Mã độc

Chương 3. Dịch ngược mã độc

Mục tiêu

- Nhắc lại một số kiến thức cơ bản về hợp ngữ
- Giới thiệu, hướng dẫn sinh viên sử dụng công cụ IDA pro và các chức năng chính

Tài liệu tham khảo

[1] Michael Sikorski, Andrew Honig, 2012, Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software, No Starch Press, (ISBN: 978-1593272906).

[2] Sam Bowne, Slides for a college course at City College San Francisco, https://samsclass.info/126/126_S17.shtml

Nội dung

1. Nhắc lại về Assembly
2. Sử dụng IDA pro để dịch ngược mã độc
3. Sử dụng đối sánh chéo
4. Phân tích hàm
5. Sử dụng biểu đồ hàm
6. Một số lưu ý

Nội dung

1. Nhắc lại về Assembly
2. Sử dụng IDA pro để dịch ngược mã độc
3. Sử dụng đối sánh chéo
4. Phân tích hàm
5. Sử dụng biểu đồ hàm
6. Một số lưu ý

Nhắc lại về Assembly

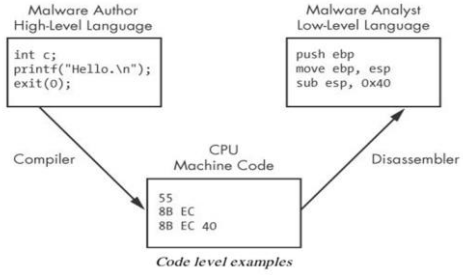
- ❑ Các mức trừu tượng của ngôn ngữ
- ❑ Reverse Engineering
- ❑ Kiến trúc x86
- ❑ Một số cấu trúc đơn giản

Nhắc lại về Assembly

- ❑ Các mức trừu tượng của ngôn ngữ
- ❑ Reverse Engineering
- ❑ Kiến trúc x86
- ❑ Một số cấu trúc đơn giản

7

Các mức trừu tượng của ngôn ngữ



Code level examples

8

Các mức trừu tượng của ngôn ngữ

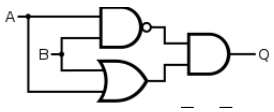
Sáu mức trừu tượng:

- ❑ Hardware
- ❑ Microcode
- ❑ Machine code
- ❑ Low-level languages
- ❑ High-level languages
- ❑ Interpreted languages

9

Hardware

- ❑ Các mạch số
- ❑ Cổng logic: AND, OR, NOT, XOR...
- ❑ Không dễ dàng thao tác được bằng phần mềm



10

Microcode

- ❑ Còn được gọi là Firmware (phần sụn)
- ❑ Chỉ hoạt động trên một vài phần cứng cụ thể

11

Machine code

Mã máy: Dạng ngôn ngữ mà chỉ máy tính mới hiểu được (Binary, Hexa...)

Opcodes (cũng được hiểu là mã máy):

- ❑ Được tạo ra khi biên dịch một chương trình từ ngôn ngữ bậc cao (như C/C++...)
- ❑ Là những chỉ thị để ra lệnh cho CPU làm một công việc
- ❑ Có thể được tái tạo lại thành dạng hợp ngữ - assembly để con người có thể đọc hiểu
- ❑ VD: 90 – NOP, 88 – MOV, FF – PUSH...

12

Low-level languages

- Dạng ngôn ngữ mà con người có thể đọc hiểu được
- Hợp ngữ - Assembly language

Vd: MOV, NOP, PUSH, POP, JMP...

- Disassembly là quá trình tái tạo lại ngôn ngữ máy thành Assembly, để con người có thể đọc hiểu
- Quá trình này cần một trình Disassembler.

13

Low-level languages

- Kết quả tái tạo không phải lúc nào cũng chính xác tuyệt đối.
- Assembly là dạng ngôn ngữ cao nhất có thể tái tạo từ các chương trình sử dụng các ngôn ngữ biên dịch khi mà mã nguồn của phần mềm độc hại không phải lúc nào cũng có.

14

High-level languages

- Là dạng ngôn ngữ lập trình phổ biến

VD: C, C++, etc...

- Quá trình chuyển đổi sang mã máy cần thông qua một trình biên dịch (Compiler).

15

Interpreted languages

- Thuộc dạng ngôn ngữ bậc cao
- VD: Java, C#, Perl, .NET, Python...
- Mã nguồn không biên dịch trực tiếp sang mã máy mà nó được biên dịch sang bytecode. Sau đó trình thông dịch của ngôn ngữ đó mới dịch bytecode sang mã máy.

16

Interpreted languages

- Bytecode còn được gọi là mã trung gian, độc lập với phần cứng và hệ điều hành
- Trình thông dịch sẽ dịch bytecode sang mã máy trong quá trình chạy và máy tính sẽ thực thi chúng
- VD: Java Bytecode sẽ chạy trong JVM (Java Virtual Machine)

17

Nhắc lại về Assembly

- Các mức trừu tượng của ngôn ngữ
- Reverse Engineering
- Kiến trúc x86
- Một số cấu trúc đơn giản

18

Reverse Engineering

- Các phần mềm độc hại tồn tại trên ổ cứng ở dạng tệp nhị phân (nằm ở mức Machine code).
- Quá trình Disassembly sẽ tái tạo và chuyển đổi phần mềm độc hại ở dạng nhị phân về dạng hợp ngữ (Assembly).
- IDA Pro là một trình Disassembler phổ biến trong việc dịch ngược.

19

Reverse Engineering

- Linear Disassembly: dịch ngược các lệnh (Instruction) một cách tuần tự, sử dụng kích thước của lệnh đã được dịch ngược để xác định byte nào sẽ được dịch ngược tiếp theo, không quan tâm đến các lệnh điều khiển luồng.
- Flow-Oriented Disassembly: dịch ngược theo luồng hoạt động của chương trình.

20

Linear Disassembly

- Thường được sử dụng trong trình gỡ lỗi và trình dịch ngược cơ bản.
- Kích thước của lệnh đã được dịch ngược được sử dụng để tính toán byte tiếp theo trong quá trình dịch ngược .
- Bỏ qua tất cả luồng điều khiển chương trình

21

Linear Disassembly

- Nhược điểm là dịch ngược quá nhiều lệnh vì quá trình dịch ngược thực hiện cho đến hết phần mã.
- Dữ liệu trong phần mã sẽ gây ra các vấn đề như giá trị con trỏ và bảng nhảy.

