PROTECTING THE BINARY PROGRAM

Anti-Reversing and Anti-Debugging

Luigi Coppolino, Luigi Romano

CONTACT INFO

Prof. Luigi Coppolino
luigi.coppolino@uniparthenope.it

Prof. Salvatore D'Antonio
salvatore.dantonio@uniparthenope.it

Prof. Luigi Romano
luigi.romano@uniparthenope.it

Università degli Studi di Napoli "Parthenope"
Dipartimento di Ingegneria

ROADMAP

- ...

Anti-Reverse Techniques

ANTI-REVERSE ENGINEERING

- Protect your software: avoid cracks, license key generators, …
- Protect your IP: avoid understanding of a new, best of breed alogirthm
- Make more difficult the life of hackers…trying to get knowledge about vulnerabilities in a software

- All of this is especially true when your software is … VIRUS/WORM …

Mixing data and instructions

LET'S START FROM A SIMPLE CODE: BASIC.ASM

```
section .text
global _start

_start:
    mov     eax, 0xf001
```

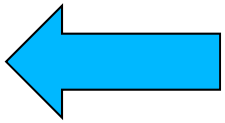
```
>> nasm -f elf64 -o basic.o basic.asm
>> ld -o basic basic.o -m elf_x86_64
>> objdump -d -M intel basic
```

Disassembling the binary properly returns the original assembly

```
basic:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:       b8 01 f0 00 00          mov    eax,0xf001
```

ANTIREVERSE01.ASM

```
section .text
global _start

_start:
    jmp label+1
label: DB 0x90
    mov    eax, 0xf001
```



We are mixing data and code

```
>> nasm -f elf64 -o antiReverse.o antiReverse01.asm
>> ld -o antiReverse antiReverse.o -m elf_x86_64
```

OBJDUMP -D -M INTEL64 ANTIREVERSE

```
antiReverse:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:       eb 01                   jmp    400083 <label+0x1>

0000000000400082 <label>:
  400082:       90                      nop
  400083:       b8 01 f0 00 00          mov    $0xf001,%eax
```

During the disassembly it is not possible to distinguish data and instructions…thus the 0x90 is interpreted as an instruction

OBJDUMP -D -M INTEL64 ANTIREVERSE

```
antiReverse:     file format elf64-x86-64


Disassemb

00000000
  400080:

0000000000400082 <label>:
  400082:       90                      nop
  400083:       b8 01 f0 00 00          mov    $0xf001,%eax
```

Linear Disassembly: go ahead interpreting byte-by-byte
0x90 = NOP
0xE8 0xXX 0xXX 0xXX 0xXX = CALL

```
section .text
global _start

_start:
    jmp label+1
label: DB 0xE9
    mov eax, 0xf001
```

E9 is the first byte of a jmp instruction

```
>> nasm -f elf64 -o antiReverse.o antiReverse02.asm
>> ld -o antiReverse antiReverse.o -m elf_x86_64
```

