                      push   rax
5f                      pop    rdi
be 76 69 6c 0a          mov    esi,0xa6c6976
48 c1 c6 08             rol    rsi,0x8
48 83 f6 45             xor    rsi,0x45
56                      push   rsi
54                      push   rsp
5e                      pop    rsi
6a 05                   push   0x5
5a                      pop    rdx
0f 05                   syscall
6a 3c                   push   0x3c
58                      pop    rax
48 31 ff                xor    rdi,rdi
0f 05                   syscall
```

objdump with opcode of compiled code.nasm

Notice that there are, as in previous examples, two syscalls: the write to stdout file descriptor and the exit (process). Without this last one, the process breaks (rip is incremented out of the .text memory section and tries to execute data in memory where it has no permissions to execute) and you’ll see an error being shown.

```
pop rdx
syscall
# push 60
# pop rax
# xor rdi,rdi
# syscall
~
```

exit syscall commented out

```
vagrant@precise64:/vagrant$ vi code.
vagrant@precise64:/vagrant$ nasm -fe
vagrant@precise64:/vagrant$ ./code
Evil
Segmentation fault
```

segmentation fault due to not properly exiting the process

This is interesting because, as you’ll see ahead, the skeleton itself doesn’t properly exit the process, so it should crash. However, it doesn’t actually crash, because the actual code it’ll be executing does exit properly.

```
vagrant@precise64: /vagrant
global _start
section .text
_start:
mov rax,0x04eb905f5058016a
jmp short -9
mov rax,0x02eb900a6c6976be
mov rax,0x02eb909008c6c148
mov rax,0x02eb909045f68348
mov rax,0x02eb5a056a5e5456
mov rax,0x02eb90583c6a050f
mov rax,0x02eb90050fff3148
~
```

final skeleton code

```
vagrant@precise64: /vagrant
vagrant@precise64:/vagrant$ vi skeleton.nasm
vagrant@precise64:/vagrant$ nasm -felf64 skeleton.nasm -o skeleton.o && ld skeleton.o -o skeleton
vagrant@precise64:/vagrant$ ./skeleton
Evil
vagrant@precise64:/vagrant$
```

hidden code executed from within the skeleton

Success!! And the disassemblers will simply show the skeleton and not the hidden code:

```
48 b8 6a 01 58 50 5f    movabs rax,0x4eb905f5058016a
90 eb 04
eb f6                   jmp    400082 <_start+0x2>
48 b8 be 76 69 6c 0a    movabs rax,0x2eb900a6c6976be
90 eb 02
48 b8 48 c1 c6 08 90    movabs rax,0x2eb909008c6c148
90 eb 02
48 b8 48 83 f6 45 90    movabs rax,0x2eb909045f68348
90 eb 02
48 b8 56 54 5e 6a 05    movabs rax,0x2eb5a056a5e5456
5a eb 02
48 b8 0f 05 6a 3c 58    movabs rax,0x2eb90583c6a050f
90 eb 02
48 b8 48 31 ff 0f 05    movabs rax,0x2eb90050fff3148
90 eb 02
```

disassembly by objdump

```
_start:                                 ; CODE XRE
                                        ; DATA XRE
                mov     rax, 4EB905F5058016Ah
                jmp     short near ptr _start+2
; ---------------------------------------------------
                mov     rax, 2EB900A6C6976BEh
                mov     rax, 2EB909008C6C148h
                mov     rax, 2EB909045F68348h
                mov     rax, 2EB5A056A5E5456h
                mov     rax, 2EB90583C6A050Fh
                mov     rax, 2EB90050FFF3148h
_text           ends
```

disassembly by IDA

Now while this specific example is very easy to recognize as an anti-disassembly technique (a first mov rax,… then a jmp -x, and a never-ending sequence of mov rax,…), you have to consider its flexibility. If you spread the movs further (even though no longer than 256 bytes as per the relative jump: “jmp short”) and place other code (that will simply never be executed) in between, you can make this look a lot like something else completely benign, which could be a huge advantage in hiding the real code. Another advantage is the fact that this is a pain to manually instruct the disassembler on how to interpret the code/data. So you’d have to end up writing some plugin to help you if the hidden code is large (albeit shellcode), which could be very tricky if you think about the fact that you’ll have to automate the distinction between real “mov rax,…” instructions and the “wrappers”.

Also, you can choose longer instructions as wrappers, which will give you more space, per line/instruction to fit in your hidden code.

So, there you go. I hope you found it as interesting as I did.