22

Flow-Oriented Disassembly

- Không cho rằng tất cả dữ liệu là lệnh.
- Khi gặp lệnh nhảy không điều kiện, địa chỉ đích được lưu và quá trình dịch ngược bắt đầu trở lại tại địa chỉ đã lưu trước đó.
- Khi gặp phải lệnh nhảy có điều kiện, địa chỉ đích được lưu lại để dịch ngược sau khi quá trình dịch ngược tuần tự kết thúc.

23

Flow-Oriented Disassembly

- Lệnh call được xử lý tương tự như lệnh nhảy có điều kiện.
- Flow-Oriented Disassembly có thể dịch ngược một cách chính xác khi các lệnh tuân theo một lệnh nhảy vô điều kiện.
- Hầu hết Flow-Oriented Disassembly trước tiên sẽ xử lý nhánh sai của các lệnh rẽ nhánh có điều kiện

24

Assembly Language

- Mỗi bộ vi xử lý khác nhau sẽ có những tập lệnh assembly khác nhau
- Kiến trúc x86 – 32 bits, Kiến trúc x64 – 64 bits
- Hệ điều hành Windows chạy x86 hoặc x64
- Hầu hết các mã độc đều được thiết kế chạy trên x86.

25

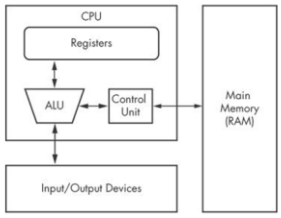
Nhắc lại về Assembly

- Các mức trừu tượng của ngôn ngữ
- Reverse Engineering
- Kiến trúc x86
- Một số cấu trúc đơn giản

26

Kiến trúc máy tính Von Neumann

- CPU (Central Processing Unit) sẽ thực thi các mã lệnh,
- RAM lưu trữ dữ liệu và mã lệnh,
- Hệ thống vào-ra kết nối với các thiết bị ngoại vi: bàn phím, màn hình, đĩa cứng...



Von Neumann architecture

27

CPU Components

- Khối CU - Control Unit
  - Tìm và nạp các mã lệnh từ bộ nhớ RAM, sử dụng thanh ghi có tên là IP/EIP (Instruction Poiter)
- Khối ALU - Arithmetic Logic Unit
  - Thực hiện việc tính toán số học, đưa kết quả vào thanh ghi hoặc bộ nhớ RAM

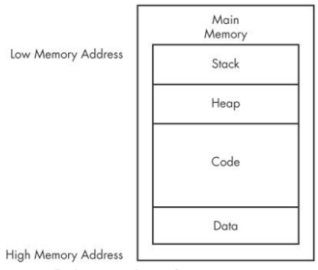
28

CPU Components

- Thanh ghi - Register
  - Vùng nhớ nằm bên trong CPU, lưu trữ dữ liệu đợi CPU xử lý
  - Tốc độ của Register là rất nhanh, nhanh hơn nhiều lần so với bộ nhớ RAM

29

Main Memory (RAM)



Basic memory layout for a program

30

Data Segment (RAM)

- Lưu các giá trị của chương trình khi được nạp vào RAM.
- Các giá trị tĩnh (biến static), không thể thay đổi trong khi chương trình đang chạy.
- Chứa các biến toàn cục, hằng số... (phạm vi toàn chương trình).

31

Code Segment (RAM)

- Chứa mã nguồn đã được biên dịch của chương trình (Các chỉ dẫn CPU thực thi)
- Kiểm soát chương trình làm việc

32

Heap Segment (RAM)

- Vùng nhớ dùng trong cấp phát động (malloc, new...),
- Vùng nhớ này thay đổi thường xuyên trong quá trình thực hiện chương trình,
- Khi không có nhu cầu sử dụng cần phải giải phóng (free, delete...).

33

Stack Segment (RAM)

- Vùng nhớ lưu chứa các biến cục bộ trong hàm, các tham số truyền vào cho hàm, các giá trị trả về của hàm...
- Vùng nhớ quản lý bởi CPU.

34

Instructions

- Một lệnh (Instruction) bao gồm: toán hạng đích, nguồn và tên lệnh
- VD: MOV ECX, 0x42
  - Tên lệnh: MOV – Lệnh di chuyển giá trị 0x42 (hexa) vào thanh ghi ECX,
  - Giá trị 0x42 ở dạng thập lục phân (hex).
- Lệnh này ở dạng nhị phân (mã máy) sẽ có dạng: 0xB942000000

35

Endianness

- Thuật ngữ Big-Endian và Little-Endian diễn tả sự khác nhau về cách đọc và ghi dữ liệu giữa các nền tảng máy tính.
- Big-Endian
  - Bit LSB lưu ở ô nhớ có địa chỉ lớn nhất (ngoài cùng bên phải)
  - Bit MSB lưu ở ô nhớ có địa chỉ nhỏ nhất (ngoài cùng bên trái)
  - VD: 0x42 được biểu diễn: 0x00000042

36

Endianness

- Little-Endian
  - Bit LSB lưu ở ô nhớ có địa chỉ nhỏ nhất (ngoài cùng bên trái)
  - Bit MSB lưu ở ô nhớ có địa chỉ lớn nhất (ngoài cùng bên phải)
  - VD: 0x42 được biểu diễn: 0x42000000
- Dữ liệu mạng biểu diễn dạng Big-Endian
- Chương trình trên Windows x86 sử dụng kiểu Little-Endian

37

IP Address

IP: 127.0.0.1 chuyển sang dạng hex: 7F 00 00 01

- Gửi qua mạng dưới dạng: 0x7F000001
- Lưu trữ trong RAM dưới dạng: 0x0100007F

38

Operands

Toán tử trong một lệnh của hợp ngữ có thể là:

- Một giá trị trực tiếp: 0x42h
- Toán tử là một thanh ghi: EAX, EBX, ECX...
- Toán tử lấy địa chỉ: [EAX], [ECX + 10h]...

39

Registers

The x86 Registers

General registers	Segment registers	Status register	Instruction pointer
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

40

Registers

- Nhóm thanh ghi chung: Được CPU sử dụng như bộ nhớ siêu tốc trong việc tính toán, đặt biến tạm, tham số: EAX, EBX.
- Nhóm thanh ghi xử lý chuỗi: Sao chép, tính độ dài chuỗi: EDI.
- Thanh ghi ngăn xếp: Thanh ghi được dùng trong quản lý cấu trúc bộ nhớ ngăn xếp: ESP, EBP.

