Phân tích Kỹ thuật Anti-Disassembly

(Chống Dịch ngược Mã máy)

Bùi Việt Dũng

Ngày 26 tháng 6 năm 2025

Tóm tắt nội dung

Tài liệu này trình bày các khái niệm cơ bản về disassembly (dịch ngược mã máy) và tập trung phân tích các kỹ thuật Anti-Disassembly (chống dịch ngược). Mục tiêu của các kỹ thuật này là gây nhiễu, đánh lừa các công cụ phân tích tĩnh như IDA Pro, Ghidra, khiến cho mã Assembly được tạo ra không chính xác hoặc khó hiểu, từ đó cản trở quá trình phân tích của các nhà nghiên cứu bảo mật và kỹ sư dịch ngược. Chúng ta sẽ đi sâu vào các phương pháp phổ biến, cách chúng khai thác những "giả định"của disassembler, và cách để khắc phục chúng.

Mục lục

1 Hiểu về Disassembly (Dịch ngược Mã máy)

1.1 Disassembly là gì?

Khi một chương trình được biên dịch từ mã nguồn (C, C++, Rust,...), trình biên dịch sẽ chuyển đổi nó thành **mã máy** (machine code) – một chuỗi các byte mà CPU có thể thực thi trực tiếp.

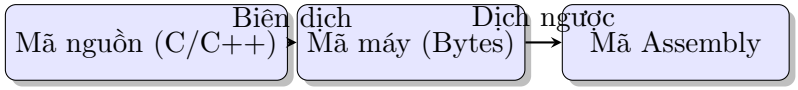
Ví dụ, một chuỗi byte như sau:

```
B8 01 00 00 00
```

Đối với con người, chuỗi byte này hoàn toàn vô nghĩa. Tuy nhiên, đối với CPU x86, nó tương ứng với một chỉ thị Assembly:

```
mov eax, 1
```

Disassembly là quá trình ngược lại: biến đổi mã máy (chuỗi byte) thành các chỉ thị Assembly mà con người có thể đọc và hiểu được. Quá trình này được thực hiện bởi các công cụ gọi là **disassembler**.



figureSơ đồ quy trình biên dịch và dịch ngược.

Các công cụ disassembler phổ biến bao gồm:

- **Phân tích tĩnh (Static Analysis):** IDA Pro, Ghidra, Radare2, objdump.
- **Trình gỡ lỗi (Debuggers):** x64Dbg, WinDbg, GDB (cũng có khả năng disassembly).

1.2 Các Thuật toán Disassembly chính

Disassembler hoạt động dựa trên các thuật toán để quyết định byte nào là code, byte nào là dữ liệu. Hai thuật toán phổ biến nhất là:

- **Linear Disassembly (Dịch ngược tuyến tính):** Thuật toán này bắt đầu từ một địa chỉ (ví dụ: entry point) và dịch ngược tuần tự từng byte một. Nó giả định rằng tất cả các byte trong một section code đều là mã lệnh. Điểm yếu của nó là dễ bị "lạc lối"nếu gặp phải dữ liệu được nhúng xen kẽ trong code.
- **Flow-Oriented Disassembly (Dịch ngược theo luồng):** Thuật toán này thông minh hơn. Nó cũng bắt đầu từ entry point, nhưng sau đó nó sẽ đi theo các luồng điều khiển (control flow) của chương trình, chẳng hạn như các lệnh nhảy (‘jmp‘), gọi hàm (‘call‘). Nó chỉ dịch ngược những gì nó tin là code có thể được thực thi. Đây là thuật toán được sử dụng bởi các công cụ hiện đại như IDA Pro và Ghidra.

2 Nguyên lý của Kỹ thuật Anti-Disassembly

Anti-Disassembly là tập hợp các kỹ thuật được sử dụng để khai thác những điểm yếu và các "giả định"của thuật toán disassembly, nhằm mục đích làm cho kết quả dịch ngược bị sai lệch, không đầy đủ hoặc khó hiểu.