OBJDUMP -D -M INTEL64 ANTIREVERSE

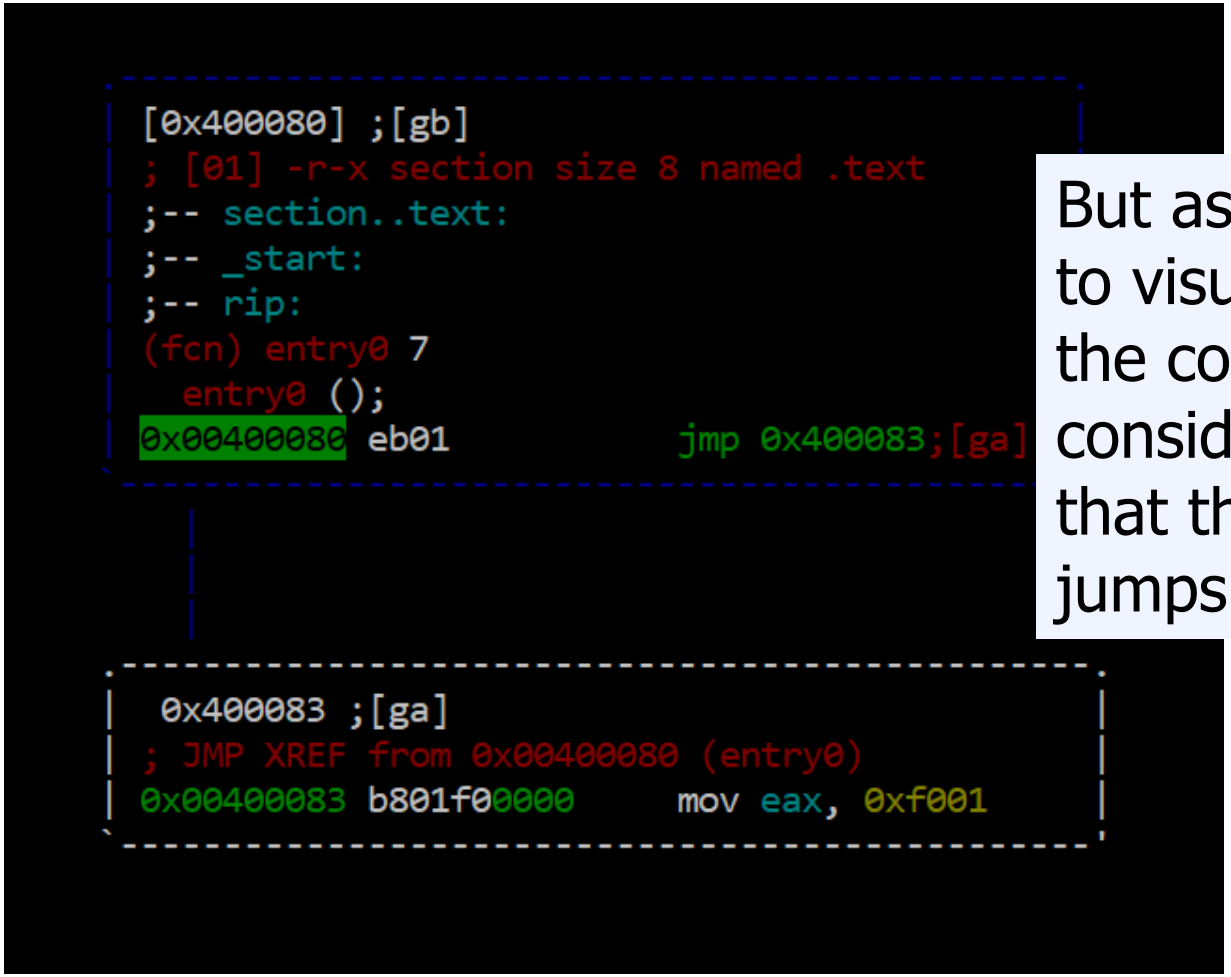
```
antiReverse02:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:       eb 01                   jmp    400083 <label+0x1>

0000000000400082 <label>:
  400082:       e9 b8 01 f0 00          jmpq   130023f <__bss_start+0xcff23f>
```

When the disassembler reaches E9 it interprets it as the start of a jmp thus the following bytes are considere part of this instruction

RADARE2…FOOLED…BUT…

```
idattico/working$ r2 antiReverse02
Warning: Cannot initialize dynamic strings
 -- Setup dbg.fpregs to true to visualize the fpu registers in the debugger view.
[0x08048060]> aaa
[x] Analyze all flags
[x] Analyze len bytes
[x] Analyze function
[x] Use -AA or aaaa t
[x] Constructing a fu
[0x08048060]> afl
0x08048060    3 4            entry0
[0x08048060]> s entry0
[0x08048060]> pdf
            ;-- section..text:
            ;-- _start:
            ;-- eip:
/ (fcn) entry0 4
|   entry0 ();
|           0x08048060      eb01           jmp 0x8048063              ; [01] -r-x
            ;-- label:
            0x08048062      00eb           add bl, ch
[0x08048060]>
```

Radare2 makes the same error when a static analysis is performed (*pdf*)



But as soon as we move to visualization mode and the control workflow is considered, it comes out that the instruction flow jumps the E9 byte

IDA…OK: CONTROL FLOW DISASSEMBLER…DOESN'T GO INSTRUCTRION BY INSTRUCTION BUT FOLLOWS THE EXECUTION FLOW


```
text:0000000000400080 ; Attributes: noreturn
text:0000000000400080
text:0000000000400080                 public _start
text:0000000000400080 _start          proc near               ; DATA XREF: LOAD:00
text:0000000000400080                 jmp     short loc_400083
text:0000000000400080 ; ---------------------------------------------------------
text:0000000000400082 label           db 0E9h
text:0000000000400083 ; ---------------------------------------------------------
text:0000000000400083
text:0000000000400083 loc_400083:                             ; CODE XREF: _start↑
text:0000000000400083                 mov     eax, 0F001h
text:0000000000400083 _start          endp
text:0000000000400083
text:0000000000400083 _text           ends
text:0000000000400083
text:0000000000400083
text:0000000000400083                 end _start
```

