HELLO ANTI DISASSEMBLY

Anti-Disassembly techniques and their mitigations

Contents

List of figures

The workshop description

The goal of the workshop is a short introduction to anti-disassembly techniques. We will review how the two main types of disassembler works, and why they can be fooled, then we will cover 3 typical techniques. As part of each exercise we will create our own short C code, which will cause the disassembler to incorrectly parse our code, then we will see how we can manually find and correct it in IDA Pro. As a last step we will create a short Python script for IDA Pro, which will automatically find and mark these techniques for us. We will also check how we can patch the code from an IDA Script to defeat the anti-disassembly techniques.

Requirements

- Dev-C++ http://sourceforge.net/projects/orwelldevcpp/
- Python https://www.python.org/
- Ida Pro Demo 6.8 https://www.hex-rays.com/products/ida/support/download_demo.shtml
- Idapython 1.7.1 https://code.google.com/p/idapython/wiki/Downloads

Setting up the environment

Configuring IDA Pro 6.8

After downloading and installing IDA Pro 6.8, we need to download idapython, and extract it to the IDA Pro installation directory. This will enable us to use Python scripts in IDA.
To see the opcodes we need to set it at Options -> General:

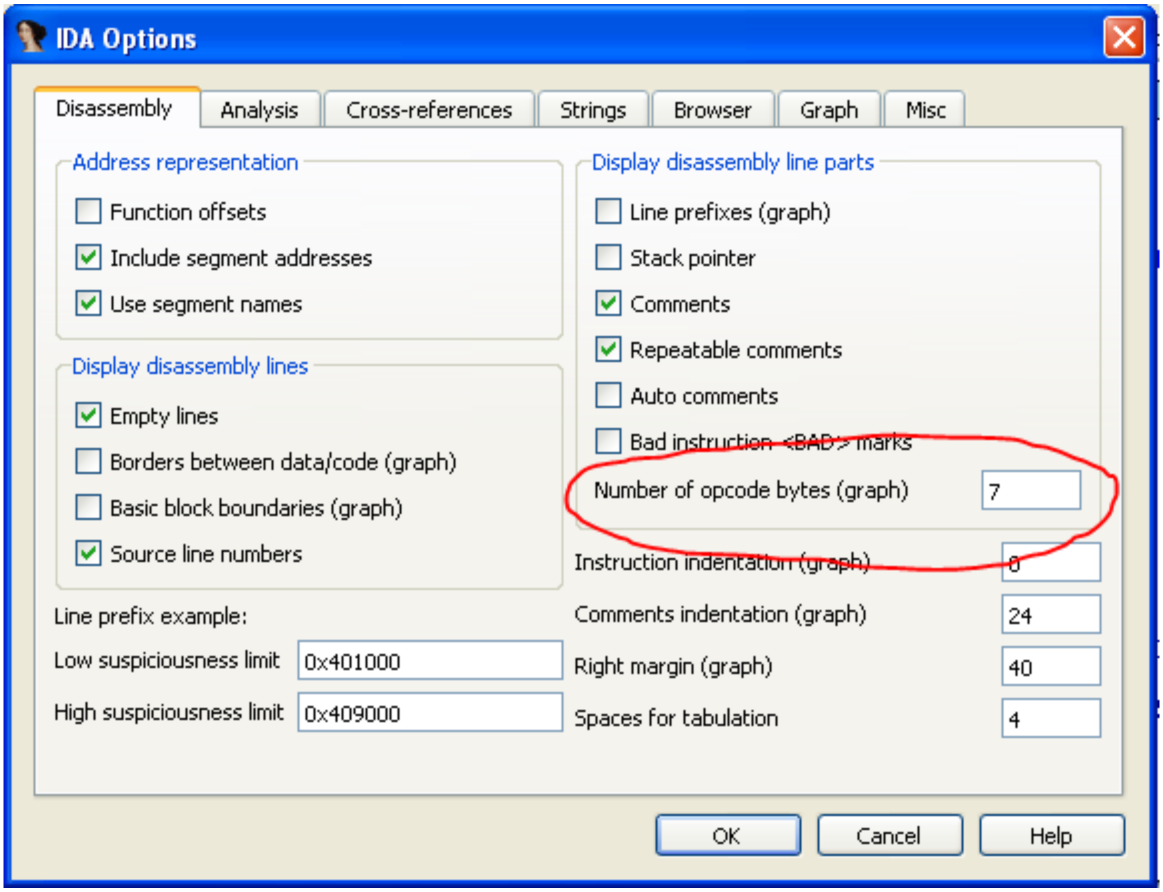


Figure 1: IDA Options

This has to be done each time we open IDA Pro.