2.1 Khai thác các "Giả định"của Disassembler

Disassembler không thực sự "hiểu"code. Chúng hoạt động dựa trên một bộ quy tắc và giả định:

- **Giả định về điểm bắt đầu:** Disassembler cho rằng một lệnh bắt đầu tại một địa chỉ cụ thể (ví dụ, sau lệnh trước đó hoặc tại một đích nhảy).
- **Giả định về ranh giới lệnh:** Nó giả định rằng các lệnh không chồng chéo lên nhau.
- **Giả định về luồng điều khiển:** Nó cố gắng đi theo các lệnh 'jmp' và 'call' để xác định các khối code hợp lệ.

Malware và các phần mềm được bảo vệ khai thác những giả định này bằng cách:

- **Chèn byte rác:** Thêm các byte vào những vị trí không bao giờ được thực thi để làm rối trình phân tích.
- **Nhảy vào giữa một lệnh:** Thực hiện một cú nhảy đến một địa chỉ nằm giữa một lệnh đã được dịch ngược trước đó.
- **Nhúng dữ liệu vào code:** Xen kẽ dữ liệu và code để đánh lừa thuật toán Linear Disassembly.
- **Làm rối luồng điều khiển:** Sử dụng các lệnh nhảy phức tạp, tự sửa đổi mã (self-modifying code) để che giấu luồng thực thi thực sự.

3 Các Kỹ thuật Anti-Disassembly Tĩnh

Phần này sẽ tập trung vào các kỹ thuật tĩnh, tức là có thể bị phát hiện và phân tích mà không cần chạy chương trình.

3.1 Kỹ thuật 1: Hai Lệnh Nhảy Điều kiện có cùng Đích đến

Đây là một trong những kỹ thuật phổ biến và đơn giản nhất. Kẻ tấn công đặt hai lệnh nhảy điều kiện đối nghịch nhau và cùng trỏ đến một địa chỉ.

3.1.1 Nguyên lý

Xét đoạn mã sau:

```
    jz  same_target
    jnz same_target
    ; Mã rác (junk code) bắt đầu từ đây
db 0xE8 ; Byte này trông giống như một lệnh CALL
    ...
same_target:
    ; Mã thực thi thực sự bắt đầu từ đây
    mov eax, ebx
    ...
```

- 'jz' (Jump if Zero): Nhảy nếu cờ Zero (ZF) được bật.
- 'jnz' (Jump if Not Zero): Nhảy nếu cờ Zero (ZF) bị tắt.

Bất kể trạng thái của cờ ZF là gì, một trong hai lệnh này **luôn luôn** được thực hiện. Do đó, luồng thực thi sẽ luôn nhảy đến ‘same$_t$arget‘.Œoncodenmngaysaulnh‘jnz‘skhngbaogiØcthcthi.

3.1.2 Phân tích của Disassembler

Flow-Oriented Disassembler như IDA Pro sẽ phân tích từng lệnh. Khi gặp ‘jnz same$_t$arget‘, nsgiØnhchaikhnng :

Nhánh đúng (True branch): Nhảy được thực hiện, đi đến ‘same$_t$arget‘.**Nhánh sai (False branch):**Nh
IDA Pro thường ưu tiên phân tích nhánh "fall-through"(nhánh sai) trước. Nó sẽ dịch ngược byte ‘0xE8‘ thành một lệnh ‘call‘ và tiếp tục phân tích sai từ đó, tạo ra một biểu đồ luồng điều khiển (graph view) sai lệch và khó hiểu.

```
.text:00401161 00
.text:00401168 73 03                    jnb     short near ptr loc_40116C+1
.text:0040116A 72 01                    jb      short near ptr loc_40116C+1
.text:0040116C
.text:0040116C                  loc_40116C:                             ; CODE XREF: .text:00401168↑j
.text:0040116C                                                          ; .text:0040116A↑j
.text:0040116C E8 64 A1 30 00           call    near ptr 70B2D5h
```

Hình 1: IDA Pro dịch ngược sai do kỹ thuật hai lệnh nhảy cùng đích. Byte `0xE9` bị hiểu nhầm là một lệnh `JMP` thay vì là dữ liệu rác.