Exploiting weaknesses in the disassembler behavior: fake jumps

ANTI CONTROL FLOW DISASSEMBLY: THE OLD WAY (NOT WORKING ANYMORE)

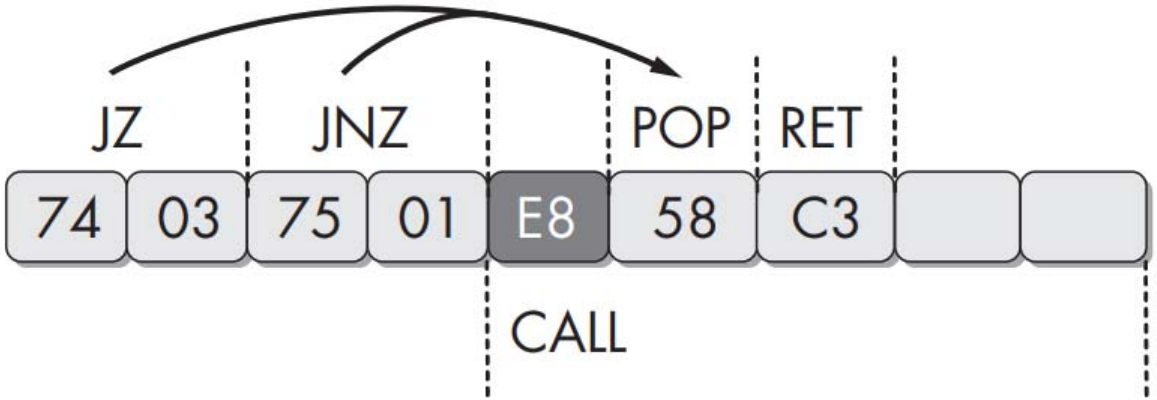


Figure 15-2: A jz instruction followed by a jnz instruction

Jump instruction with the same target: the disassembler takes memory of jump destinations for following analysis, but then first analyze the instruction after jump (i.e. the call E8 58 C3 …); finished the false branch start analyzing the other branch but finds the instructions already analyzed thus go forward without further analysis, thus missing POP and RET instructions

ANTI CONTROL FLOW DISASSEMBLY: THE OLD WAY (NOT WORKING ANYMORE)


```
33 C0                   xor     eax, eax
74 01                   jz      short near ptr loc_4011C5
        ; ---------------------------------------------------------------------
E9                      db 0E9h
        ; ---------------------------------------------------------------------
        loc_4011C5:                             ; CODE XREF: 004011C2j
                                                ; DATA XREF: .rdata:004020AC0
58                      pop     eax
C3                      retn
```

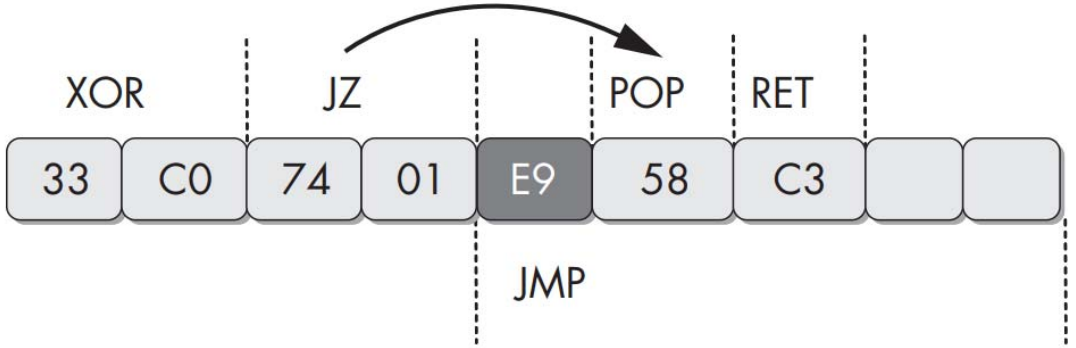


Figure 15-3: False conditional of xor followed by a jz instruction

False conditional branches: the JZ will be always taken
The disassembler would have taken note of jump destination but continued with next byte (JMP)

Overlapping Instructions

INSTRUCTION SCISSION

- This is a practice which consists in “hiding an instruction” within another decoy instruction
- The idea: given a binary sequence it can be interpreted in different ways based on the offset from which decoding starts:
- Example:

  let’s consider the byte sequence: 68 c3 90 90 90

  offset 00: 68 c3 90 90 90 => push 0x909090c3

  offset 01: c3 => retn

```
00: EB 01           jmp 3
02: 68 c3 90 90 90  push 0x909090c3
```

will be effectively executed as

```
00: EB 01    jmp 3
03: C3       retn
```

MULTIPLE OVERLAPS

- An instruction can contain several lines of code:

```
00: EB02                          jmp  4
02: 69846A40682C104000EB02        imul eax, [edx + ebp*2 +
                                      0102C6840], 0x002EB0040
```