41

Registers

Thanh ghi đặc biệt

- EIP: Thanh ghi con trỏ lệnh, luôn trỏ đến lệnh tiếp theo mà CPU sẽ thực thi
- EFLAGS: Thanh ghi cờ, các bit cờ được bật hay không thể hiện trạng thái sau khi thực hiện một lệnh của CPU.

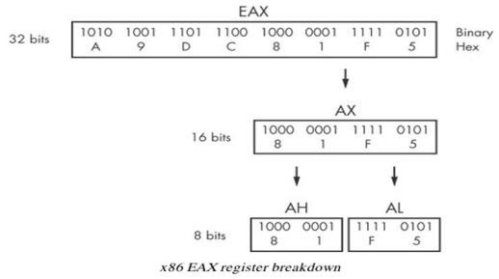
42

Kích thước Register

- Các thanh ghi trên kiến trúc x86 phổ biến có kích thước là 32 bits.
  - Các thanh ghi có thể tham chiếu dưới dạng 32 bit (VD: EAX) hoặc dưới dạng 16 bit (VD: AX).
- Bốn thanh ghi chung EAX, EBX, ECX, EDX cũng có thể tham chiếu dưới dạng các giá trị 8 bits:
  - AL: Biểu diễn giá trị 8 bit thấp
  - AH: Biểu diễn giá trị 8 bit cao

43

Kích thước Register



x86 EAX register breakdown

44

General Registers

- Thường lưu trữ dữ liệu hoặc địa chỉ bộ nhớ
- Một số quy định riêng về việc sử dụng các thanh ghi
  - Phép nhân và chia thường dùng thanh ghi EAX và EDX
  - Trình biên dịch sử dụng các thanh ghi một cách nhất quán (theo chuẩn)
  - Các giá trị trả về trong một hàm thường lưu vào thanh ghi EAX

45

Flags

- EFLAGS là một thanh ghi trạng thái
- Kích thước 32 bits
- Mỗi bit là một cờ (Chỉ có vài cờ: SF, ZF, AF, PF, CF..., chứ không phải có tận 32 cờ)
- Các cờ được bật (1) hoặc không được bật (0)

46

Flags

Cờ Zero – ZF (cờ không):

Cờ này được bật khi kết quả của phép toán là 0

Cờ Carry – CF (cờ nhớ):

Cờ này được bật khi có mượn hoặc nhớ bit MSB

Cờ Sign – SF (cờ dấu):

Cờ này được bật khi bit MSB của kết quả = 1 tức đây là một kết quả âm

Cờ Trap – TF (cờ bẫy):

Cờ này được bật để sử dụng chế độ gỡ lỗi, CPU sẽ chỉ thực hiện một lệnh tại một thời điểm

47

Flags

Cờ Overflow – OF (cờ tràn):

Cờ này được bật khi thực hiện phép tính với hai số cùng dấu mà kết quả là số có dấu => tràn

Cờ Parity – PF (cờ chẵn lẻ):

Cờ này được bật = 1 khi tổng số bit 1 trong kết quả là chẵn

Cờ này được bật = 0 khi tổng số bit 1 trong kết quả là lẻ

VD cờ tràn

EAX = 7fffffff (Giá trị số dương lớn nhất dạng 32 bit). Ta thực hiện lệnh: add eax, 1. Kết quả sau khi thực hiện: EAX = 80000000h, OF = 1

48

EIP (Extended Instruction Pointer)

- Thanh ghi chứa địa chỉ trên bộ nhớ của lệnh tiếp theo mà CPU sẽ thực thi
- Nếu EIP chứa dữ liệu sai, CPU sẽ tìm ra lệnh không hợp lệ và sẽ gây lỗi chương trình
- EIP là mục tiêu của lỗi tràn bộ đệm (Buffer Overflow)

49

Nhắc lại về Assembly

- Các mức trừu tượng của ngôn ngữ
- Reverse Engineering
- Kiến trúc x86
- Một số cấu trúc đơn giản

50

Một số cấu trúc đơn giản

- Cú pháp lệnh mov cơ bản:

  MOV Toán hạng đích, Toán hạng nguồn
  - Di chuyển dữ liệu từ toán hạng nguồn sang toán hạng đích
- Toán hạng được biểu diễn trong cặp dấu [toán hạng] hiểu đó là toán hạng lấy địa chỉ. VD:
  - mov giá trị: mov eax, ebx hoặc mov eax, 19h
  - mov địa chỉ: mov eax, [ebx] hoặc mov eax, [ebx+10]

51

Một số cấu trúc đơn giản

mov Instruction Examples

Instruction	Description
`mov eax, ebx`	Copies the contents of EBX into the EAX register
`mov eax, 0x42`	Copies the value 0x42 into the EAX register
`mov eax, [0x4037C4]`	Copies the 4 bytes at the memory location 0x4037C4 into the EAX register
`mov eax, [ebx]`	Copies the 4 bytes at the memory location specified by the EBX register into the EAX register
`mov eax, [ebx+esi*4]`	Copies the 4 bytes at the memory location specified by the result of the equation `ebx+esi*4` into the EAX register

52

LEA (Load Effective Address)

LEA EAX, [EBX+8]

- So sánh MOV EAX, [EBX+8] và LEA EAX, [EBX+8]
  - MOV: chuyển giá trị tại địa chỉ EBX + 8 vào EAX
  - LEA: Chuyển địa chỉ mà EBX đang giữ + 8 vào EAX

53

LEA (Load Effective Address)

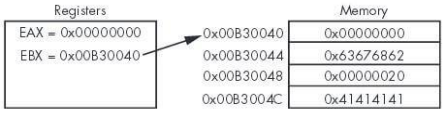
LEA EAX, [EBX+8]

- So sánh MOV EAX, [EBX+8] và LEA EAX, [EBX+8]
  - MOV: chuyển giá trị tại địa chỉ EBX + 8 vào EAX
  - LEA: Chuyển địa chỉ mà EBX đang giữ + 8 vào EAX

54

LEA (Load Effective Address)

❑ Các giá trị của thanh ghi EAX và EBX ở bên trái, dữ liệu được lưu trữ trong bộ nhớ ở bên phải. EBX được set giá trị **0xb30040**. Ở địa chỉ **0xb30048** là giá trị **0x20**. Lệnh **mov eax, [ebx+8]** sẽ truyền giá trị **0x20** (lấy từ bộ nhớ) vào thanh ghi EAX, còn lệnh **lea eax, [ebx+8]** sẽ set giá trị **0xb30048** trên thanh ghi EAX.