Configuring Dev-C++

After installing Dev-C++ we need to tell it that we will use assembly with Intel syntax. This can be done in Tools -> Compiler Options, with adding **`-masm=intel`** on the General tab, and selecting "32-bit Release" at "Compiler set to configure" drop down list.

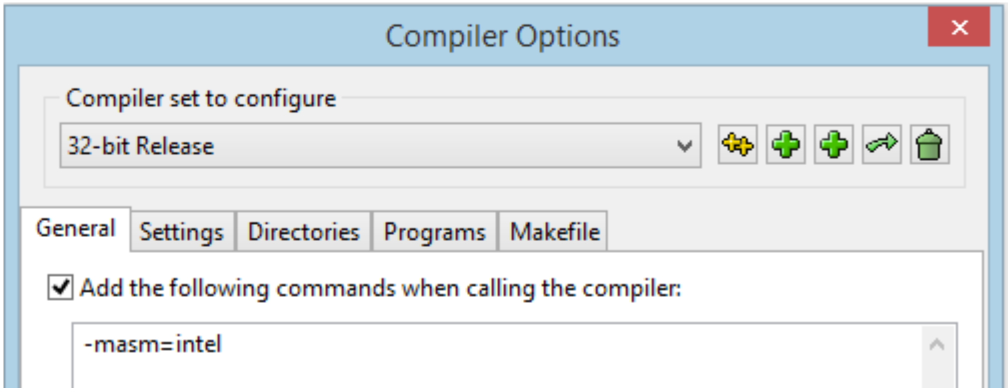


Figure 2: Dev-C++ Compiler Options

IDA Pro Python functions

Here is a quick review of the specific IDA Pro Python functions that we will use in our scripts.

Source: https://www.hex-rays.com/products/ida/support/idapython_docs/

Heads(start=None, end=None)

Get a list of heads (instructions or data).

Parameters:

start - start address (default: inf.minEA)
end - end address (default: inf.maxEA)

Returns:

list of heads between start and end

GetMnem(ea)

Get instruction mnemonics.

Parameters:

ea - linear address of instruction

Returns:

"" - no instruction at the specified location

Note: this function may not return exactly the same mnemonics as you see on the screen.

Message(msg)

Display a message in the message window.

Parameters:

msg - message to print (formatting is done in Python)

This function can be used to debug IDC scripts

GetOpnd(ea,n)

Get operand of an instruction.

Parameters:

ea - linear address of instruction

n - number of operand: 0 - the first operand 1 - the second operand

Returns:

the current text representation of operand or ""

FindBinary(ea, flag, searchstr, radix=16)

Parameters:

ea - start address

flag - combination of SEARCH_* flags

searchstr - a string as a user enters it for Search Text in Core

radix - radix of the numbers (default=16)

Returns:

ea of result or BADADDR if not found

Note: Example: "41 42" - find 2 bytes 41h,42h (radix is 16)

Flags:

SEARCH_UP = 0
SEARCH_DOWN = 1
SEARCH_NEXT = 2
SEARCH_CASE = 4
SEARCH_REGEX = 8
SEARCH_NOBRK = 16
SEARCH_NOSHOW = 32

SegStart(ea)

Get start address of a segment

Parameters:

ea - any address in the segment

Returns:

start of segment BADADDR - the specified address doesn't belong to any segment

SetColor(ea, what, color)

Set item color.

Parameters:

ea - address of the item

what - type of the item (one of CIC_* constants)

color - new color code in RGB (hex 0xBBGGRR)

Returns:

success (True or False)

CIC_* constants:
CIC_ITEM = 1
CIC_FUNC = 2
CIC_SEGM = 3

Quick overview of disassembler methods

The next section will cover the basics of the two typical disassembler methods: linear and flow oriented.

Linear disassembly

This technique is the simplest one. The disassembler will go through the code, byte by byte in a linear way, and will try to translate each opcode to assembly instruction, it will calculate the length of it and the next instruction will begin where the previous one ended. This strategy is commonly used by debuggers.

