26 - Anti-disassembly

Anti-disassembly: What is it, Really?
Defeating Disassembly Algorithms
Call Example
IDA Helps
Anti-Disassembly Techniques
Jump Instructions with Same Target
Jump with Constant Condition
Impossible Disassembly
Obscuring Control Flow: Function Pointers
Obscuring Control Flow: Return Pointer Abuse
Anti-disassembly: What is it, Really?

- Attempts to trick disassemblers into showing instructions that are not those that will actually be executed.
- Exploit assumptions used to perform disassembly (an inherently ill-posed problem).
- Two fundamental disassembly strategies:
  - Linear disassembly
  - Flow-oriented disassembly
- Both generic and target-specific anti-disassembly techniques can be employed.
Defeating Disassembly Algorithms

• Linear Disassembly
  - Disassembles one instruction at a time without deviating
  - Uses size of instructions to determine next byte
  - Generally works but may experience error even in some nonmalicious binaries.
  - May disassemble too much code (i.e., data, jump tables, etc.)

• Flow-oriented Disassembly
  - Used by commercial disassemblers (IDA)
  - Examines control flow of instructions to determine branch locations
  - Avoids disassembly after unconditional branch, retn, etc.
  - Branch prediction used (trust false branch of conditional branch)
  - Erroneous interpretation example: Call to get data in a register.
Call Example

• Consider this code

```
Call    'h' 'e' 'l' 'l' 'o' \n
E8 06 00 00 00 68 65 6c 6c 6f 00 58 C3
```

• This will probably be interpreted by either a flow or linear disassembler as

```
E8 06 00 00 00    call near ptr loc_4011CA+1
68 65 6C 6C 6F    push 6F6C6C65h
00 58 C3          add [eax-3Dh], bl
```
IDA Helps

- You can switch bytes from data to instructions or vice versa
  - C turns cursor location into code
  - D turns cursor location into data
Anti-Disassembly Techniques

• Use disassembler's assumptions to mislead it.
• Many (most) can be fixed by the malware analyst.
Jump Instructions with Same Target

• This is the code as IDA disassembles it:

```
74 03 jz   short near ptr loc_4011C4+1
75 01 jnz short near ptr loc_4011C4+1
loc_4011C4:
E8 58 C3 90 90 call near ptr 9D0D521h
```

• This exploits the “choose false branch as code” rule. The correct disassembly is this:

```
74 03 jz   short near ptr loc_4011C5
75 01 jnz short near ptr loc_4011C5
;----------------------------------------
E8 db E8h
;----------------------------------------
loc_4011C5:
58 pop eax
C3 retn
```
Jump with Constant Condition

- This is the code as IDA disassembles it:

  33 C0 xor eax,eax  
  74 01 jz short near ptr loc_0411C4+1  
  loc_4011C4:  
  E9 58 C3 68 94 jmp near ptr 94A8D521h  

- The jump will **always** be taken. The correct disassembly is as follows:

  33 C0 xor eax,eax  
  74 01 jz short near ptr loc_0411C5  
  ;----------------------------------------  
  E9 db0E9h  
  ;----------------------------------------  
  loc_4011C5:  
  58 pop eax  
  C3 retn
Impossible Disassembly

• Previous techniques used a rogue byte ($\text{E}8\text{h}$, $\text{E}9\text{h}$) that was never used in an instruction.

• What if the rogue is used in an instruction?

• Code:
  
  jmp -1
  
  $\text{inc eax dec eax}$
  $\text{EB FF C0 48}$

  $\text{Mov ax,053Bh xor eax,eax jz -6}$
  $\text{66 BB EB 05 31 C0 74 FA E8 (Real Code) jmp 5}$
Obscuring Control Flow Function Pointers

- Function Pointers are used extensively in C++ and to some extent in C. Disassemblers will not be able to statically identify locations of function pointers.

    mov [ebp+var_4], offset sub_4011C0
    push 24h
    call [ebp+var_4]

- Can use IDC scripts to set reference
  AddCodeXref(0x004011DE, 0x004011C0, f1_CF)
Obscuring Control Flow
Return Pointer Abuse

• Don't forget, retn does control flow too.

• Example:

```
004011C0 sub_4011C0    proc near
004011C0 var_4         = byte ptr -4
004011C0               call $+5
004011C5               add [esp+4+var_4],5
004011C9               retn
004011C9 sub_4011C0    endp ;
```

004011CA subroutine code
Next Time

• Read PMA Chapter 16
• More of Chapter 15 and some of Chapter 16