Anti-Disassembly techniques used by malware (a primer)

Rahul Nair | 22 Nov 2015

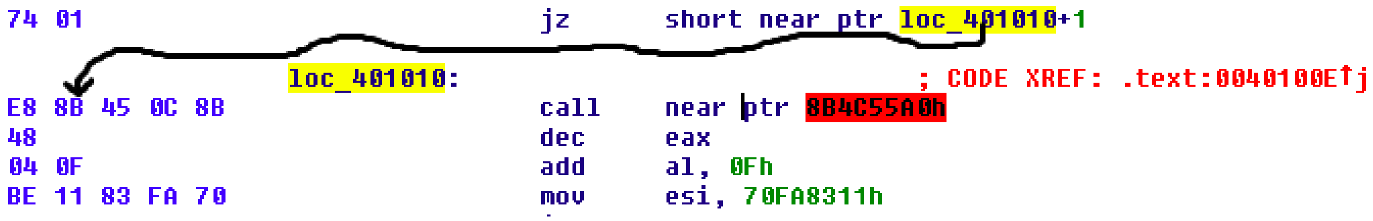
There are chances that malware authors implement some kind of trolling so that a malware analyst has a hard time figuring out code during static analysis (IDA Pro ?). Implementing these cunning asm instruction will not cause any issues to the flow of the program but will confuse static analysis tools such as IDA Pro from interpreting the code correctly.

Once upon a time there were 2 kinds of disassembly algorithms - Linear disassembly and flow-oriented disassembly.The former ~~was used in tutorials/ nobody gives a damn~~ is not used that much in disassemblers.

What we are concerned about is the latter which is used in IDA Pro and sometime gamed by malware authors-

1.Jump Instructions to a location with constant value

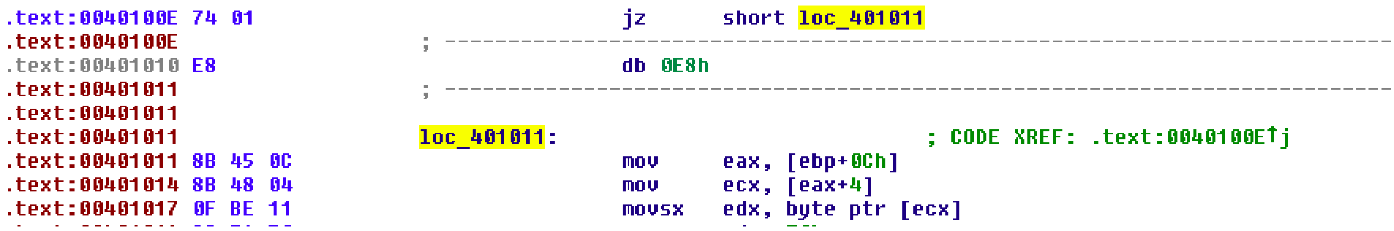
This is the most used trick by malware writers/anti-disassembly programs which create jumps into the same location + 1 or 2 bytes. It would lead to interpretation of completely different byte code by the system.



```
74 01                                  jz      short near ptr loc_401010+1
                  loc_401010:                                  ; CODE XREF: .text:0040100E↑j
E8 8B 45 0C 8B                         call    near ptr 8B4C55A0h
48                                     dec     eax
04 0F                                  add     al, 0Fh
BE 11 83 FA 70                         mov     esi, 70FA8311h
```

For instance the actual jump instance here would take the flow of program to the bytecode mentioned above.
Since tools like IDA pro are not that clever(no offense to the creator) it cannot make such judgements and instead interprets the opcode from E8 instead which shows us a bunch of call instructions to some random crappy address, weird decrements and adds.