The problem with this strategy is that it doesn't consider the program flow, and if there are data segments (e.g.: jump table, constant data) in the code, it will try to interpret those as opcodes, and thus it will break the assembly comes after.

Flow oriented disassembly

Flow oriented disassemblers (like IDA Pro) will interpret the various instructions, and based on the flow controls (JMP, CALL, RET, etc…) will build a table of locations, which needs to be disassembled, and will only do those parts, thus if we have data in the middle of the code, which the program flow will never reach, it won't be disassembled.

When going through a list of the places identified, the program has to choice which part to disassemble first. In compiler generated code, this shouldn't make a difference, but with hand written assembly (what we will also do below), these selections can be utilized to break the assembled code. Two examples:

1. If there is a conditional jump (e.g.: JZ/JNZ), IDA Pro will first disassemble to false branch
2. In case of CALL instruction, IDA Pro will disassemble the instructions after the CALL, and only later the called function location

Anti-disassembly techniques

Initial C code

We will use the following C code during the workshop to demonstrate the various techniques. The code simple prints two strings to the standard output. We will place our bogus instruction between the two printf calls, which will cause the 2nd one to disappear in disassembly.

```
#include <stdio.h>

void main()
      {
            printf("Hello, World!");
            printf("Not seen");
      }
```

This is how it looks in IDA Pro:

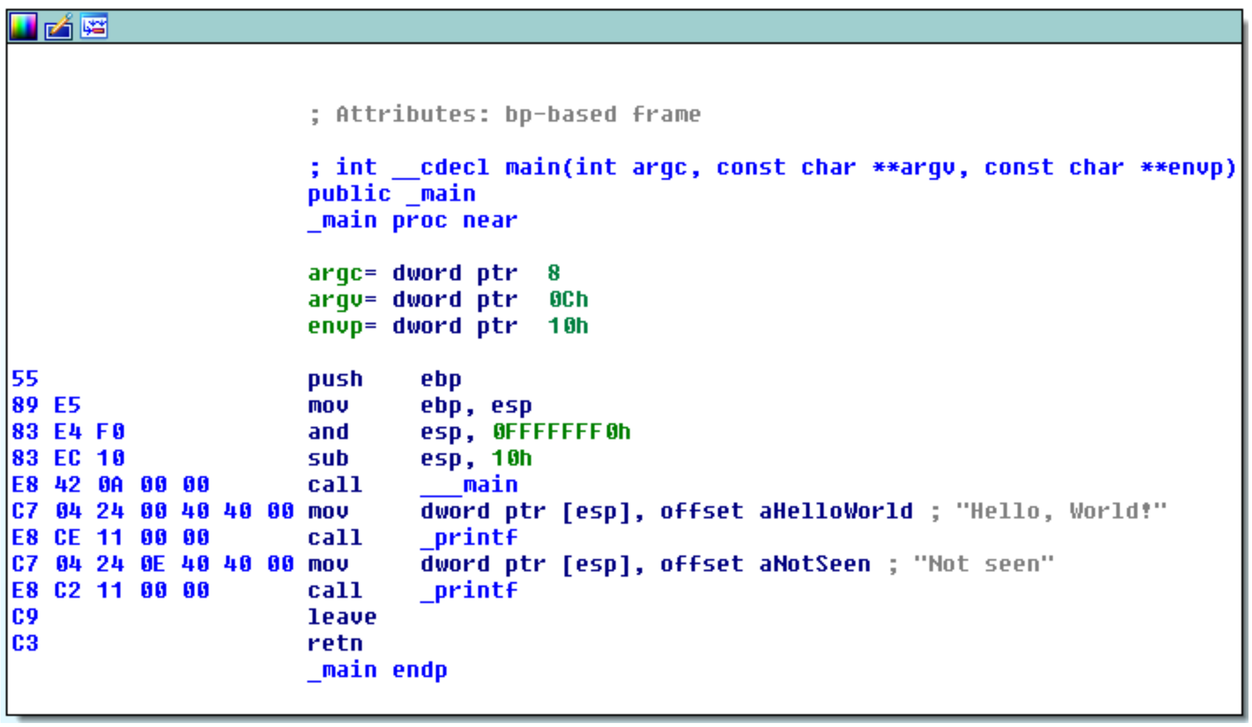


Figure 3: The initial C code representation by IDA Pro

Case 1: Overlapping instructions

In this case a particular byte is part of multiple instructions. When running the code, the processor doesn't have any problem with this, but during static disassembly there is no way to represent this correctly with the standard ways, and in fact none of the disassemblers can do that.