Một dấu hiệu nhận biết là mũi tên chỉ đến đích nhảy (‘loc$_4$01161‘tronghnh)scmuØ.ŒiunybohiurngIDAPro

3.1.3 Giải pháp của Nhà phân tích

Chúng ta cần "hướng dẫn"lại cho IDA Pro.

1. Đặt con trỏ vào dòng lệnh rác ngay sau ‘jnz‘.
2. Nhấn phím **‘D‘ (Data)** để chuyển đổi các byte này thành dữ liệu, báo cho IDA biết đây không phải là code.

3. Đặt con trỏ vào địa chỉ đích thực sự (‘loc$_4$01161‘).Nhnphm**‘C‘ (Code)**ØbucIDAdchngclitØachnythnhcode.

```
.text:00401168 73 03                    jnb     short loc_40116D
.text:0040116A 72 01                    jb      short loc_40116D
.text:0040116A                  ; ---------------------------------------------------------------------------
.text:0040116C E8                       db 0E8h
.text:0040116D                  ; ---------------------------------------------------------------------------
.text:0040116D
.text:0040116D                  loc_40116D:                             ; CODE XREF: .text:00401168↑j
.text:0040116D                                                          ; .text:0040116A↑j
.text:0040116D 64 A1 30 00 00 00        mov     eax, large fs:30h
.text:00401173 33 DB                    xor     ebx, ebx
```

Hình 2: Kết quả sau khi đã sửa lỗi bằng phím ‘D‘ và ‘C‘. Byte `0xE9` giờ được định nghĩa là dữ liệu (‘db 0E9h‘), và mã lệnh thực sự tại đích nhảy đã được dịch ngược chính xác.

*Mẹo: Để hiển thị các byte mã máy như trong hình, vào **Options -> General -> Number of opcode bytes** và nhập một số lớn hơn 0.*

3.2 Kỹ thuật 2: Lệnh Nhảy Điều kiện với Điều kiện Hằng

Kỹ thuật này tạo ra một lệnh nhảy điều kiện mà kết quả của nó có thể được dự đoán trước một cách chắc chắn.

3.2.1 Nguyên lý

Xét đoạn mã sau:

```
    xor eax, eax  ; Luôn luôn làm cho eax = 0, và đặt cờ ZF = 1
    jz  real_code ; Nhảy nếu ZF=1. Lệnh này LUÔN LUÔN được thực thi
    ; Mã rác bắt đầu ở đây
db 0xCC          ; Lệnh INT3 (breakpoint), dùng để gây nhiễu
    ...
real_code:
    ; Mã thực sự
    push ebp
    ...
```

Lệnh 'xor eax, eax' là một cách phổ biến để gán giá trị 0 cho thanh ghi 'eax'. Một hiệu ứng phụ quan trọng của nó là nó luôn **bật cờ Zero (ZF=1)**. Lệnh 'jz real$_{c}$ode'ngaysauØslunØcthcthi.Tngtnhkthuttrc, Øoncod

```
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
```

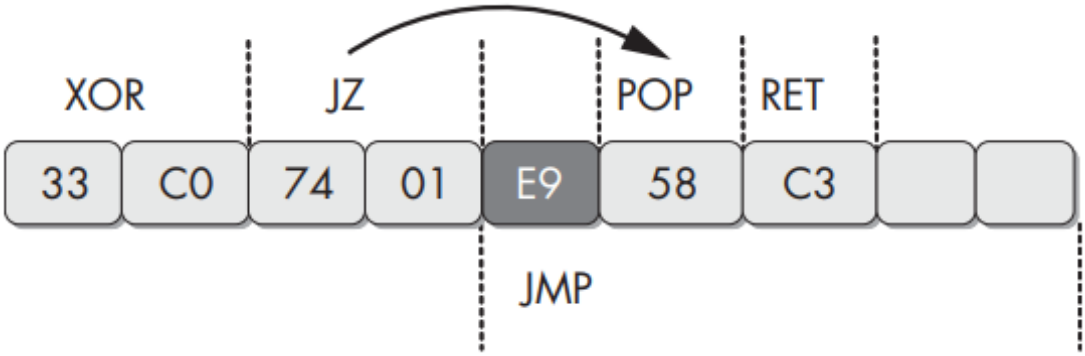
Hình 3: Disassembler bị lừa bởi lệnh nhảy với điều kiện hằng.