No we can fix this with ease in IDA PRO. Do that by pressing D on the E8 and C key on the 8B Opcode and voila! you get what is actually being interpreted.
After playing around more with the C & D key you get the following in IDA which seems legit :P

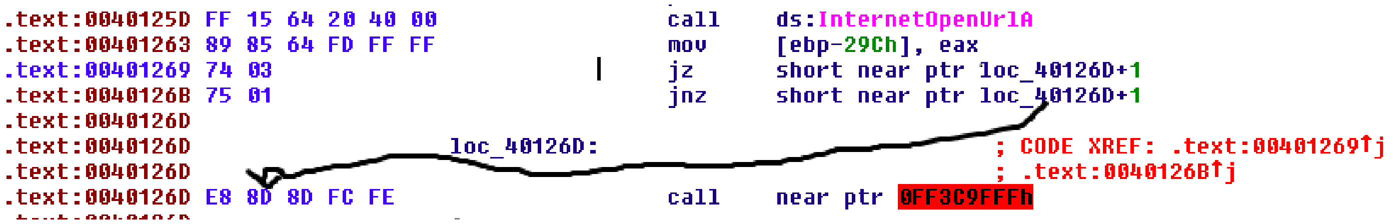


```
.text:0040100E 74 01                          jz      short loc_401011
.text:0040100E                  ; ---------------------------------------------------------------------------
.text:00401010 E8                             db 0E8h
.text:00401011                  ; ---------------------------------------------------------------------------
.text:00401011
.text:00401011                  loc_401011:                         ; CODE XREF: .text:0040100E↑j
.text:00401011 8B 45 0C                       mov     eax, [ebp+0Ch]
.text:00401014 8B 48 04                       mov     ecx, [eax+4]
.text:00401017 0F BE 11                       movsx   edx, byte ptr [ecx]
```

Now what has happened here is that the the author might have inserted something known as a **rogue byte** which confuses IDA pro leading to a wrong interpretation of the rest of the opcode.This is a simple technique and if you dont like to see that ugly E8 byte you could NOP it out :)

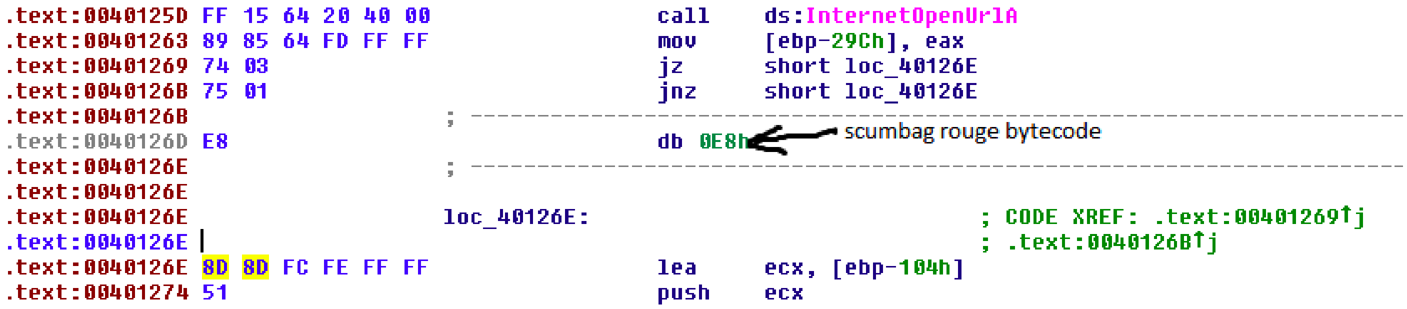
2.Jump Instructions to the Same target

IDA Pro usually follows this behavior where for a conditional instruction (**jnz**) it first disassembles the false branch of the conditional instruction and then moves forward to the true part.
From a malware POV since both the jz and jnz are present it is similar to an unconditional jump



```
.text:0040125D FF 15 64 20 40 00                       call    ds:InternetOpenUrlA
.text:00401263 89 85 64 FD FF FF                       mov     [ebp-29Ch], eax
.text:00401269 74 03                                   jz      short near ptr loc_40126D+1
.text:0040126B 75 01                                   jnz     short near ptr loc_40126D+1
.text:0040126D
.text:0040126D                 loc_40126D:                                     ; CODE XREF: .text:00401269↑j
.text:0040126D                                                                 ; .text:0040126B↑j
.text:0040126D E8 8D 8D FC FE                          call    near ptr 0FF3C9FFFh
```