Our example will be the following simple situation:

JMP	-1	
EB	FF	C0
	INC	EAX

We have a short jump instruction (EB FF / JMP -1), which will jump back -1 byte, meaning that the next instruction will start with FF, which will be the beginning of INC EAX (FF C0). When IDA Pro goes over this it will disassemble EB FF as JMP -1 (it will print the exact location instead of "-1"), and the next instruction will start with C0.

Our modified example code will be the following:

```
#include <stdio.h>

void main() {
      printf("Hello, World!");

      asm(".intel_syntax noprefix\n"   //set assembly to Intel syntax
      ".byte 0xeb\n"                   //short jump
      ".byte 0xff\n"                   //-1
      ".byte 0xc0\n"                   //FF C0 = INC EAX, C0 will break
the following code segment
      );

      printf("Not seen");

}
```

If we load it to IDA Pro, we will get the following:

```
.text:00401520
.text:00401520                 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401520                                 public _main
.text:00401520                 _main:                                  ; CODE XREF: ___tmainCRTStartup+269↑p
.text:00401520 55                              push    ebp
.text:00401521 89 E5                           mov     ebp, esp
.text:00401523 83 E4 F0                        and     esp, 0FFFFFFF0h
.text:00401526 83 EC 10                        sub     esp, 10h
.text:00401529 E8 42 0A 00 00                  call    ___main
.text:0040152E C7 04 24 00 40 40 00            mov     dword ptr [esp], offset aHelloWorld ; "Hello, World!"
.text:00401535 E8 CE 11 00 00                  call    _printf
.text:0040153A
.text:0040153A                 loc_40153A:                             ; CODE XREF: .text:loc_40153A↑j
.text:0040153A EB FF                           jmp     short near ptr loc_40153A+1
.text:0040153C                 ; ---------------------------------------------------------------------------
.text:0040153C C0 C7 04                        rol     bh, 4
.text:0040153F 24 0E                           and     al, 0Eh
.text:00401541 40                              inc     eax
.text:00401542 40                              inc     eax
.text:00401543 00 E8                           add     al, ch
.text:00401545 BF 11 00 00 C9                  mov     edi, 0C9000011h
.text:0040154A C3                              retn
.text:0040154A                 ; ---------------------------------------------------------------------------
.text:0040154B 90 66 90 66 90                  align 10h
.text:00401550
```

Figure 4: Case1: Messed up code

We can see that the 2nd printf is gone, and code is completely messed up.

It's pretty common to see "jmp short near ptr loc_xxxxx +1" in places where anti-disassembly was done, IDA Pro also gives a red colored comment, with the two it's easy to spot suspicious places. To fix this, we can covert the "EB" instructions to data segment, and the rest to code segment. To convert between data and code, we need to move our cursor to the memory segment we want to update, and we press "D" (convert to data) or "C" (convert to code), depends to what we want to do. We will need to do multiple code conversions.

```
.text:0040152E C7 04 24 00 40 40 00            mov     dword ptr [esp], offset aHelloWorld ; "Hello, World!"
.text:00401535 E8 CE 11 00 00                  call    _printf
.text:00401535                 ; ---------------------------------------------------------------------------
.text:0040153A EB                              db 0EBh
.text:0040153B FF                              db 0FFh
.text:0040153C                 ; ---------------------------------------------------------------------------
.text:0040153C C0 C7 04                        rol     bh, 4
.text:0040153F 24 0E                           and     al, 0Eh
.text:00401541 40                              inc     eax
.text:00401542 40                              inc     eax
.text:00401543 00 E8                           add     al, ch
.text:00401545 BF 11 00 00 C9                  mov     edi, 0C9000011h
.text:0040154A C3                              retn
.text:0040154A                 ; ---------------------------------------------------------------------------
```