55

Các phép toán đại số

❑ **SUB: Subtracts**

❑ **ADD: Adds**

❑ **INC: Increments**

❑ **DEC: Decrements**

❑ **MUL: Multiplies**

❑ **DIV: Divides**

56

NOP

❑ **Lệnh đặc biệt, nó không làm gì cả**

❑ **Có mã opcode là 90**

❑ **Thường sử dụng NOP như để "trượt" qua**

57

So sánh

❑ **TEST**

- **So sánh hai toán hạng mà không làm thay đổi chúng**
- **TEST EAX, EAX => Kiểm tra EAX có bằng 0 hay không, nếu bằng 0 thì cờ ZF được bật**

❑ **CMP EAX, EBX**

Lệnh này thực hiện việc so sánh hai toán hạng bằng cách lấy toán hạng đích – toán hạng nguồn

- **Nếu đích = nguồn => CF = 0, ZF = 1, SF = 0**
- **Nếu đích > nguồn => CF = 0, ZF = 0, SF = 0**
- **Nếu đích < nguồn => CF = 1, ZF = 0, SF = 1**

58

Rẽ nhánh

❑ **JZ loc: Nhảy đến nhãn loc nếu cờ ZF được set (= 1)**

❑ **JNZ loc: Nhảy đến nhãn loc nếu cờ ZF không được set (= 0)**

59

The Stack

❑ **Vùng nhớ cho các biến cục bộ, tham số của hàm, theo dõi luồng điều khiển.**

❑ **Hoạt động theo nguyên lý "Vào sau ra trước - LIFO"**

❑ **Thanh ghi ESP (Extended Stack Pointer): Luôn trỏ vào đỉnh Stack**

❑ **Thanh ghi EBP (Extended Base Pointer): Thanh ghi con trỏ cơ sở, đáy Stack**

❑ **Lệnh PUSH: Đẩy dữ liệu vào ngăn xếp**

❑ **Lệnh POP: Lấy dữ liệu ra khỏi ngăn xếp**

60

Function Calls

- Việc tổ chức chương trình theo các hàm thuận tiện cho việc tái sử dụng trong một chương trình.
- Việc gọi hàm có hai cơ chế là Prologue và Epilogue.
- Prologue hay Epilogue: mô tả quá trình gọi hàm - cần chuẩn bị Stack như thế nào và các thanh ghi làm việc ra sao

61

Function Prologue

- Lưu thanh ghi con trỏ cơ sở (EBP) vào ngăn xếp. Sau khi thực hiện xong và quay trở lại hàm trước đó thì phải trả lại EBP ban đầu.
- Cập nhập giá trị mới cho EBP là bằng giá trị của ESP.
- Dịch chuyển ESP một khoảng phù hợp dành cho các biến cục bộ của hàm.

```
push ebp
mov ebp, esp
sub esp, N
```

62

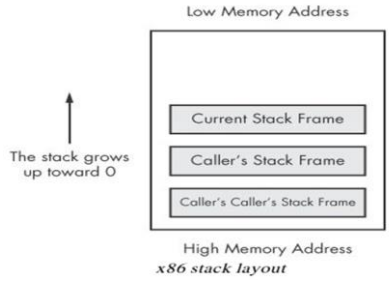
Function Epilogue

- Các lệnh ở cuối hàm sẽ khôi phục lại Stack và thanh ghi ở trạng thái trước khi hàm được gọi.

```
mov esp, ebp
pop ebp
ret
```

63

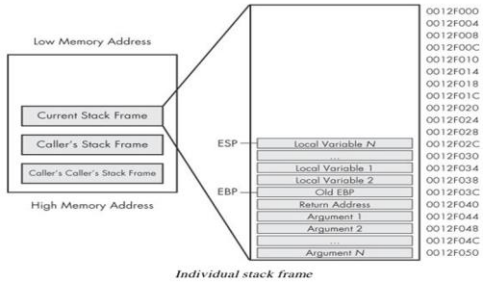
Function Calls



x86 stack layout

64

Function Calls



Individual stack frame

65

Function Calls

Danh sách sau tóm tắt luồng thực hiện của lời gọi hàm trong hầu hết trường hợp:

1. Các tham số đầu vào được đưa vào stack bằng lệnh push.
2. Một hàm được gọi bởi call memory_location. Địa chỉ lệnh hiện tại (giá trị của thanh ghi EIP) sẽ được push vào stack. Địa chỉ này sẽ được dùng để trở về chương trình chính khi hàm đã hoàn tất thực thi. Khi hàm bắt đầu, EIP được set giá trị memory_location (điểm bắt đầu của hàm).

66

Function Calls

3. Qua mở đầu hàm (prologue), không gian được chỉ định trên stack cho các biến cục bộ và EBP được push vào stack.
4. Hàm bắt đầu thực hiện chức năng của mình.
5. Qua kết thúc hàm (epilogue), stack được khôi phục về trạng thái ban đầu. ESP được điều chỉnh giá trị để giải phóng các biến cục bộ. EBP được khôi phục và hàm gọi đến có thể đánh đúng địa chỉ cho các biến của nó. Lệnh leave có thể được dùng như một kết thúc hàm vì nó set ESP bằng EBP và pop EBP khỏi stack.

67

Function Calls

6. Hàm trả về giá trị bằng lệnh ret. Lệnh này pop địa chỉ trả về khỏi stack và truyền giá trị đó vào EIP, chương trình sẽ tiếp tục thực thi từ đoạn lời gọi ban đầu được gọi.
7. Stack được điều chỉnh để xóa các tham số đã được gửi đi trừ khi các tham số đó sẽ được sử dụng lại về sau.