3.2.2 Giải pháp của Nhà phân tích

Giải pháp tương tự như trên: sử dụng phím 'D' để định nghĩa phần mã rác là dữ liệu và phím 'C' để định nghĩa lại mã tại đích nhảy nếu cần.

```
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
```

Hình 4: Đoạn mã sau khi được sửa chữa.



Hình 5: Biểu đồ luồng điều khiển cho thấy nhánh sai (false branch) được tạo ra để gây nhiễu.

3.3 Kỹ thuật 3: "Impossible Disassembly"(Chồng lấn Lệnh)

Đây là một kỹ thuật nâng cao hơn, tạo ra tình huống mà disassembler tĩnh không thể biểu diễn chính xác tất cả các luồng thực thi trên cùng một khung nhìn.

3.3.1 Nguyên lý

Kỹ thuật này dựa trên việc một byte có thể là một phần của **hai hoặc nhiều lệnh khác nhau**, tùy thuộc vào luồng thực thi. CPU không gặp vấn đề gì với việc này vì nó chỉ đọc lệnh từ địa chỉ mà con trỏ lệnh (EIP) đang trỏ tới. Tuy nhiên, disassembler tĩnh cố gắng ánh xạ mỗi byte tới một lệnh duy nhất, và do đó sẽ thất bại.

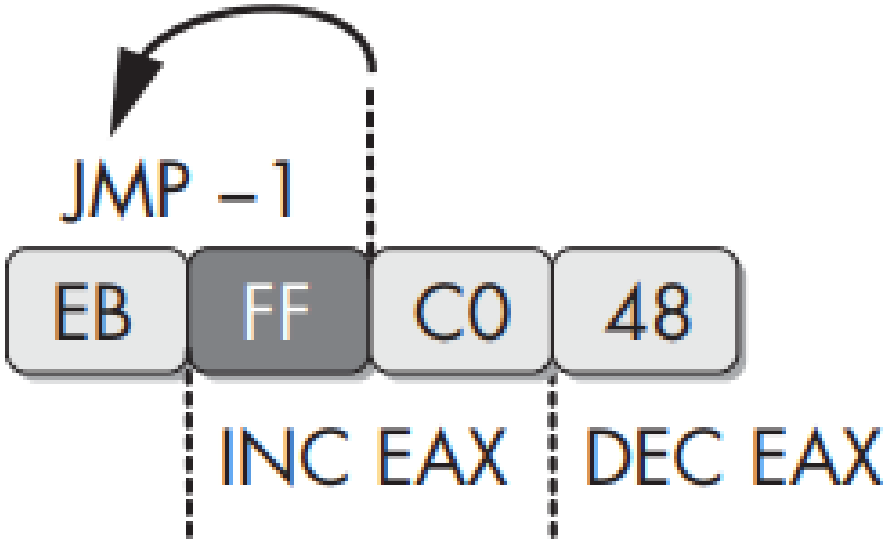
Ví dụ đơn giản:

```
; Địa chỉ | Bytes      | Lệnh
; -----------------------------------
; 0x401000 | EB FF      | jmp 0x401001
; 0x401002 | 40         | inc eax
; 0x401003 | 40         | inc eax
```

Trong ví dụ trên, lệnh ‘jmp 0x401001‘ tại địa chỉ ‘0x401000‘ có 2 byte là ‘EB FF‘. Lệnh này sẽ nhảy đến ‘0x401001‘, tức là nhảy vào **byte thứ hai của chính nó**.

- Byte ‘FF‘ tại ‘0x401001‘ và byte ‘40‘ tại ‘0x401002‘ sẽ được CPU diễn giải thành một lệnh mới: ‘inc ecx‘ (mã máy là ‘FF C1‘, giả sử ‘40‘ là ‘C1‘ trong ngữ cảnh khác) hoặc một lệnh khác tùy vào byte tiếp theo.