Figure 5: Case 1: fixing - step #1

```
.text:0040152E C7 04 24 00 40 40 00            mov     dword ptr [esp], offset aHelloWorld ; "Hello, World!"
.text:00401535 E8 CE 11 00 00                  call    _printf
.text:00401535                 ; ---------------------------------------------------------------------------
.text:0040153A EB                              db 0EBh
.text:0040153B                 ; ---------------------------------------------------------------------------
.text:0040153B FF C0                           inc     eax
.text:0040153B                 ; ---------------------------------------------------------------------------
.text:0040153D C7                              db 0C7h ; ¦
.text:0040153E 04                              db    4
.text:0040153F                 ; ---------------------------------------------------------------------------
.text:0040153F 24 0E                           and     al, 0Eh
.text:00401541 40                              inc     eax
.text:00401542 40                              inc     eax
.text:00401543 00 E8                           add     al, ch
.text:00401545 BF 11 00 00 C9                  mov     edi, 0C9000011h
.text:0040154A C3                              retn
.text:0040154A                 ; ---------------------------------------------------------------------------
```

Figure 6: Case 1: fixing - step #2

```
.text:0040152E C7 04 24 00 40 40 00                    mov     dword ptr [esp], offset aHelloWorld ; "Hello, World!"
.text:00401535 E8 CE 11 00 00                          call    _printf
.text:00401535                             ; ---------------------------------------------------------------------------
.text:0040153A EB                                      db 0EBh
.text:0040153B                             ; ---------------------------------------------------------------------------
.text:0040153B FF C0                                   inc     eax
.text:0040153D C7 04 24 0E 40 40 00                    mov     dword ptr [esp], offset aNotSeen ; "Not seen"
.text:0040153D                             ; ---------------------------------------------------------------------------
.text:00401544 E8                                      db 0E8h ; F
.text:00401545                             ; ---------------------------------------------------------------------------
.text:00401545 BF 11 00 00 C9                          mov     edi, 0C9000011h
.text:0040154A C3                                      retn
.text:0040154A                             ; ---------------------------------------------------------------------------
```

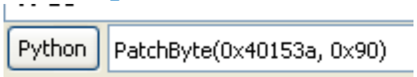
Figure 7: Case 1: fixing - step #3

```
:0040152E C7 04 24 00 40 40 00                    mov     dword ptr [esp], offset aHelloWorld ; "Hello, World!"
:00401535 E8 CE 11 00 00                          call    _printf
:00401535                             ; ---------------------------------------------------------------------------
:0040153A EB                                      db 0EBh
:0040153B                             ; ---------------------------------------------------------------------------
:0040153B FF C0                                   inc     eax
:0040153D C7 04 24 0E 40 40 00                    mov     dword ptr [esp], offset aNotSeen ; "Not seen"
:00401544 E8 BF 11 00 00                          call    _printf
:00401549 C9                                      leave
:0040154A C3                                      retn
:0040154A                             ; ---------------------------------------------------------------------------
```

Figure 8: Case 1: fixing - step #4

Although we can see the correct code now, the "data" entry still makes it impossible to make a graph mode. As a last step we can patch that byte to a NOP instruction, with a simple Python function from the command bar at the bottom.

```
PatchByte(0x40153A, 0x90)
```



Here is a Python script that will search for locations where we have "EB FF" opcodes.

```
def find_jmp_ff():
      results = []
      ea = FindBinary(SegStart(ScreenEA()), SEARCH_DOWN, "EB FF")
      while(ea != BADADDR):
            if GetMnem(ea) == "jmp":
                  results.append(ea)
                  Message("Found possibly anti-disassembly technique at 
0x%x, instruction: %s\n" % (ea,GetDisasm(ea)))
            ea = FindBinary(ea, SEARCH_NEXT, "EB FF")
      return results

def main():
      anti_da_locations = []
      anti_da_locations.extend(find_jmp_ff())
      for i in anti_da_locations:
            SetColor(i, CIC_ITEM, 0x0000ff)

if __name__ == "__main__":
    main()
```

The find_jmp_ff function will start the search from the beginning of the section, where our cursor is, and start searching for “EB FF”. If found it will check if this is really part of a JMP (and not somewhere else in the middle of another opcode). If yes, it will store the results, and search until the end (BADADDR found). The SEARCH_DOWN parameter means that it will search from the location downwards, the SEARCH_NEXT is a search for the next place.