68

Nội dung

1. Nhắc lại về Assembly
2. Sử dụng IDA pro để dịch ngược mã độc
3. Sử dụng đối sánh chéo
4. Phân tích hàm
5. Sử dụng biểu đồ hàm
6. Một số lưu ý

69

Sử dụng IDA pro để dịch ngược mã độc

- ❑ IDA Pro
- ❑ Một số tính năng hữu ích

70

Sử dụng IDA pro để dịch ngược mã độc

- ❑ IDA Pro
- ❑ Một số tính năng hữu ích

71

IDA pro

72

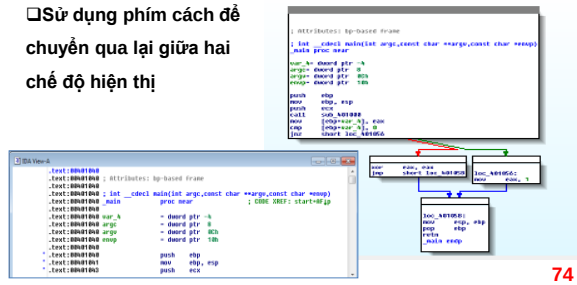
IDA Pro Versions

- Phiên bản thương mại sẽ đầy đủ tính năng hơn
- Phiên bản miễn phí thì hạn chế tính năng hơn
  - Cả hai phiên bản sẽ đều hỗ trợ kiến trúc x86
  - Với phiên bản thương mại thì hỗ trợ kiến trúc x64 và nhiều bộ vi xử lý khác nữa, ví dụ như với các dòng vi xử lý trên thiết bị di động.
- Cả hai phiên bản đều có những mẫu nhận dạng thư viện và những công nghệ nhận dạng giúp cho quá trình Disassembly

73

Graph and Text Mode

- Sử dụng phím cách để chuyển qua lại giữa hai chế độ hiện thị



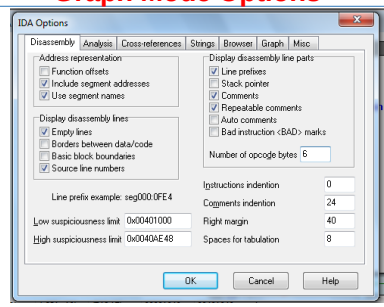
74

Default Graph Mode Display

```
; Attributes: bp-based frame

; int __cdecl main(int argc,const char **argv,const char *envp)
_main proc near

var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
push    ecx
call    sub_401000
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0
jnz     short loc_401056
```

75

Graph Mode Options



76

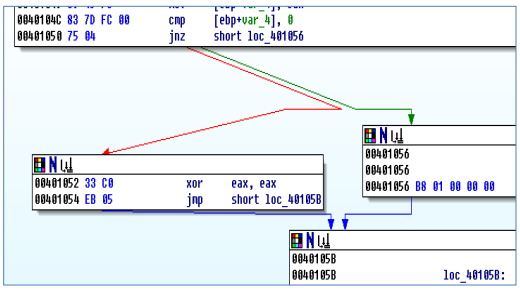
Arrows

Các kiểu mũi tên trong chế độ Graph mode:

- Màu
  - Đỏ (Red): Nhảy có điều kiện nhưng chưa thực hiện,
  - Xanh lá cây (Green): Nhảy có điều kiện và được thực hiện,
  - Xanh dương (Blue): Nhảy không điều kiện.
- Hướng
  - Lên: Biểu diễn đây là một vòng lặp.

77

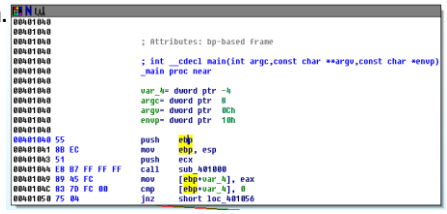
Arrow Color Example



78

Highlighting

❑ Đánh dấu tên một biến, thanh ghi... Trong chế độ Graph mode làm nổi bật những biến, thanh ghi giống nhau.



79

Text Mode

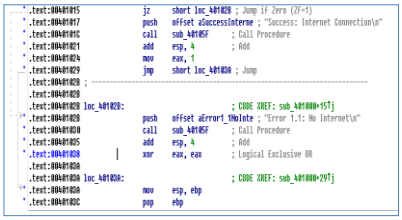
Mũi tên nét liền = Nhảy không có điều kiện
Mũi tên nét đứt = Nhảy có điều kiện
Mũi tên đi lên = Vòng lặp

Những comment được sinh ra bởi IDA Pro

Section
Address

80

Thêm nhận xét cho mỗi lệnh



81

Sử dụng IDA pro để dịch ngược mã độc

❑ IDA Pro

❑ Một số tính năng hữu ích

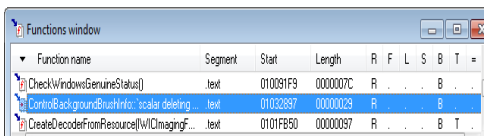
82

Functions

❑ Cửa sổ này hiển thị các hàm, độ dài và các cờ

L = Library function

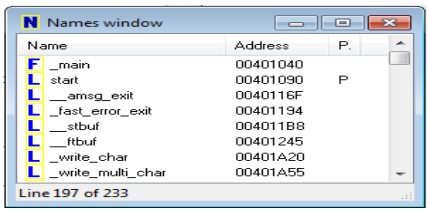
❑ Windows > Function để hiển thị cửa sổ này



83

Names Window

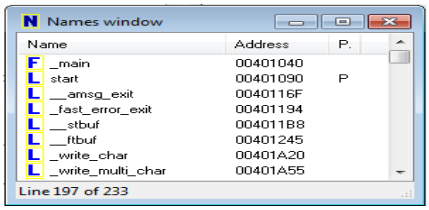
❑ Cửa sổ này hiển thị tên các hàm, dữ liệu, chuỗi đi kèm với địa chỉ của chúng.



84

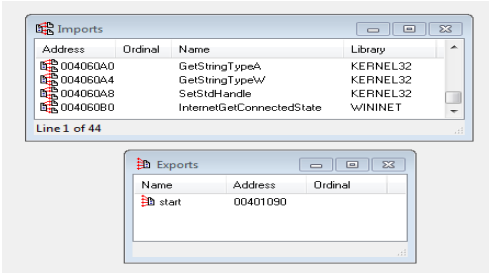
Strings

❑ Cửa sổ này liệt kê các chuỗi trong đoạn .rdata của binary.



