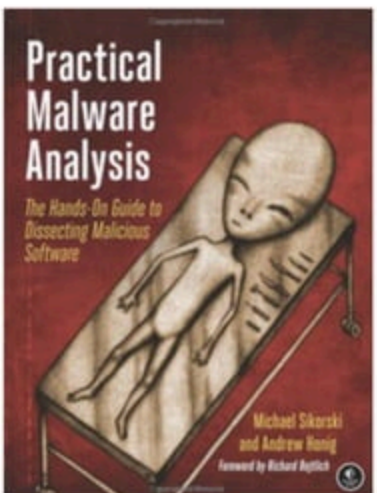
Practical Malware Analysis

Ch 15: Anti-Disassembly



Understanding Anti-Disassembly

Anti-Disassembly

- Specially crafted code or data
- Causes disassembly analysis tools to produce an incorrect listing
- Analysis requires more skill and/or time
- Prevents automated analysis techniques

Understanding Anti-Disassembly

- Assumptions and limitations of disassemblers
  - Each byte of a program can only be part of one instruction at a time
  - Tricking a disassembler into using the wrong offset will obscure a valid instruction

Wrong Offset

- jmp enters this code one byte past loc_2
- First instruction is not really a **call**
- **Linear disassembler** fails

```
                jmp     short near ptr loc_2+1
; ---------------------------------------------------------------------------

loc_2:                                  ; CODE XREF: seg000:00000000j
                call    near ptr 15FF2A71h
                or      [ecx], dl
                inc     eax
; ---------------------------------------------------------------------------
                db    0
```

Correct Disassembly

- **Flow-oriented disassembler** succeeds

```
                jmp     short loc_3
; ---------------------------------------------------------------------------
                db 0E8h
; ---------------------------------------------------------------------------

loc_3:                                  ; CODE XREF: seg000:00000000j
                push    2Ah
                call    Sleep
```

Defeating Disassembly Algorithms

Linear Disassembly

- Integrates through a block of bytes
- Disassembling one instruction at a time linearly, without deviating
- The size of an instruction determines which byte begins the next instruction
- Does not pay attention to flow-control instructions like **jmp**

Linear Disassembly Code

```
char buffer[BUF_SIZE];
int position = 0;

while (position < BUF_SIZE) {
   x86_insn_t insn;
   int size = x86_disasm(buf, BUF_SIZE, 0, position, &insn);

   if (size != 0) {
      char disassembly_line[1024];
        x86_format_insn(&insn, disassembly_line, 1024, intel_syntax);
        printf("%s\n", disassembly_line);
      1position += size;
   } else {
        /* invalid/unrecognized instruction */
      2position++;
      }
}
x86_cleanup();
```

- From The Bastard disassembler (Link Ch15a)

Linear Disassembly Flaws

- Always disassembles every byte
  - Even when only a portion of the code is used by flow-control code
- the **.text** section almost always includes some data as well as code

Example Case Instruction

```
            jmp     ds:off_401050[eax*4] ; switch jump

            ; switch cases omitted ...

            xor     eax, eax
            pop     esi
            retn
; ---------------------------------------------------------------------------
off_401050  dd offset loc_401020    ; DATA XREF: _main+19r
            dd offset loc_401027    ; jump table for switch statement
            dd offset loc_40102E
            dd offset loc_401035
```

- Runs function based on eax
- List of pointers at the end is data, not instructions

Linear Disassembly Output

- Misinterprets the table of pointers as instructions

```
and [eax],dl
inc eax
add [edi],ah
adc [eax+0x0],al
adc cs:[eax+0x0],al
xor eax,0x4010
```

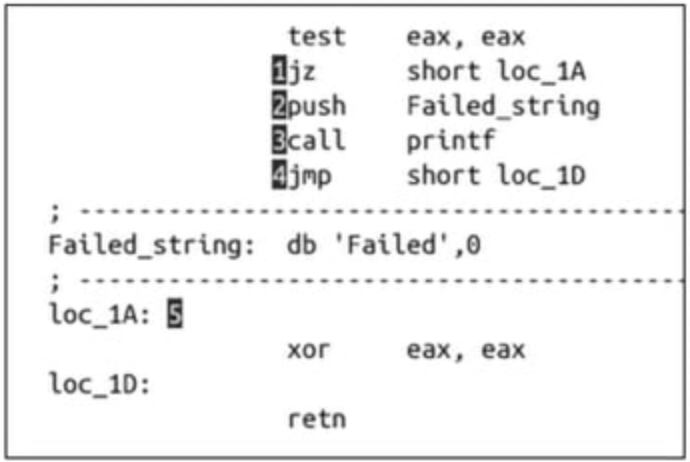
Multibyte Instructions

- **call**
  - 5 bytes long
  - 0xE8 followed by a 4-byte address
- Malware authors place bytes like 0xE8 in code to confuse linear disassemblers