Finally we set the background color of the found instruction to RED.

Case 2: Jump with constant condition

In this case we utilize that disassemblers, will disassemble the false branch of a tree, which means if we have a JZ instruction, the opcodes starting at the jump address, will be parsed only after the false branch was examined. Let's see the following example:

XOR		JZ			POP
33	C0	74	01	E9	58
				JMP	

The XOR instruction will always make the JZ statement true, so the false branch will never be hit in real life. The first instruction in the false path would start right after the JZ instruction, so the first opcode would be E9, which is a JMP which will take an address as an argument. The true branch starts with the opcode 58, which is a POP statement. The problem is that JMP will be interpreted first, and the 58 opcode will be interpreted as part of the address, so it will mess up the code.

Let's see how can we modify our C code to create such a trick:

```
void main() {
      printf("Hello, World!");

      asm(".intel_syntax noprefix\n"
      "xor eax, eax\n"
      //"jz .later\n"
      ".byte 0x74\n"
      ".byte 0x01\n"
      //".later:\n"
      ".byte 0xe9\n"
      );

      printf("Not seen");

}
```

74 01 will jump to the location after E9. If we use labels in our assembly code, the compiler will leave traces to that, and IDA will find it so our anti-disassembly wouldn't be successful. I left it there in comments, to ease the understanding of the code.

If we load it to IDA Pro, we can see that our method was successful again, and we can't see the second printf instruction.

```
.text:00401520                        _main:                               ; CODE XREF: ___tmainCRTStartup+269↑p
.text:00401520 55                                  push    ebp
.text:00401521 89 E5                               mov     ebp, esp
.text:00401523 83 E4 F0                            and     esp, 0FFFFFFF0h
.text:00401526 83 EC 10                            sub     esp, 10h
.text:00401529 E8 42 0A 00 00                      call    ___main
.text:0040152E C7 04 24 00 40 40 00                mov     dword ptr [esp], offset aHelloWorld ; "Hello, World!"
.text:00401535 E8 CE 11 00 00                      call    _printf
.text:0040153A 31 C0                               xor     eax, eax
.text:0040153C 74 01                               jz      short near ptr loc_40153E+1
.text:0040153E
.text:0040153E                        loc_40153E:                          ; CODE XREF: .text:0040153C↑j
.text:0040153E E9 C7 04 24 0E                      jmp     near ptr 0E641A0Ah
.text:00401543                        ; ---------------------------------------------------------------------------
.text:00401543 40                                  inc     eax
.text:00401544 40                                  inc     eax
.text:00401545 00 E8                               add     al, ch
.text:00401547 BD 11 00 00 C9                      mov     ebp, 0C9000011h
.text:0040154C C3                                  retn
                                      ; ---------------------------------------------------------------------------
```

Figure 9: Case 2: Messed up code

We can use the same method to manually correct it. Convert E9 to data, and the following parts to code.

```
.text:00401520                        _main:                               ; CODE XREF: ___tmainCRTStartup+269↑p
.text:00401520 55                                  push    ebp
.text:00401521 89 E5                               mov     ebp, esp
.text:00401523 83 E4 F0                            and     esp, 0FFFFFFF0h
.text:00401526 83 EC 10                            sub     esp, 10h
.text:00401529 E8 42 0A 00 00                      call    ___main
.text:0040152E C7 04 24 00 40 40 00                mov     dword ptr [esp], offset aHelloWorld ; "Hello, World!"
.text:00401535 E8 CE 11 00 00                      call    _printf
.text:0040153A 31 C0                               xor     eax, eax
.text:0040153C 74 01                               jz      short loc_40153F
.text:0040153C                        ; ---------------------------------------------------------------------------
.text:0040153E E9                                  db 0E9h
.text:0040153F                        ; ---------------------------------------------------------------------------
.text:0040153F
.text:0040153F                        loc_40153F:                          ; CODE XREF: .text:0040153C↑j
.text:0040153F C7 04 24 0E 40 40 00                mov     dword ptr [esp], offset aNotSeen ; "Not seen"
.text:00401546 E8 BD 11 00 00                      call    _printf
.text:0040154B C9                                  leave
.text:0040154C C3                                  retn
.text:0040154C                        ; ---------------------------------------------------------------------------
```