85

Imports & Exports

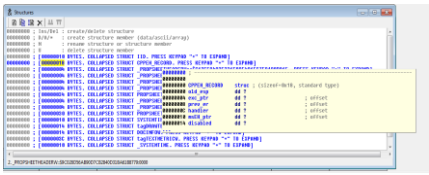


86

Structures

❑ Hiển thị tất cả dữ liệu cấu trúc

❑ Di chuột vào để hiển thị một Pop-up màu vàng

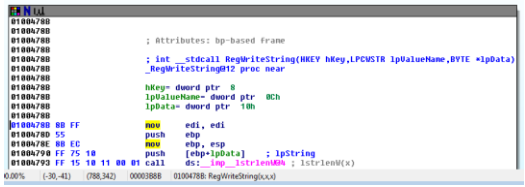


87

Function Call

❑ Các tham số được đẩy vào Stack

❑ Lệnh Call hiểu là sẽ gọi một thủ tục/hàm



88

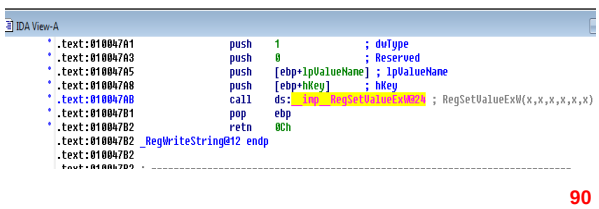
Imports or Strings

❑ Nhấn đúp chuột vào bất kỳ entry nào để hiển thị cửa sổ Disassembly.

89

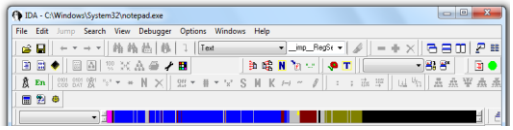
Using Links

❑ Nhấn đúp chuột vào bất kỳ địa chỉ nào trong cửa sổ Disassembly để hiển thị vị trí đó.



90

Dải điều hướng



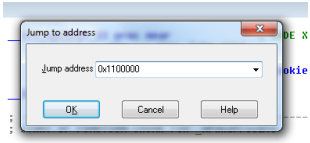
Thanh điều hướng hiển thị dải màu này rất hữu dụng, nó cho chúng ta biết được nên phân tích vùng nào:

- Light Blue (Xanh nhạt): Vùng code của thư viện
- Red (đỏ): Vùng code mà trình biên dịch sinh ra
- Dark Blue (Xanh đậm): Vùng code của người dùng viết, đây là phần cần phân tích

91

Nhảy tới địa chỉ

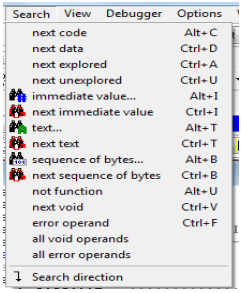
- Nhấn phím G và điền địa chỉ muốn nhảy đến vào trong khung nhập địa chỉ
- Nhấn Ok để chương trình nhảy đến vị trí đã điền



92

Tìm kiếm

- Tìm kiếm tên hàm
- Tìm kiếm tên biến
- Tìm kiếm địa chỉ
- Tìm kiếm comment
- .vv.



93

Nội dung

1. Nhắc lại về Assembly
2. Sử dụng IDA pro để dịch ngược mã độc
3. Sử dụng đối sánh chéo
4. Phân tích hàm
5. Sử dụng biểu đồ hàm
6. Một số lưu ý

94

Nội dung

1. Nhắc lại về Assembly
2. Sử dụng IDA pro để dịch ngược mã độc
3. Sử dụng đối sánh chéo
4. Phân tích hàm
5. Sử dụng biểu đồ hàm
6. Một số lưu ý

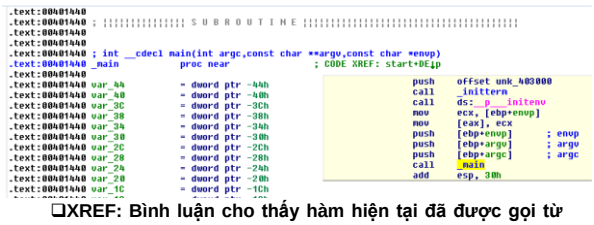
95

Sử dụng đối sánh chéo

- Code Cross-References
- Data Cross-References

96

Code Cross-References

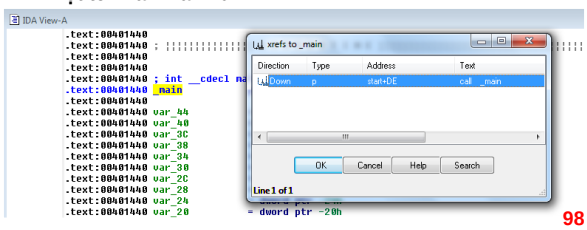


- XREF: Bình luận cho thấy hàm hiện tại đã được gọi từ đâu
- Chỉ hiện thị một vài tham chiếu chéo mặc định

97

Code Cross-References

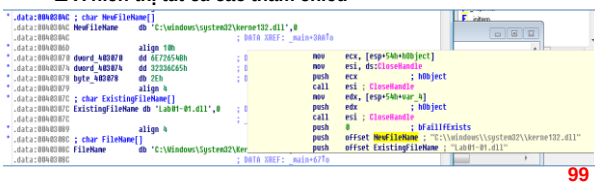
- Để xem tất cả Code Cross-References: Click vào một tên hàm và nhấn 'X'



98

Data Cross-References

- Bắt đầu với chuỗi, nháy đúp chuột vào chuỗi
- Di chuột qua DATA XREF để xem chuỗi đó ở đâu sử dụng
- X hiển thị tất cả các tham chiếu



99

Nội dung

1. Nhắc lại về Assembly
2. Sử dụng IDA pro để dịch ngược mã độc
3. Sử dụng đối sánh chéo
4. Phân tích hàm
5. Sử dụng biểu đồ hàm
6. Một số lưu ý

10(

Phân tích hàm

- Hàm và tham số
- Biến toàn cục và biến cục bộ
- Các phép toán cơ bản

10:

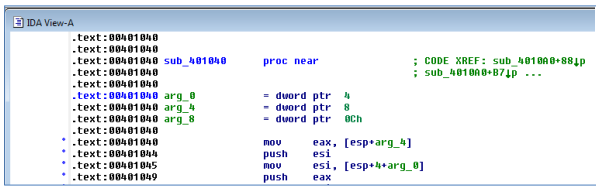
Phân tích hàm

- Hàm và tham số
- Biến toàn cục và biến cục bộ
- Các phép toán cơ bản

10:

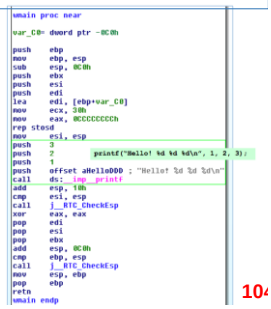
Hàm và tham số

- IDA xác định các hàm, các tham số và đặt tên chúng
- Không phải luôn luôn đúng



Disassembly in IDA Pro

- Hàm printf() sẽ được xử lý theo thứ: tham số thứ n xử lý trước rồi đến n-1, n-2...