Once IDA pro reaches the jz instruction it would first branch out and interpret the false instruction and move on to jnz where it would do the same.A nice and dirty trick is to insert a rogue byte code and make the disassembler interpret the instructions as a call.
If we do the *C & D thingy* in IDA pro as mentioned in **1.** we get the following code

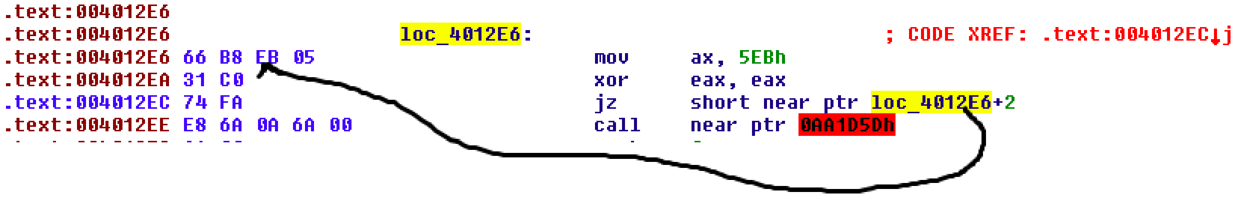


```
.text:0040125D FF 15 64 20 40 00                       call    ds:InternetOpenUrlA
.text:00401263 89 85 64 FD FF FF                       mov     [ebp-29Ch], eax
.text:00401269 74 03                                   jz      short loc_40126E
.text:0040126B 75 01                                   jnz     short loc_40126E
.text:0040126B                 ; ---------------------------------------------------------------
.text:0040126D E8                                      db  0E8h
.text:0040126E                 ; ---------------------------------------------------------------
.text:0040126E
.text:0040126E                 loc_40126E:                                     ; CODE XREF: .text:00401269↑j
.text:0040126E                                                                 ; .text:0040126B↑j
.text:0040126E 8D 8D FC FE FF FF                       lea     ecx, [ebp-104h]
.text:00401274 51                                      push    ecx
```

scumbag rouge bytecode

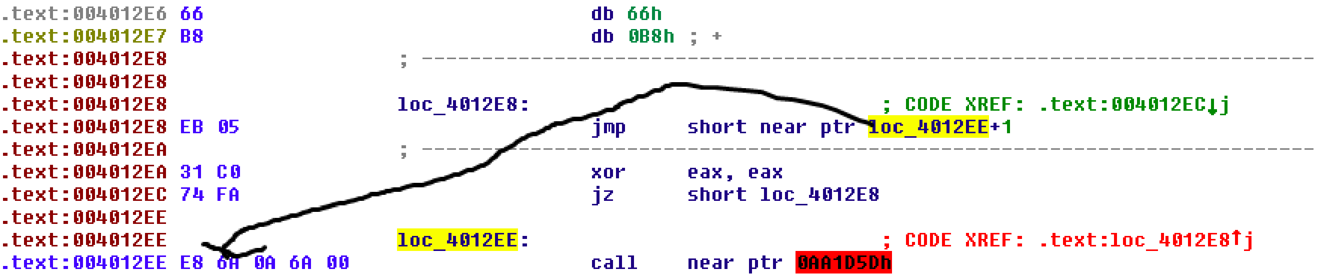
3.Ping-Pong jumps I have no idea what this technique is named as but it involves doing a lot of jumping around using the method mentioned in **1.** and maybe even a bit of **2**

Let's look at this innocent jump below.



```
.text:004012E6
.text:004012E6                  loc_4012E6:                          ; CODE XREF: .text:004012EC↓j
.text:004012E6 66 B8 EB 05                      mov     ax, 5EBh
.text:004012EA 31 C0                            xor     eax, eax
.text:004012EC 74 FA                            jz      short near ptr loc_4012E6+2
.text:004012EE E8 6A 0A 6A 00                   call    near ptr 0AA1D5Dh
```