Flow-Oriented Disassembly

- Used by most commercial disassemblers such as IDA Pro
- Doesn't assume the bytes are all instructions
- Examines each instruction and builds a list of locations to disassemble

Example



1. **jz** tells the assembler to start later at **loc_1A**
4. **jmp** tells assembler to start later at **loc_1D** and also to stop disassembling this byte series, since it's unconditional

Linear Disassembler

Mixes data and code together

Shows the wrong instructions

```
                test    eax, eax
                1jz     short loc_1A
                2push   Failed_string
                3call   printf
                4jmp    short loc_1D
; ---------------------------------------
Failed_string:  db 'Failed',0
; ---------------------------------------
loc_1A: 5
                xor     eax, eax
loc_1D:
                retn
```

```
                test    eax, eax
                jz      short near ptr loc_15+5
                push    Failed_string
                call    printf
                jmp     short loc_15+9
Failed_string:
                inc     esi
                popa
loc_15:
                imul    ebp, [ebp+64h], 0C3C03100h
```

Problematic Code

- Pointers
- Exceptions
- Conditional branches

Conditional Branches

- Give flow-oriented disassembler two places to disassemble
  - True branch and False branch
- They'd be the same in compiler-generated code
- But can have different disassembly in handwritten and anti-disassembly code
- Most flow-oriented disassemblers trust the false branch first

Using **call** to Get a String Pointer

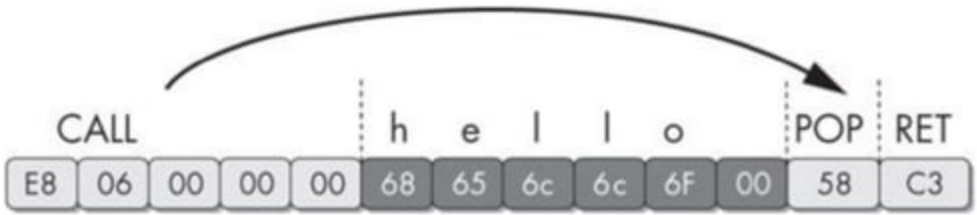


Figure 16-1. `call` instruction followed by a string

- This code puts a pointer to "hello" into eax
  - Because it's the return pointer (next eip value)

IDA Pro Output

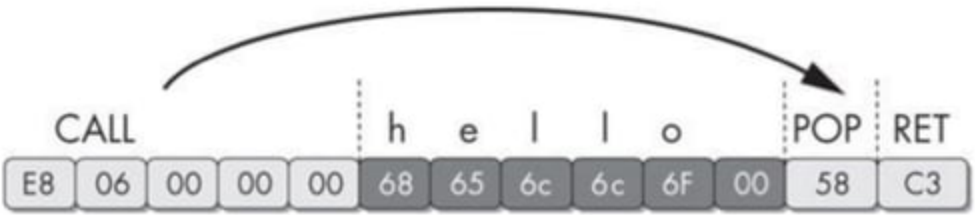


Figure 16-1. `call` instruction followed by a string

```
E8 06 00 00 00       call    near ptr loc_4011CA+1
68 65 6C 6C 6F      1push    6F6C6C65h

                    loc_4011CA:
00 58 C3            add     [eax-3Dh], bl
```

Manual Cleanup

- In IDA Pro
  - **C** turns cursor location to code
  - **D** turns cursor location to data

```
E8 06 00 00 00                      call    loc_4011CB
68 65 6C 6C 6F 00    aHello         db 'hello',0
                                    loc_4011CB:
58                                  pop     eax
C3                                  retn
```

Anti-Disassembly Techniques

Jump Instructions with the Same Target

- This code has the same effect as an unconditional jump
  - **jz loc_512**
  - **jnz loc_512**
- But IDA Pro doesn't see that, and continues disassembling the false branch of the second conditional jump

Fooling IDA Pro

```
74 03                   jz      short near ptr loc_4011C4+1
75 01                   jnz     short near ptr loc_4011C4+1
                        loc_4011C4:                     ; CODE XREF: sub_4011C0
                                                        ; 2sub_4011C0+2j
E8 58 C3 90 90         1call    near ptr 90D0D521h
```

- Actual instruction starts at **58**, not **E8**
- Cross-references to loc_4011C4 will appear in red
  - Because actual references point inside the instruction
  - A warning sign of anti-disassembly