Phân tích hàm

- Hàm và tham số
- Biến toàn cục và biến cục bộ
- Các phép toán cơ bản

Biến toàn cục và biến cục bộ

Biến toàn cục (Global Variable)

- Được định nghĩa bên ngoài hàm, phạm vi toàn chương trình, sử dụng được ở tất cả các hàm.

Biến cục bộ (Local Variable)

- Được định nghĩa trong một hàm và phạm vi nằm trong hàm/khối lệnh đó.

Biến toàn cục và biến cục bộ

```
#include "stdafx.h"

int i=1; // GLOBAL VARIABLE

int _tmain(int argc, _TCHAR* argv[])
{
    int j=2; // LOCAL VARIABLE
    printf("YOURNAME-8a: %d %d\n", i, j);
    return 0;
}
```

```
C:\Windows\system32\cmd.exe
YOURNAME-8a: 1 2
Press any key to continue . . .
```


Biến toàn cục và biến cục bộ

```
#include "stdafx.h"

int i=1; // GLOBAL VARIABLE

int _tmain(int argc, _TCHAR* argv[])
{
    00401003    mov    eax, dword_40CF60
    00401008    add    eax, dword_40C000
    0040100E    mov    dword_40CF60, eax
    00401013    mov    ecx, dword_40CF60
}
```

Biến toàn cục và biến cục bộ

❑ mov [ebp+var_8], 2 tương ứng với biến j là biến cục bộ

```
#include "stdafx.h"

int i=1; // GLOBAL VARIABLE

int _tmain(int argc, _TCHAR* argv[])
{
    int j=2; // LOCAL VARIABLE
    printf("YOURNAME-8a: %d %d\n", i, j);
    return 0;
}
```

```
mov     [ebp+var_8], 2
mov     esi, esp
mov     eax, [ebp+var_8]
push    eax
mov     ecx, i
push    ecx
push    offset aYourname8aDD ; "YOURNAME-8a: %d %d\n"
call    ds:__imp__printf
```

109

Phân tích hàm

❑ Hàm và tham số

❑ Biến toàn cục và biến cục bộ

❑ Các phép toán cơ bản

110

Các phép toán cơ bản

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
int i=10;
int j=2;
int k;
i = i + 2;
k = i / j;
printf("YOURNAME-9a: %d %d %d\n", i, j, k);
return 0;
}
```

```
C:\Windows\system32\cmd.exe
YOURNAME-9a: 12 2 6
Press any key to continue . . .
```

111

Các phép toán cơ bản

```
mov     [ebp+var_8], 0Ah
mov     [ebp+var_14], 2
mov     eax, [ebp+var_8]
add     eax, 2
mov     [ebp+var_8], eax
mov     eax, [ebp+var_8]
cdq
idiv    [ebp+var_14]
mov     [ebp+var_20], eax
```

```
int i=10;
int j=2;

i = i + 2;