Figure 10: Case 2: After fix

At the very end we can patch the byte, and convert it to code.

```
.text:00401520                        _main:                               ; CODE XREF: ___tmainCRTStartup+269↑p
.text:00401520 55                                  push    ebp
.text:00401521 89 E5                               mov     ebp, esp
.text:00401523 83 E4 F0                            and     esp, 0FFFFFFF0h
.text:00401526 83 EC 10                            sub     esp, 10h
.text:00401529 E8 42 0A 00 00                      call    ___main
.text:0040152E C7 04 24 00 40 40 00                mov     dword ptr [esp], offset aHelloWorld ; "Hello, World!"
.text:00401535 E8 CE 11 00 00                      call    _printf
.text:0040153A 31 C0                               xor     eax, eax
.text:0040153C 74 01                               jz      short loc_40153F
.text:0040153E 90                                  nop
.text:0040153F
.text:0040153F                        loc_40153F:                          ; CODE XREF: .text:0040153C↑j
.text:0040153F C7 04 24 0E 40 40 00                mov     dword ptr [esp], offset aNotSeen ; "Not seen"
.text:00401546 E8 BD 11 00 00                      call    _printf
.text:0040154B C9                                  leave
.text:0040154C C3                                  retn
.text:0040154C                        ; ---------------------------------------------------------------------------
```

Figure 11: Case 2: After patching

In order to find these locations, 0ur script for IDA Pro would need to look for places, where we XOR the values of the same registers (any), followed by a JZ instruction. The search function would look like this:

```
def find_xor_jz():
    heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))
```

```
    results = []
    found_first = False
    previous = ""
    for i in heads:
        if (found_first and GetMnem(i) == "jz"):
            results.append(previous)
            results.append(i)
            Message("Found possibly anti-disassembly technique at 
0x%x, instruction: %s\n" % (previous,GetDisasm(previous)))
            Message("Found possibly anti-disassembly technique at 
0x%x, instruction: %s\n" % (i,GetDisasm(i)))
            found_first = False
        elif GetMnem(i) == "xor" and GetOpnd(i,0) == GetOpnd(i,1):
            found_first = True
        else: found_first = False
        previous = i
    return results
```

The script will go through the entire section, looking for XOR instruction, followed by a JZ.

Case3: JZ/JNZ instruction with the same target

This method is a bit similar to the previous one. There is a JZ instruction followed by a JNZ, where both of them pointing to the same location. The effect will be that this will create an unconditional jump, but the disassembler won't recognize it. After JZ the false branch will be checked first, which will be JNZ, and there again the false branch will be checked, which again will be an E9 (JMP) in our case, which will never be executed in real life. The real code will start with 58 (POP):

JZ		JNZ			POP
74	03	75	01	E9	58
				JMP	

Our C code to make this happen:

```
#include <stdio.h>

void main() {
      printf("Hello, World!");

      asm(".intel_syntax noprefix\n"
      //"jz .later\n"
      ".byte 0x74\n"
      ".byte 0x03\n"

      //"jnz .later\n"
      ".byte 0x75\n"
      ".byte 0x01\n"

      //".later:\n"
      ".byte 0xe9\n"
      );
      printf("Not seen");

}
```

Loading it to IDA Pro:

```
.text:00401520 _main:                                       ; CODE XREF: ___tmainCRTStartup+269↑p
.text:00401520                 push    ebp
.text:00401521                 mov     ebp, esp
.text:00401523                 and     esp, 0FFFFFFF0h
.text:00401526                 sub     esp, 10h
.text:00401529                 call    ___main
.text:0040152E                 mov     dword ptr [esp], offset aHelloWorld ; "Hello, World!"
.text:00401535                 call    _printf
.text:0040153A                 jz      short near ptr loc_40153E+1
.text:0040153C                 jnz     short near ptr loc_40153E+1
.text:0040153E
.text:0040153E loc_40153E:                                  ; CODE XREF: .text:0040153A↑j
.text:0040153E                                              ; .text:0040153C↑j
.text:0040153E                 jmp     near ptr 0E641A0Ah
.text:00401543 ; ---------------------------------------------------------------------------
.text:00401543                 inc     eax
.text:00401544                 inc     eax
.text:00401545                 add     al, ch
.text:00401547                 mov     ebp, 0C9000011h
.text:0040154C                 retn
.text:0040154C ; ---------------------------------------------------------------------------
```