Code Fixed with D Key

```
74 03                   jz      short near ptr loc_4011C5
75 01                   jnz     short near ptr loc_4011C5
        ; ---------------------------------------------------------------------
E8                      db 0E8h
        ; ---------------------------------------------------------------------
                        loc_4011C5:                     ; CODE XREF: sub_4011C0
                                                        ; sub_4011C0+2j
58                      pop     eax
C3                      retn
```

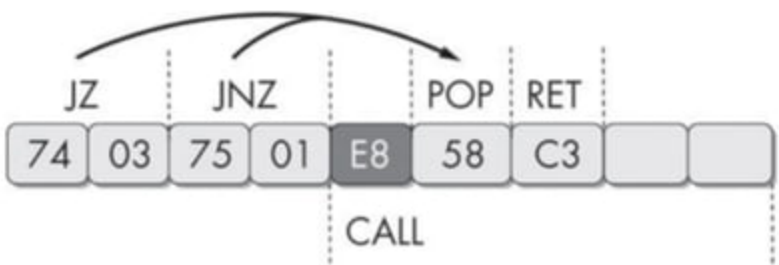


Figure 16-2. A jz instruction followed by a jnz instruction

A Jump Instruction with a Constant Condition

- Condition is always true
- IDA Pro sees a conditional jmp, but it's actually unconditional
- Processes the false branch first and trusts that result

```
33 C0                   xor     eax, eax
74 01                   jz      short near ptr loc_4011C4+1
        loc_4011C4:                             ; CODE XREF: 004011C2j
                                                ; DATA XREF: .rdata:004020ACo
E9 58 C3 68 94          jmp     near ptr 94A8D521h
```

Fixed with C and D Keys

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

Figure 16-3. False conditional of `xor` followed by a `jz` instruction

Impossible Disassembly

- Previous techniques use an extra byte after the jumps
  - A rogue byte
- In those examples, the rogue byte can be ignored

Re-Using a Byte

- **FF** is used twice
  - As an argument for a JMP
  - As an opcode for INC
- Disassemblers can't represent this situation

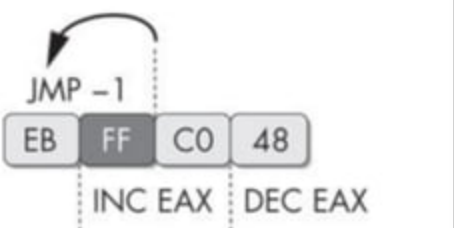


Figure 16-4. Inward-pointing `jmp` instruction

More Complex Example

- Dark-colored bytes are used twice
- Result of all this is **xor eax, eax**

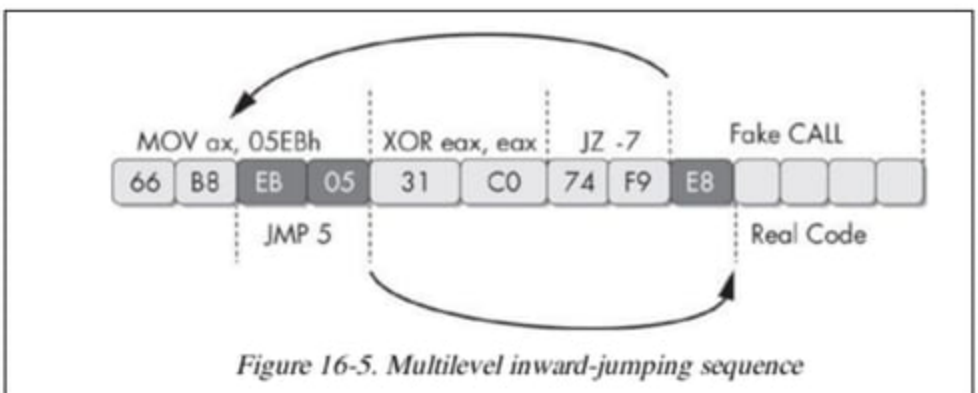


Figure 16-5. Multilevel inward-jumping sequence

NOP-ing Out Code

- Replacing the code with this sequence of bytes creates working code that IDA can understand

```
90                      nop
90                      nop
90                      nop
90                      nop
31 C0                   xor     eax, eax
90                      nop
90                      nop
90                      nop
58                      pop     eax
C3                      retn
```

Obscuring Flow Control

The Function Pointer Problem

- Here's a function at 4010C0
- Q: How many XREFs are there to this function?

```
004011C0 sub_4011C0      proc near               ; DATA XREF: sub_4011D0+5o
004011C0
004011C0 arg_0           = dword ptr  8
004011C0
004011C0                 push    ebp
004011C1                 mov     ebp, esp
004011C3                 mov     eax, [ebp+arg_0]
004011C6                 shl     eax, 2
004011C9                 pop     ebp
004011CA                 retn
004011CA sub_4011C0      endp
```