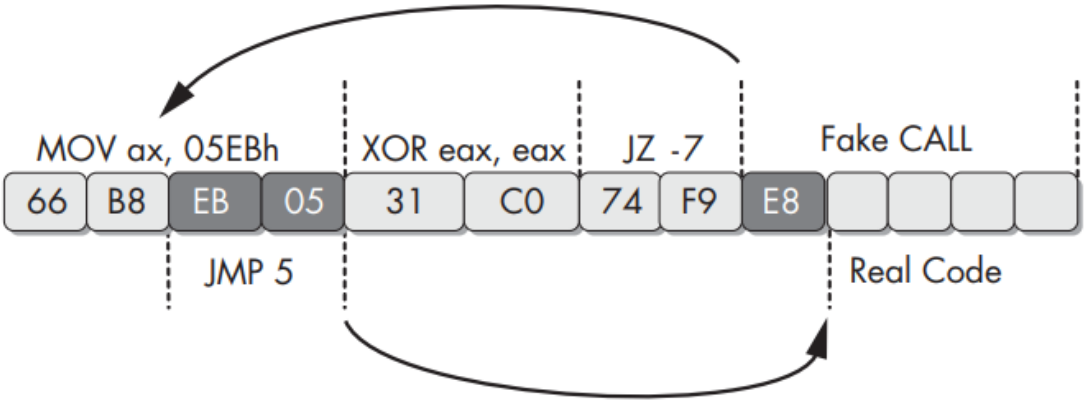
Đây là một tình huống mà byte ‘FF‘ vừa là toán hạng của lệnh ‘jmp‘, vừa là opcode của một lệnh khác. IDA Pro sẽ chỉ có thể hiển thị một trong hai cách diễn giải này.



Hình 6: Một lệnh `jmp` nhảy vào chính giữa nó, tạo ra một luồng thực thi mới.

3.3.2 Ví dụ phức tạp

Hãy xem xét một chuỗi lệnh phức tạp hơn, nơi nhiều byte bị dùng chung.



Hình 7: Một chuỗi lệnh chồng lấn đa cấp.

Phân tích chuỗi trong Hình 7:

1. **‘mov ax, 0E905h‘ (Bytes: ‘66 B8 05 E9‘):** Lệnh này nạp giá trị ‘0E905h‘ vào thanh ghi ‘AX‘.
2. **‘xor eax, eax‘:** Lệnh này được thực thi, reset ‘EAX‘ về 0 và bật cờ ZF.
3. **‘jz short loc**$_4011C2$‘ : $Lnhny$**luôn nhảy**$Øn‘0x4011C2‘$.**Luồng thực thi thật:**$ContrlnhnhyØn‘0x$

4. **Lệnh ẩn:** Hai byte ‘05 E9‘ được CPU diễn giải là một lệnh mới: ‘jmp 0x004011CC‘ (nhảy tương đối 5 byte).

Kết quả: Disassembler sẽ hiển thị sai vì nó không thể biểu diễn việc hai byte ‘05 E9‘ vừa là toán hạng của ‘mov‘, vừa là một lệnh ‘jmp‘ độc lập. Nó sẽ cố gắng dịch ngược đoạn mã rác sau lệnh ‘jz‘.

```
66 B8 EB 05             mov     ax, 5EBh
31 C0                   xor     eax, eax
74 F9                   jz      short near ptr sub_4011C0+1
                loc_4011C8:
E8 58 C3 90 90          call    near ptr 98A8D525h
```

Hình 8: Cách IDA Pro hiển thị đoạn mã chồng lấn ban đầu - hoàn toàn sai lệch.

3.3.3 Giải pháp: NOP-ing Out (Vô hiệu hóa bằng NOP)

Trong trường hợp này, việc chỉ dùng ‘D‘ và ‘C‘ có thể không đủ để làm cho mã nguồn trở nên dễ đọc. Một giải pháp tốt hơn là vá (patch) các byte không cần thiết hoặc gây nhiễu thành lệnh ‘NOP‘ (No Operation, mã máy là ‘0x90‘). Lệnh ‘NOP‘ không làm gì cả, chỉ đơn giản là cho CPU đi tiếp.