  - Actually is…

```
00: EB02          jmp  4
04: 6A40          push 040
06: 682C104000    push 0x40102C
0B: EB02          jmp  0xF
0F: …
```

- An instruction jumps within itself

```
00: EB FF       jmp 1
02: C0 C3 00    rol bl, 0
```

- Will execute as

```
00: EB FF    jmp 1
01: FF C0    inc eax
03: C3       retn
```

DIFFERENT CPU MODES

- The actual interpretation of the binary could depend on the CPU mode…
  - 32 bit vs 64 bit modes…

  The sequence **63 D8**
    - in 32 bit is the **ARPL** instruction
    - in 64 bit is the **MOVSXD** instruction
- By switching from 32 to 64 bit mode the code we have

32 bit

```
00: 63D8      arpl ax,bx
02: 48        dec eax
03: 01C0      add eax,eax
05: CB        retf
```

64 bit

```
00: 63D8      movsxd rbx,eax
02: 4801C0    add rax,rax
05: CB        retf
```

- About switching 32<-->64 bit modes: http://blog.rewolf.pl/blog/?p=102

LAB: AntiReversing Hello World

ANTI CONTROL FLOW DISASSEMBLY: ANTIREVERSE03_01.ASM

```
SECTION .data

Hello:      db "Hello world!",10
len_Hello:  equ $-Hello

SECTION .text

global _start

_start:
        mov rax,1           ; write syscall (x86_64)
        mov rdi,1           ; fd = stdout

        mov rsi,Hello       ; *buf = Hello
        mov rdx,len_Hello   ; count = len_Hello
        syscall

        mov rax,60          ; exit syscall (x86_64)
        mov rdi,0           ; status = 0 (exit normally)
        syscall
```

```
B8 01 00 00 00
BF 01 00 00 00
48 BE D8 00 60 00 00 00+
BA 0D 00 00 00
0F 05
B8 3C 00 00 00
BF 00 00 00 00
0F 05
```

Let's start from a simple helloworld program…

```
$ nasm -f elf64 -o antiReverse03_01.o antiReverse03_01.asm
$ ld -o antiReverse antiReverse03_01.o
$ ./antiReverse
Hello world!
```

FOOLING IDA PRO: ANTIREVERSE03_02.ASM

We are inserting this preamble…and apparently nothing changes

```
fool:    DB 0x66, 0xB8, 0xEB, 0x05, 0x31, 0xC0, 0x74, 0xFA, 0xE8
```

```
mov rax,1               ; write syscall (x86_64)
mov rdi,1               ; fd = stdout

mov rsi,Hello           ; *buf = Hello
mov rdx,len_Hello       ; count = len_Hello
syscall

mov rax,60              ; exit syscall (x86_64)
mov rdi,0               ; status = 0 (exit normally)
syscall
```

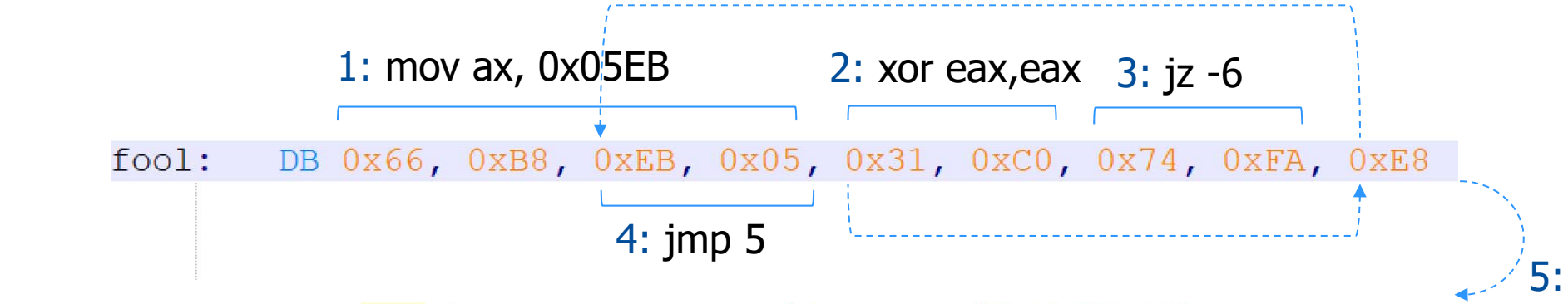
```
$ nasm -f elf64 -o antiReverse03_02.o antiReverse03_02.asm
$ ld -o antiReverse antiReverse03_02.o
$ ./antiReverse
Hello world!
```

