Anti-Reverse Engineering

Chapter 15
Understanding Anti-Disassembly

- Special code to cause disassembly analysis to produce incorrect program listings
  - Anti-disassembly techniques – delay/prevent analysis, increase the level of skill required of the malware analyst
  - Reduce the chances of malware being detected by static analysis (anti-virus software)

- Tricking disassembly at an incorrect offset
  - Jump to nonexistent target location (blank address, jump back to entry point, do this back and forth)
Adding Junk

Example (will see more)

```
PUSH EBP
MOV EBP, ESP
push edx
pop edx
inc ecx
dec ecx
add esp, 21
add esp, -21
add esp, 10
sub esp, 10
```

Nothing changes with these steps -> can mingle these ops with other ops (or even junk)

```
add eax, ebx
mul ecx
```

Jmp back to original entry point

```
xor esi, 01122344h ; garbage
add esi, eax ; garbage
add eax, ebx
mov edx, eax ; garbage
shl edx, 4 ; garbage
mul ecx
xor esi, ecx ; garbage
```
Linear Disassembler

Insert a data “hello” into the instructions (“special locations”)

1. Disassemble bytes follow call first, then called location
2. PUSH -> Op code 68

Correct using C or D keys (turn cursor into code/data)
Anti-Disassembly Techniques

- **Fake conditionals**
  - Back-to-back conditional jumps with the same target - jz followed by jnz should be treated as unconditional jmp

  ```
  normalcode
  jz label
  jnz label
  db thunkcode
  label:
  maliciouscode
  ```

  Disassembly does not realize it is unconditional jmp (one instr at time, not smart enough)

  Usually add 0xE8 (call) to make this become calling a function

  Label is 1 byte beyond 0xE8, execute call xxx in error (example 1 next)

- Confuses flow-disassemblers that choose fall-through path first and linear disassemblers

  ![Diagram]

  Insert junk 0xE8 – CALL
  Disassemble error
Example 1: 1 byte after loc_4011C4

Error from IDAPro: jump into a calling xxx

Use ‘D’ to remove the “E8” -> turn into DATA
Jump Instr. With a Const Cond.

- xor eax, eax -> clear eax and set the zero flag (implicit)
- Then jz is an unconditional jump -> zero flag is set
- However, disassembler will process the false branch first -> lead to a conflict
  - Write 0xE9 into junk code (jmp)

Jmp will be erroneously disassembled->hide the next 4 bytes of code
Jump Instr. With a Const Cond.

Use D key to turn jmp (junk code) into harmless data
Impossible Disassembly

- So far, add a byte into “strategic locations” and make the disassembler not working
- This “rogue byte” is not part of the program can be thrown off by changing into Data -> “D”
- What if the rogue byte is part of the program?
- Single byte is part of two instructions -> no disassembler can represent a single byte as being part of two instructions.
Impossible Disassembly

- Impossible disassembly
  - Using a single byte in two instructions (rogue byte)
  - Any given byte may be part of multiple instructions
  - Disassembler limited to picking one interpretation, but processor can use both
    - Inward jump

**Example:**

- FF is part of both instructions – EB FF/FF C0
- INC EAX/DEC EAX – NOP sequence
- Can be inserted anywhere to break the Disassembler (no effect on the code itself)
Impossible Disassembly

1. Mov ax, 05EBh -> populates ax register with data
2. Xor eax eax -> clear eax and set zero flag
3. Jz -7 -> unconditional jump to EB 05 -> as code means “jmp 5”

• No good way to clean the code; Leave the xor eax, eax, and set the rest as data
Nop-out (No operation)

- Make those “rogue bytes” into NOP instructions

```
Python> PatchByte(0x004012E7, 0x90)
True
Python> PatchByte(0x0040126D, 0x90)
True
Python> PatchByte(0x00401215, 0x90)
True
Python> PatchByte(0x004011D4, 0x90)
True
Python> PatchByte(0x0040115E, 0x90)
True
```

- Use the PatchByte in IDA command line
- 0x90 is the NOP Op code.
- Press “P” patch byte when you are done.