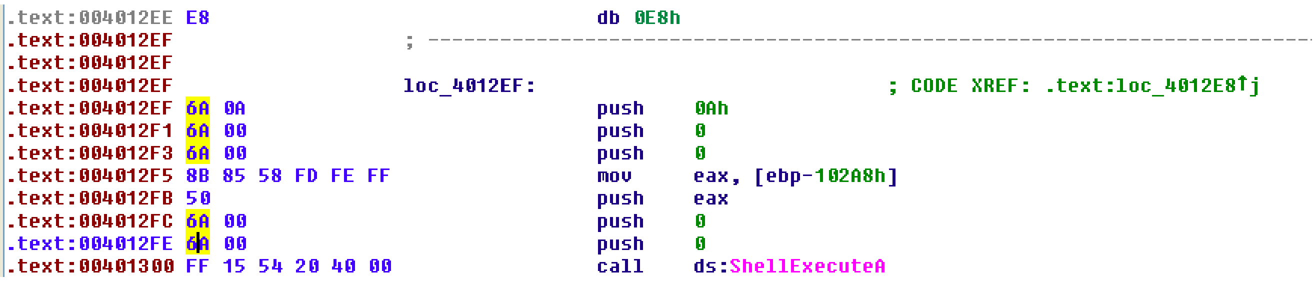
This jumps goes back to loc_4012E6+2 which would be the EB opcode. If we ignore the 66 and B8 opcode ,make IDA interpret the rest as code instead we get the following



```
.text:004012E6 66                               db 66h
.text:004012E7 B8                               db 0B8h ; +
.text:004012E8                  ; ---------------------------------------------------------------
.text:004012E8
.text:004012E8                  loc_4012E8:                          ; CODE XREF: .text:004012EC↓j
.text:004012E8 EB 05                            jmp     short near ptr loc_4012EE+1
.text:004012EA                  ; ---------------------------------------------------------------
.text:004012EA 31 C0                            xor     eax, eax
.text:004012EC 74 FA                            jz      short loc_4012E8
.text:004012EE
.text:004012EE                  loc_4012EE:                          ; CODE XREF: .text:loc_4012E8↑j
.text:004012EE E8 6A 0A 6A 00                   call    near ptr 0AA1D5Dh
```

Yay more jumps.

Once again ignoring the other E8 byte and considering the rest as code the result is as follows-



```
.text:004012EE E8                               db 0E8h
.text:004012EF                  ; ---------------------------------------------------------------
.text:004012EF
.text:004012EF                  loc_4012EF:                          ; CODE XREF: .text:loc_4012E8↑j
.text:004012EF 6A 0A                            push    0Ah
.text:004012F1 6A 00                            push    0
.text:004012F3 6A 00                            push    0
.text:004012F5 8B 85 58 FD FE FF                mov     eax, [ebp-102A8h]
.text:004012FB 50                               push    eax
.text:004012FC 6A 00                            push    0
.text:004012FE 6A 00                            push    0
.text:00401300 FF 15 54 20 40 00                call    ds:ShellExecuteA
```

We can see how incorporating rogue bytes obscures the real function call from being hidden in static analysis.

4.Usage of Function Pointers

Instead of a screen shot here is a piece of code

```
mov [ebp+var8],offset sub4211C1

push 4Ah

call [ebp+var_8]
```

What happens above is that a function is called via use of a reference to an address. For example for the function call it would get the funciton stringname by the use of some weird bunch of dec
subroutine and save the value in an offset sub4211C1. This
make static analysis really hard since IDA won't recognize i
From a static analysis point of view though it dosen't seem
massive harm this coupled with other anti-disassembly tech
lead to annoyance for an analyst.

There are a couple more annoying techniques which I will
another post such as abusing the return pointer (for fun and
,using your own Structed Exception Handler (SEH) and scr
around with the stack-frame construction in IDA pro.

Menu

Home

Test2

Subscribe

Rahul Nair

Welcome to Malwina

Welcome to malwinator.Thi
by a grad student to...

Theme Attila by zutrinken

Publ