k = i / j;
```

112

Các phép toán cơ bản

ASM Code	Explanation	C Code
mov [ebp+var_8], 0Ah	Put the number 10 into a local variable (i)	int i=10;
mov [ebp+var_14], 2	Put the number 2 into a local variable (j)	int j=2;
mov eax, [ebp+var_8]	Put i into eax	
add eax, 2	Add 2 to eax	i = i + 2;
mov [ebp+var_8], eax	Put eax (the result) into a local variable (i)	
mov eax, [ebp+var_8]	Put i into eax	
cdq	Convert double to quad (required for division)	k = i / j;
idiv [ebp+var_14]	Divide the value in eax by a local variable (j)	
mov [ebp+var_20], eax	Put eax (the result) into a local variable (k)	

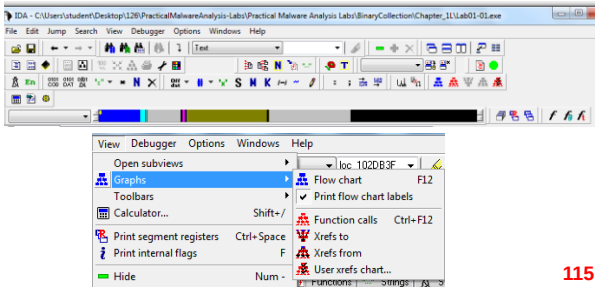
113

Nội dung

1. Nhắc lại về Assembly
2. Sử dụng IDA pro để dịch ngược mã độc
3. Sử dụng đối sánh chéo
4. Phân tích hàm
5. Sử dụng biểu đồ hàm
6. Một số lưu ý

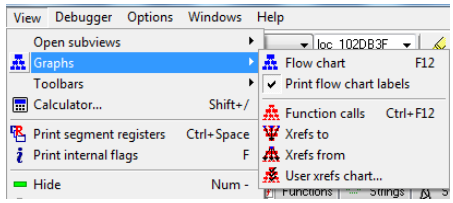
114

Sử dụng biểu đồ hàm



115

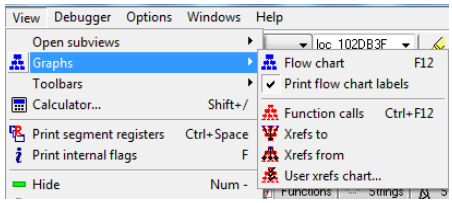
Sử dụng biểu đồ hàm



- Flow chart: Tạo biểu đồ theo dõi của hàm hiện tại
- Function call: Đồ thị hàm gọi toàn bộ chương trình

116

Sử dụng biểu đồ hàm

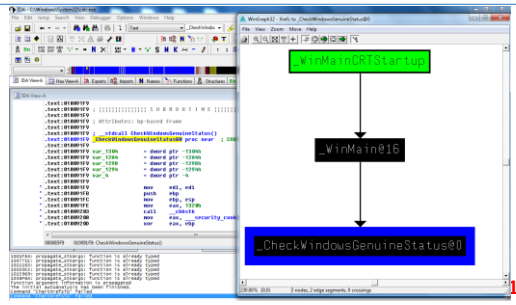


Xrefs to: Đồ thị XREFs đến XREF đã chọn

Có thể hiển thị tất cả các paths đến một hàm

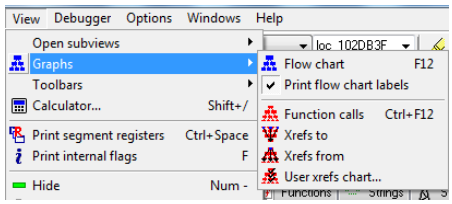
117

Xrefs to



118

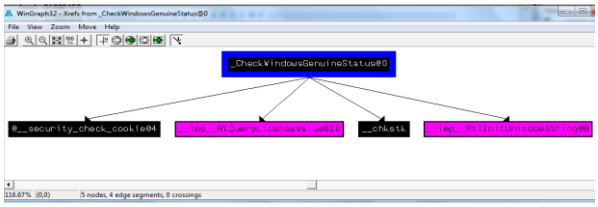
Sử dụng biểu đồ hàm



- Xrefs from: Đồ thị XREFs từ XREF đã chọn
- Có thể hiển thị tất cả các đường đi thoát khỏi một hàm

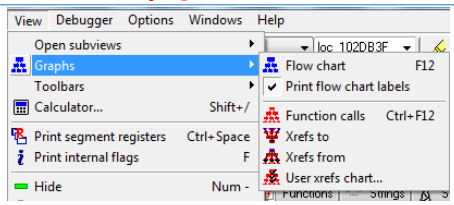
119

Xrefs from



120

Sử dụng biểu đồ hàm



User xrefs chart:

- **Tùy chỉnh đệ quy của biểu đồ, ký hiệu được sử dụng...**
- **Cách duy nhất để sửa đổi các đồ thị kế thừa**

Nội dung

1. **Nhắc lại về Assembly**
2. **Sử dụng IDA pro để dịch ngược mã độc**
3. **Sử dụng đối sánh chéo**
4. **Phân tích hàm**
5. **Sử dụng biểu đồ hàm**
6. **Một số lưu ý**

Một số lưu ý

Một số tùy chọn sau đây giúp việc phân tích dễ dàng hơn:

- ❑ **Đổi tên hàm**
- ❑ **Comments**
- ❑ **Kiểu hiển thị các toán tử**
- ❑ **Sử dụng hằng số được đặt tên**

Đổi tên hàm

- ❑ **Có thể thay đổi một tên như: sub_401000 thành ReverseBackdoorThread**
- ❑ **Khi thay đổi tên ở một nơi thì IDA sẽ tự động đồng bộ tên mới ở tất cả những nơi khác.**

Đổi tên hàm

Function Operand Manipulation

Without renamed arguments

```
004013C8  mov   eax, [ebp+arg_4]
004013CB  push  eax
004013CC  call  _atoi
004013D1  add   esp, 4
004013D4  mov [ebp+var_598], ax
004013DB  movzx ecx, [ebp+var_598]
004013E2  test  ecx, ecx
004013E4  jnz   short loc_4013F8
004013E6  push  offset aError
004013EB  call  printf
004013F0  add   esp, 4
004013F3  jmp   loc_4016FB
004013F8 ; ---------------------
004013F8
004013F8 loc_4013F8:
004013F8  movzx edx, [ebp+var_598]
004013FF  push  edx
00401400  call  ds:htons
```

With renamed arguments

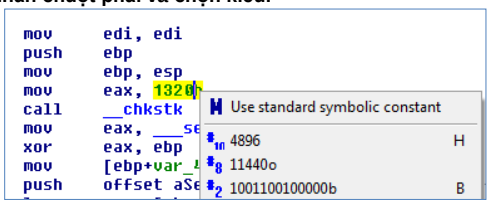
```
004013C8  mov   eax, [ebp+port_str]
004013CB  push  eax
004013CC  call  _atoi
004013D1  add   esp, 4
004013D4  mov   [ebp+port], ax
004013DB  movzx ecx, [ebp+port]
004013E2  test  ecx, ecx
004013E4  jnz   short loc_4013F8
004013E6  push  offset aError
004013EB  call  printf
004013F0  add   esp, 4
004013F3  jmp   loc_4016FB
004013F8 ; ---------------------
004013F8
004013F8 loc_4013F8:
004013F8  movzx edx, [ebp+port]
004013FF  push  edx
00401400  call  ds:htons
```


Comments

- ❑ **Nhấp dấu hai chấm (:) để thêm một comment**
- ❑ **Comment được đặt sau dấu chấm phẩy (;) cho tất cả các Xrefs**

Kiểu hiển thị các toán tử

- Các toán tử mặc định kiểu thập lục phân (Hex)
- Có thể sử dụng định dạng kiểu khác bằng cách nhấn chuột phải và chọn kiểu.



127

Sử dụng hằng số được đặt tên

- Làm cho các tham số của Windows API được rõ ràng hơn

Before symbolic constants

```
mov     esi, [esp+1Ch+argv]
mov     edx, [esi+4]
mov     edi, ds:CreateFileA
push    0    ; hTemplateFile
push    80h  ;
dwFlagsAndAttributes
push    3    ;
dwCreationDisposition
push    0    ;
lpSecurityAttributes
push    1    ; dwShareMode
```

After symbolic constants

```
mov     esi, [esp+1Ch+argv]
mov     edx, [esi+4]
mov     edi, ds:CreateFileA
push    NULL  ; hTemplateFile
push    FILE_ATTRIBUTE_NORMAL ;
dwFlagsAndAttributes
push    OPEN_EXISTING         ;
dwCreationDisposition
push    NULL                  ;
lpSecurityAttributes
push    FILE_SHARE_READ       ; dwShareMode
```

128

Nội dung

1. Nhắc lại về Assembly
2. Sử dụng IDA pro để dịch ngược mã độc
3. Sử dụng đối sánh chéo
4. Phân tích hàm
5. Sử dụng biểu đồ hàm
6. Một số lưu ý

129