There are 3 actual references: **1**, **2**, **3**

But IDA can only find **1**

Cure: add manual comments

```
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     1[ebp+var_4], offset sub_4011C0
004011DC                 push    2Ah
004011DE                 call    2[ebp+var_4]
004011E1                 add     esp, 4
004011E4                 mov     esi, eax
004011E6                 mov     eax, [ebp+arg_0]
004011E9                 push    eax
004011EA                 call    3[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

- Normally, **call** and **jmp** are used to control flow
- However, **ret** can be abused to perform a **jmp**

Return Pointer Abuse

- **call** acts like two instructions
  - **push** return value
  - **jmp** into function
- **ret** acts like two instructions
  - **pop** return value
  - **jmp** to that address

Return Pointer Abuse Example

- The **retn** in the middle confuses IDA

```
004011C0 sub_4011C0      proc near               ; CODE XREF: _main+19p
004011C0                                         ; sub_401040+8Bp
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
```

Return Pointer Abuse Example

- Second half of code is a simple function
- Takes a value off the stack and multiplies it by 42

```
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
```

Return Pointer Abuse Example

- **call $+5** -- pushes 4011C5 onto the stack and then jumps there
- **add** -- adds 5 to the value at esp
  - Because +4 + var_4 = 0
- **ret** -- jumps to 4011CA

```
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
```

Fixing the Code

- Patch over the first three instructions with NOPs
- Adjust the function boundaries to cover the real function

Misusing Structured Exception Handlers

- SEH is a linked list
- Add an extra record to the top

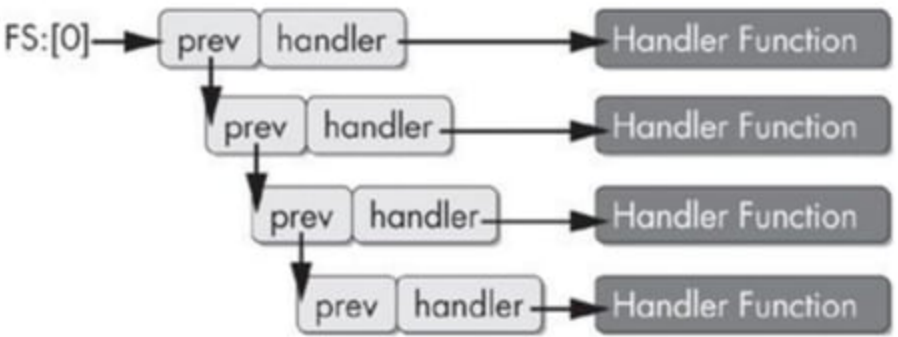


Figure 16-6. Structured Exception Handling (SEH) chain

1. Real subroutine is at 401080
2. eax set to 40106B + 1 + 14h= 401080, then added to the SEH
3. Divide by zero to trigger exception

```
00401050                2mov     eax, (offset loc_40106B+1)
00401055                 add     eax, 14h
00401058                 push    eax
00401059                 push    large dword ptr fs:0 ; dwMilliseconds
00401060                 mov     large fs:0, esp
00401067                 xor     ecx, ecx
00401069                3div     ecx
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                1dd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
00401094                 dd 6808C483h
00401098                 dd offset aMysteryCode  ; "Mystery Code"
0040109C                 dd 2DE8h, 4C48300h, 3 dup(0CCCCCCCCh)
```

Thwarting Stack-Frame Analysis

Stack Frame Analysis

- IDA has to decide how many stack bytes a function uses
- This is usually easy, the function prologue sets ebp and esp in an obvious way
- IDA adds the sizes of local variables to the size of the stack frame

Example

- The **cmp** is always false
- IDA thinks the stack frame is 104h bytes big, but it's much smaller

Example 16-1. A function that defeats stack-frame analysis

```
00401543     sub_401543      proc near               ; CODE XREF: sub_4012D0+3Cp
00401543                                             ; sub_401328+9Bp
00401543
00401543     arg_F4          = dword ptr  0F8h
00401543     arg_F8          = dword ptr  0FCh
00401543
00401543 000                 sub     esp, 8
00401546 008                 sub     esp, 4
00401549 00C                 cmp     esp, 1000h
0040154F 00C                 jl      short loc_401556
00401551 00C                 add     esp, 4
00401554 008                 jmp     short loc_40155C
00401556     ; ---------------------------------------------------------------
00401556
00401556     loc_401556:                          ; CODE XREF: sub_401543+Cj
00401556 00C                 add     esp, 104h
0040155C
```