Chúng ta có thể sử dụng script IDAPython để tự động hóa việc này.

```python
import idaapi

def NopBytes(start_ea, length):
    """Vô hiệu hóa một vùng nhớ bằng cách ghi đè các byte bằng 0x90 (NOP)."""
    for i in range(length):
        idaapi.patch_byte(start_ea + i, 0x90)
    print(f"NOP-ed out {length} bytes starting at {hex(start_ea)}")

# Các vùng nhớ cần NOP-out trong ví dụ
addr1 = 0x004011C0
len1 = 4
addr2 = 0x004011C6
len2 = 3

# Thực hiện vá lỗi
NopBytes(addr1, len1)
NopBytes(addr2, len2)

# Yêu cầu IDA phân tích lại vùng đã thay đổi
idaapi.auto_wait()
```

figureScript IDAPython để vá các byte gây nhiễu thành lệnh NOP.

Sau khi chạy script, mã assembly sẽ trở nên sạch sẽ và logic hơn rất nhiều, chỉ còn lại những lệnh thực sự được thực thi.

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

Hình 9: Kết quả disassembly sau khi đã NOP-out các lệnh rác và chồng lấn.

4 Tự động hóa với Scripting

Việc vá lỗi thủ công có thể tốn thời gian. Chúng ta có thể viết một script IDAPython đơn giản để gán một phím nóng (hotkey) cho hành động NOP-out một lệnh.

```python
import idaapi
import idautils

class NopInstructionHandler(idaapi.action_handler_t):
    def __init__(self):
        idaapi.action_handler_t.__init__(self)

    def activate(self, ctx):
        # Lấy địa chỉ hiện tại của con trỏ
        curr_addr = idaapi.get_screen_ea()
        if curr_addr == idaapi.BADADDR:
            return 1

        # Lấy kích thước của lệnh tại địa chỉ đó
        instr_size = idautils.decode_insn(curr_addr)
        if instr_size == 0:
            print(f"Error: Could not decode instruction at {hex(curr_addr)}")
            return 1

        # Thực hiện NOP-out
        for i in range(instr_size):
            idaapi.patch_byte(curr_addr + i, 0x90)

        # Di chuyển con trỏ đến lệnh tiếp theo
        idaapi.jumpto(curr_addr + instr_size)
        return 1

    def update(self, ctx):
        return idaapi.AST_ENABLE_ALWAYS

# Đăng ký hành động và gán phím nóng
```

```
action_name = "NopOutInstruction"
action_label = "NOP out current instruction"
hotkey = "Alt-N"

idaapi.register_action(
    idaapi.action_desc_t(
        action_name,
        action_label,
        NopInstructionHandler(),
        hotkey
    )
)
print(f"Hotkey '{hotkey}' is now set to NOP-out the current instruction.")
```

figureScript IDAPython tạo phím nóng `ALT-N` để NOP-out lệnh tại vị trí con trỏ.

Sau khi chạy script này, mỗi khi bạn nhấn ‘ALT-N‘ trong IDA Pro, lệnh tại con trỏ sẽ được chuyển thành các ‘NOP‘, giúp quá trình dọn dẹp mã trở nên nhanh chóng hơn rất nhiều.

5 Kết luận

Anti-disassembly là một cuộc chiến "mèo vờn chuột"giữa người viết phần mềm (đặc biệt là malware) và các nhà phân tích bảo mật. Bằng cách hiểu rõ cách các disassembler hoạt động và những giả định mà chúng đưa ra, chúng ta có thể nhận diện và vô hiệu hóa các kỹ thuật chống dịch ngược này. Các kỹ thuật từ đơn giản như nhảy điều kiện có chủ đích đến phức tạp như chồng lấn lệnh đều có thể được khắc phục bằng sự kết hợp giữa phân tích thủ công và tự động hóa bằng script.

Tài liệu tham khảo

Toàn bộ kiến thức và hình ảnh minh họa trong tài liệu này được tham khảo và trích dẫn từ sách: **Sikorski, M., & Honig, A. (2012).** ***Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*****. No Starch Press.**