IDAPRO DIASASSEMBLED VS ORIGINAL CODE

```
04000B0
04000B0
04000B0                         public _start
04000B0                 _start  proc near               ; CODE XREF: _start+6↓j
04000B0                                                 ; DATA
04000B0 66 B8 EB 05             mov     ax, 5EBh        ; Alter
04000B0                                                 ; fool
04000B4 31 C0                   xor     eax, eax
04000B6 74 FA                   jz      short near ptr _start+2
04000B8 E8 B8 01 00 00          call    near ptr 400275h
04000BD 00 BF 01 00 00+         add     [rdi+1], bh
04000C3 48 BE E0 00 60+         mov     rsi, offset Hello
04000CD BA 0D 00 00 00          mov     edx, 0Dh
04000D2 0F 05                   syscall                 ; $!
04000D4 B8 3C 00 00 00          mov     eax, 3Ch
04000D9 BF 00 00 00 00          mov     edi, 0
04000DE 0F 05                   syscall                 ; $!
04000DE                 _start  endp
04000DE
04000DE                 _text   ends
04000DE
06000E0                 ; ---------------------------------------------------------------------------
```

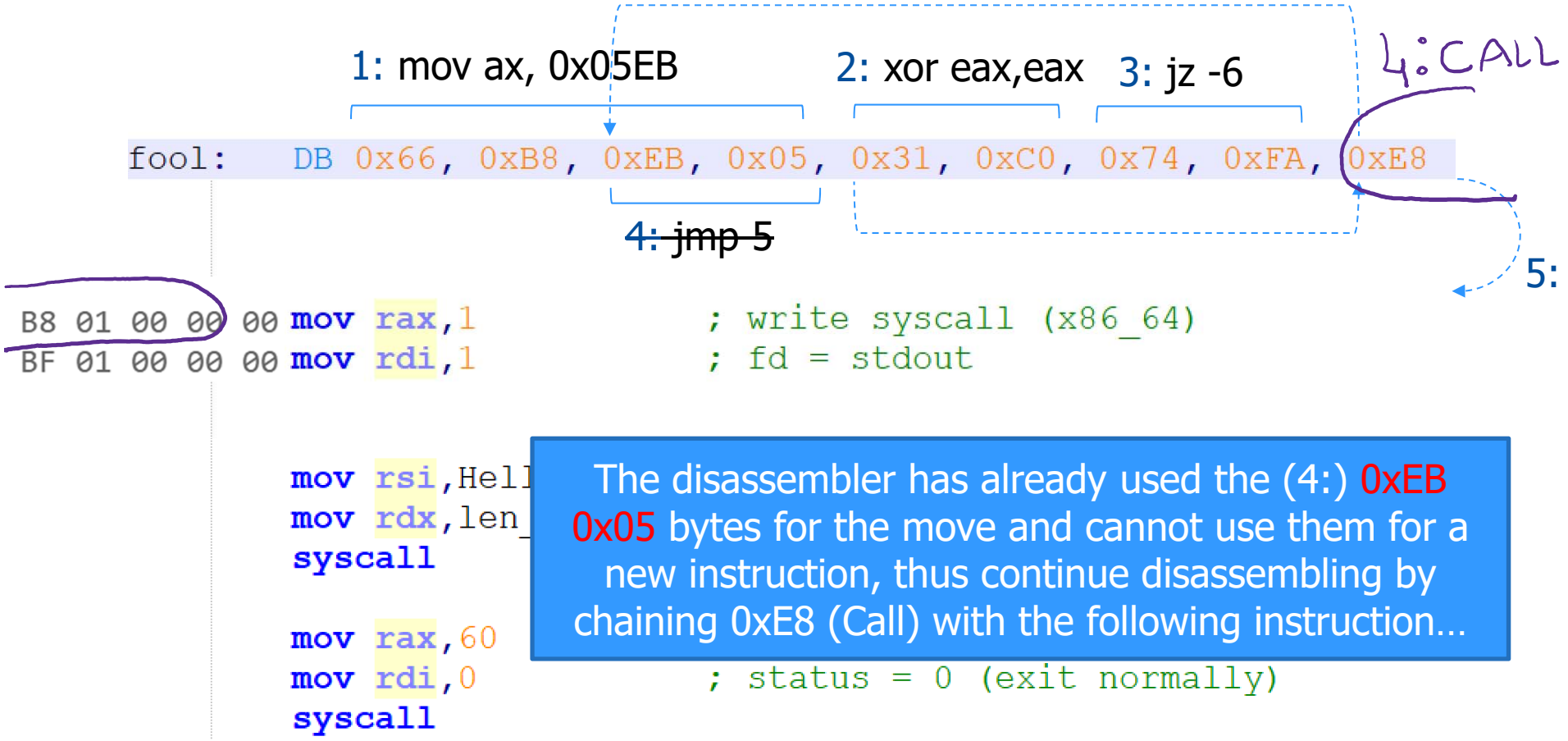
```
_start:
        mov rax,1
        mov rdi,1

        mov rsi,Hello
        mov rdx,len_Hello
        syscall

        mov rax,60
        mov rdi,0
        syscall
```