Figure 12: Case 3: Messed up code

We can see that our code is obfuscated again. To correct and patch it, we can use the very same method as in the previous two cases.

```
.text:00401520
.text:00401520 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401520                 public _main
.text:00401520 _main:                                       ; CODE XREF: ___tmainCRTStartup+269↑p
.text:00401520                 push    ebp
.text:00401521                 mov     ebp, esp
.text:00401523                 and     esp, 0FFFFFFF0h
.text:00401526                 sub     esp, 10h
.text:00401529                 call    ___main
.text:0040152E                 mov     dword ptr [esp], offset aHelloWorld ; "Hello, World!"
.text:00401535                 call    _printf
.text:0040153A                 jz      short loc_40153F
.text:0040153C                 jnz     short loc_40153F
.text:0040153E                 nop
.text:0040153F
.text:0040153F loc_40153F:                                  ; CODE XREF: .text:0040153A↑j
.text:0040153F                                              ; .text:0040153C↑j
.text:0040153F                 mov     dword ptr [esp], offset aNotSeen ; "Not seen"
.text:00401546                 call    _printf
.text:0040154B                 leave
.text:0040154C                 retn
.text:0040154C ; ---------------------------------------------------------------------------
```

Figure 13: Case 3: After fix and patching

Our IDA Pro script function to find such places:

```
def find_jz_jnz():
      results = []
      ea = FindBinary(SegStart(ScreenEA()), SEARCH_DOWN, "74 03 75 01")
      while(ea != BADADDR):
            if GetMnem(ea) == "jz":
                  results.append(ea)
                  results.append(ea+2)
                  Message("Found possibly anti-disassembly technique at 
0x%x, instruction: %s,%s\n" % (ea,GetDisasm(ea),GetDisasm(ea+2)))
            ea = FindBinary(ea, SEARCH_NEXT, "74 03 75 01")
```

```
    return results
```

The function will search through the section, looking for the bytes “74 03 75 01”, once it’s found, it will verify that it is indeed a beginning of a JZ section.

Putting the entire script together

Here is the entire script, which will look for all the 3 techniques above, and mark those locations with red:

```
from idautils import *
from idc import *

def find_xor_jz():
	heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))
	results = []
	found_first = False
	previous = ""
	for i in heads:
		if (found_first and GetMnem(i) == "jz"):
			results.append(previous)
			results.append(i)
			Message("Found possibly anti-disassembly technique at 
0x%x, instruction: %s\n" % (previous,GetDisasm(previous)))
			Message("Found possibly anti-disassembly technique at 
0x%x, instruction: %s\n" % (i,GetDisasm(i)))
			found_first = False
		elif GetMnem(i) == "xor" and GetOpnd(i,0) == GetOpnd(i,1):
			found_first = True
		else: found_first = False
		previous = i
	return results

def find_jmp_ff():
	results = []
	ea = FindBinary(SegStart(ScreenEA()), SEARCH_DOWN, "EB FF")
	while(ea != BADADDR):
		if GetMnem(ea) == "jmp":
			results.append(ea)
			Message("Found possibly anti-disassembly technique at 
0x%x, instruction: %s\n" % (ea,GetDisasm(ea)))
		ea = FindBinary(ea, SEARCH_NEXT, "EB FF")
	return results

def find_jz_jnz():
	results = []
	ea = FindBinary(SegStart(ScreenEA()), SEARCH_DOWN, "74 03 75 01")
	while(ea != BADADDR):
		if GetMnem(ea) == "jz":
			results.append(ea)
			results.append(ea+2)
			Message("Found possibly anti-disassembly technique at 
0x%x, instruction: %s,%s\n" % (ea,GetDisasm(ea),GetDisasm(ea+2)))
		ea = FindBinary(ea, SEARCH_NEXT, "74 03 75 01")
	return results

def main():
```

```
      anti_da_locations = []
      anti_da_locations.extend(find_xor_jnz())
      anti_da_locations.extend(find_jmp_ff())
      anti_da_locations.extend(find_jz_jnz())
      for i in anti_da_locations:
           SetColor(i, CIC_ITEM, 0x0000ff)

if __name__ == "__main__":
    main()
```