...WHAT'S GOING ON?




```
B8 01 00 00 00 mov rax,1              ; write syscall (x86_64)
BF 01 00 00 00 mov rdi,1              ; fd = stdout


               mov rsi,Hello          ; *buf = Hello
               mov rdx,len_Hello      ; count = len_Hello
               syscall

               mov rax,60             ; exit syscall (x86_64)
               mov rdi,0              ; status = 0 (exit normally)
               syscall
```

...WHAT'S GOING ON?



The disassembler has already used the (4:) 0xEB 0x05 bytes for the move and cannot use them for a new instruction, thus continue disassembling by chaining 0xE8 (Call) with the following instruction...

WHEN DISASSEMBLING

```
04000B0
04000B0                                 public _start
04000B0                         _start  proc near               ; CODE XREF: _start+6↓j
04000B0                                                         ; DATA XREF: LOAD:0000000000400018↑o
04000B0 66 B8 EB 05                     mov     ax, 5EBh        ; Alternative name is '_start'
04000B0                                                         ; fool
04000B4 31 C0                           xor     eax, eax
04000B6 74 FA                           jz      short near ptr _start+2
04000B8 E8 B8 01 00 00                  call    near ptr 400275h
04000BD 00 BF 01 00 00 00               add     [rdi+1], bh
04000C3 48 BE E0 00 60 00 00 00+        mov     rsi, offset Hello
04000CD BA 0D 00 00 00                  mov     edx, 0Dh
04000D2 0F 05                           syscall                 ; $!
04000D4 B8 3C 00 00 00                  mov     eax, 3Ch
04000D9 BF 00 00 00 00                  mov     edi, 0
04000DE 0F 05                           syscall                 ; $!
04000DE
04000DE
04000DE
04000DE
06000E0
06000E0
06000E0
06000E0                                 ; Segment permissions: Read/Write
06000E0
06000E0
06000E0
06000E0 48                                                                  D:0000000000400088↑o
```

mov rax, 1 ; B8 01 00 00 00 => CALL ... ; **EB** B8 01 00 00
mov rdi, 1 ; BF 01 00 00 00 => ADD ... ; 00 BF 01 00 00 00

In IDApro use D (data) and C (command) to fix the code

Additional Obfuscation Techniques

FUNCTION POINTER

1. Points to function sub_4011C0
2. and 3. still reference to the same function but they are not immediately recognized as such

```
004011D0 sub_4011D0      proc near               ; CODE XREF: _main+19p
004011D0                                         ; sub_401040+8Bp
004011D0
004011D0 var_4           = dword ptr -4
004011D0 arg_0           = dword ptr  8
004011D0
004011D0                 push    ebp
004011D1                 mov     ebp, esp
004011D3                 push    ecx
004011D4                 push    esi
004011D5                 mov     ❶[ebp+var_4], offset sub_4011C0
004011DC                 push    2Ah
004011DE                 call    ❷[ebp+var_4]
004011E1                 add     esp, 4
004011E4                 mov     esi, eax
004011E6                 mov     eax, [ebp+arg_0]
004011E9                 push    eax
004011EA                 call    ❸[ebp+var_4]
004011ED                 add     esp, 4
004011F0                 lea     eax, [esi+eax+1]
004011F4                 pop     esi
004011F5                 mov     esp, ebp
004011F7                 pop     ebp
004011F8                 retn
004011F8 sub_4011D0      endp
```

RETURN POINTER ABUSE

- Improperly used RET can be used as an alternative to JMP or CALL to redirect the flow

  CALL ⇔ PUSH EIP + JMP ref

  RET ⇔ POP EIP (EIP=ret_addr)

- The retn @0x00411C9 seems to close the sub it instead jumps @0x00411CA which seems to be a non reachable function
- call $+5 puts the EIP on the stack and jumps at it (0x004011C5)
- Then 5 is added to the RET address at esp (esp+4-4)
- retn doesn't returns to the call but jump to the actual function

```
004011C0 sub_4011C0      proc near               ; C
004011C0                                         ; s
004011C0
004011C0 var_4           = byte ptr -4
004011C0
004011C0                 call    $+5
004011C5                 add     [esp+4+var_4], 5
004011C9                 retn
004011C9 sub_4011C0      endp ; sp-analysis failed
004011C9
004011CA ; ---------------------------------------
004011CA                 push    ebp
004011CB                 mov     ebp, esp
004011CD                 mov     eax, [ebp+8]
004011D0                 imul    eax, 2Ah
004011D3                 mov     esp, ebp
004011D5                 pop     ebp
004011D6                 retn
```

MISUSING STRUCTURED EXCEPTION HANDLERS (SEH)

- Allows to manage exception conditions (e.g. divide by 0)
- Managed as a stack (last handler is first invoked)
- Every handler, if not managing the exception, invokes the previous in the stack (this->prev)
  - The last one opens an "unmanaged-exception" window
- FS:[0] points to the last one

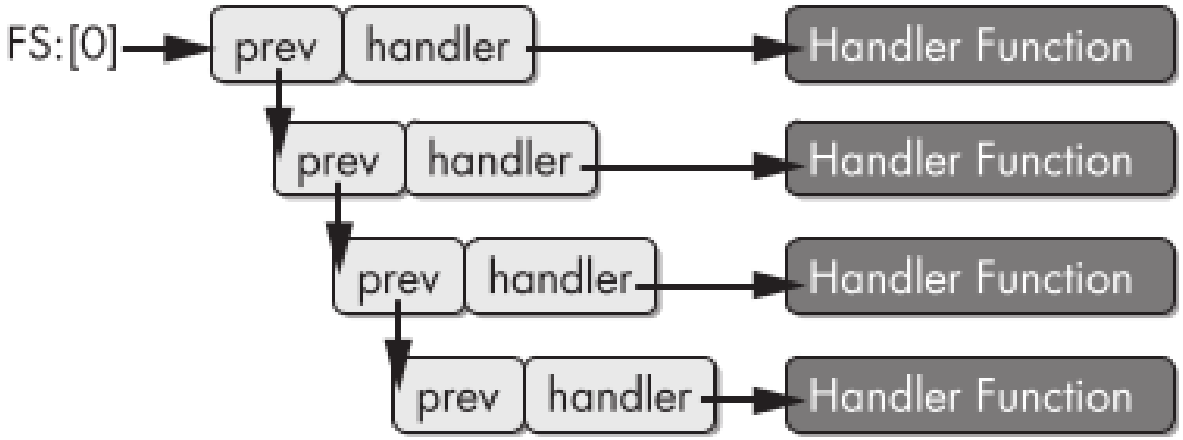


Figure 15-6: Structured Exception Handling (SEH) chain

```
struct _EXCEPTION_REGISTRATION {
    DWORD prev;
    DWORD handler;
};
```

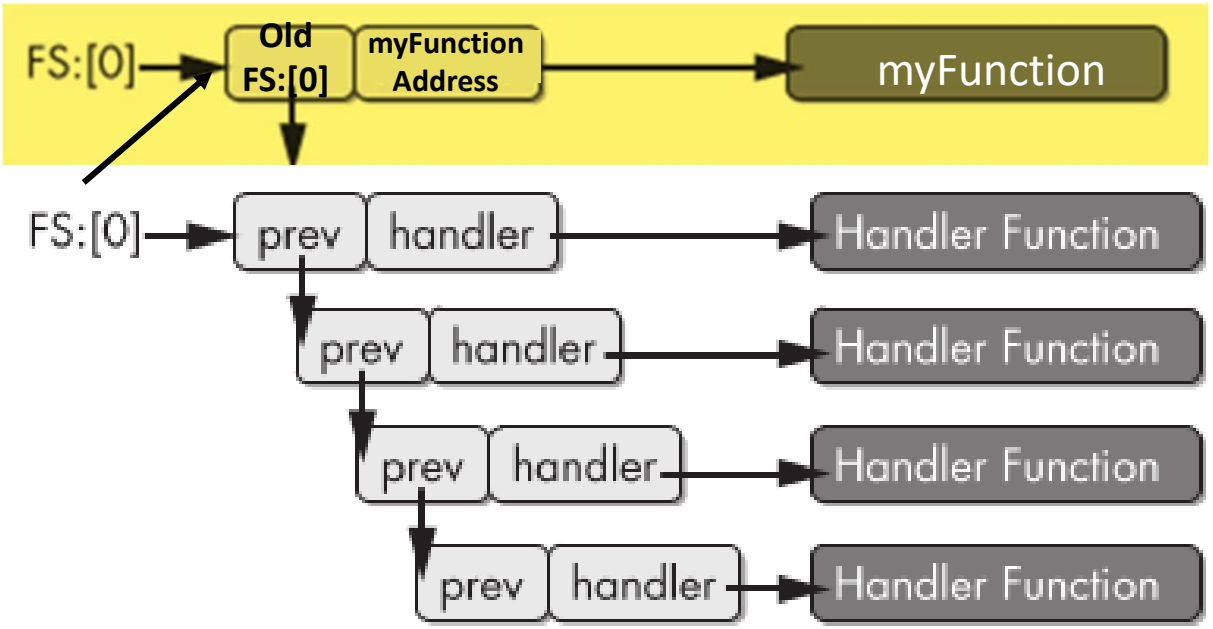


Figure 15-6: Structured Exception Handling (SEH) chain

EXPLOITING THE SEH

Prepare on the stack myFunc address: 40106Bh+1+14h=401080h

Push the old FS:[0]

register new SHE entry point (top of the stack)

Generate an exception (3) eax/ecx with ecx = 0

IDApro cannot follow exceptions thus do not see a reference to 401080 which is considered data

```
00401050                ❷mov     eax, (offset loc_40106B+1)
00401055                  add     eax, 14h
00401058                  push    eax
00401059                  push    large dword ptr fs:0 ; dwMillisecond
00401060                  mov     large fs:0, esp
00401067                  xor     ecx, ecx
00401069                ❸div     ecx
0040106B
0040106B loc_40106B:                                ; DATA XREF: sub_401
0040106B                  call    near ptr Sleep
00401070                  retn
00401070 sub_401050       endp ; sp-analysis failed
00401070
00401070 ; ---------------------------------------------------------------
00401071                  align 10h
00401080                ❶dd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
00401094                  dd 6808C483h
00401098                  dd offset aMysteryCode  ; "Mystery Code"
0040109C                  dd 2DE8h, 4C4830Oh, 3 dup(0CCCCCCCCh)
```

EXPLOITING THE SEH… IN PRACTICE

- Fixing the “fake data” with IDApro “C” button
- First restore the correct SEH structure
- Finally execute the actual code

```
00401080                mov     esp, [esp+8]
00401084                mov     eax, large fs:0
0040108A                mov     eax, [eax]
0040108C                mov     eax, [eax]
0040108E                mov     large fs:0, eax
00401094                add     esp, 8
00401097                push    offset aMysteryCode ; "Mystery Code"
0040109C                call    printf
```

```
0040106B
0040106B loc_40106B:                            ; DATA XREF: sub_401050o
0040106B                call    near ptr Sleep
00401070                retn
00401070 sub_401050     endp ; sp-analysis failed
00401070
00401070 ; ---------------------------------------------------------------
00401071                align 10h
00401080              ❶dd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
00401094                dd 6808C483h
00401098                dd offset aMysteryCode  ; "Mystery Code"
0040109C                dd 2DE8h, 4C48300h, 3 dup(0CCCCCCCCh)
```

- In Windows the Microsoft Software DEP (Data Execution Prevention) would prevent run time SEH changes
- Disable Safe SEH at compile /SAFESEH:NO (in C compilers)

EXPLOITING THE SEH... IN PRACTICE

- Fixing the "fake data" with IDApro "C" button
- First restore the correct SEH str
- Fir co
- In So Ex pr changes
- Disable Safe SEH at compile /SAFESEH:NO (in C compilers)

```
00401080        mov     esp, [esp+8]
00401084        mov     eax, large fs:0
0040108A        mov     eax, [eax]
0040108C        mov     eax, [eax]
```

In Radare2 it is possible to switch between data and code as follows:

Move to Visual mode (V and p)

Enter cursor mode (c) and use arrows to point to correct position

Press d and use the desired option from the menu

... d set as data

... c set as code

Use after positioning use shift + hjkl to select a block of bytes

THWARTING STACK-FRAME ANALYSIS

- IDApro tries to analyze the stack frame of a function (including possible code on the stack)
- cmp condition always false
  - Windows never uses first pages for stack
- Thus jl never taken but IDApro cannot know it and assumes that in the function esp is increased of 104h for local variables
- Actually first esp is increased by 4 (for two local variables), finally the real function is invoked