Reverse Engineering Tutorial

Anti-Cracking Techniques

MỤC LỤC

I. Giới thiệu

Đây là một bài viết giới thiệu các cách tiếp cận công nghệ đảo mã (Reverse Code Engineering - RCE). Thêm vào đó, là những chỉ dẫn làm thế nào để bảo vệ phần mềm khỏi việc truy ra ra các thông tin nhạy cảm, như là việc "mò" ra serial của chương trình, hay là "dịch ngược" mã nguồn ở dạng mã máy assembly để biết được thuật toán phát sinh key. Bài viết này không có tham vọng làm thay đổi những suy nghĩ không đúng về RCE, mà chỉ muốn chứng minh cho mọi người thấy rằng RCE có thể tạo ra một thế giới an toàn hơn.

II. Công cụ và target

1. Công cụ

Đa số các công cụ phục vụ cho việc nghiên cứu RCE đều có thể tìm trên NET, một trong số chúng là FREE, số khác ở dạng shareware. Sau đây là một số tool phục vụ cho việc dịch ngược mã nguồn và debug chương trình:

- **OllyDBG** http://www.ollydbg.de/
- **IDA Pro Disassembler and Debugger** http://www.hex-rays.com/
- **W32Dasm** http://www.google.com
- **WinDbg** http://www.microsoft.com/whdc/devtools/debugging/default.mspx

Ngoài ra, bạn cần sử dụng một số tool khác như:

- PROTECTiON iD http://pid.gamecopyworld.com/, được sử dụng để kiểm tra một file thực thi (executables file) có được bảo vệ bởi một packer/protector/encryptor và nhận dạng ra trình biên dịch chương trình. Bạn có thể tham khảo tại http://en.wikipedia.org/wiki/Executable_compression để biết thêm chi tiết
- Import REConstructor http://www.google.com/, được dùng để fix IAT của file thực thi sau khi đã được unpack.
- System Internals http://technet.microsoft.com/en-us/sysinternals/default.aspx. Các chương trình như là FileMon, RegMon dùng để giám sát cách hoạt động của một chương trình. Một kỹ thuật mới cần phải tìm hiểu ở đây là "**SandBox**" – nó cung cấp thông tin của tất cả các chương trình đang hoạt động.

2. Target

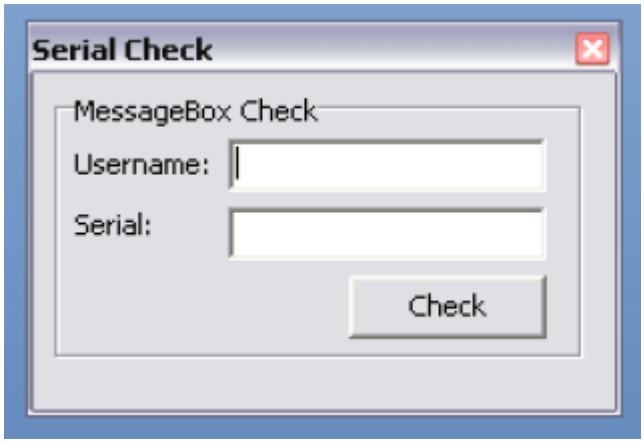
- **Tên chương trình:** Example.v1.0.exe (Serial Check)
- **MD5 CheckSum:** 4c78179f07c33e0f9ec8f2b0509bd069
- **Trình biên dịch:** Borland Delphi

Chúng ta bắt đầu xem xét phương pháp tiếp cận RCE đầu tiên. Công cụ được sử dụng trong phần này là OllyDBG dùng bản gốc down trên homepage hay bản đã được modify.

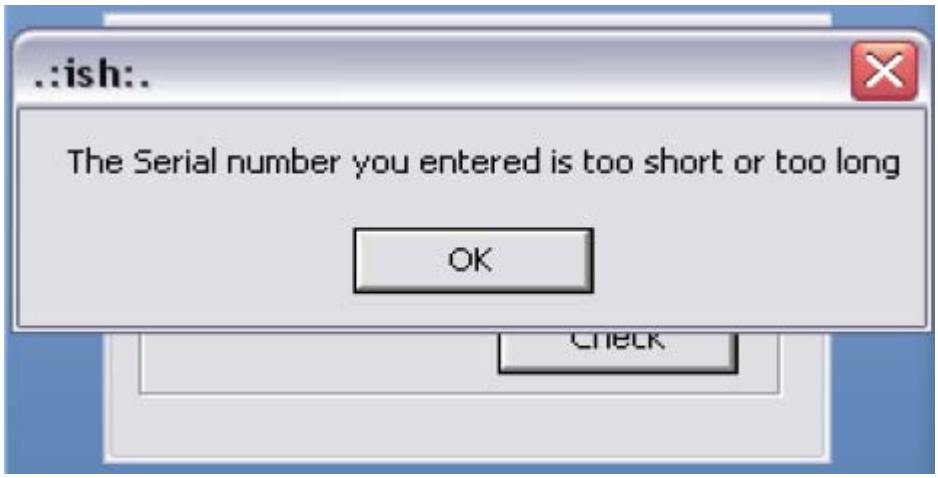
III. Phân tích chương trình

Trước khi bắt đầu, chúng ta cần phải phân tích chức năng của một chương trình cụ thể (trong ví dụ này đó chính là Serial Check) và tốt hơn thì nên hiểu nó hoạt động như thế nào?

Như bạn thấy, giao diện chương trình rất đơn giản. Chức năng chính là kiểm tra Username và serial nhập vào có ***hợp lệ*** hay không? Bước đầu tiên, là bạn phải điền dữ liệu vào 2 khung textbox, và nhấn nút Check và quan sát xem chương trình có phản hồi như thế nào (có thể là hiển thị một messagebox thông báo dữ liệu hợp lệ hay không, hay dữ liệu của bạn chưa thỏa một điều kiện nào đó, ví dụ Username tối thiểu dài hơn 6 ký tự, trong khibạn nhập có 5 ký tự, ...)



Đôi khi kết quả của việc phản hồi từ chương trình, có thể cung cấp cho chúng ta một số gợi ý trước khi bắt tay vào việc tìm ra thuật toán kiểm tra serial và cũng hy vọng rằng thuật toán check serial cũng không quá phức tạp, chỉ là thuật toán kiểm tra chiều dài serial phải nằm trong giới hạn lập trình viên quy định.

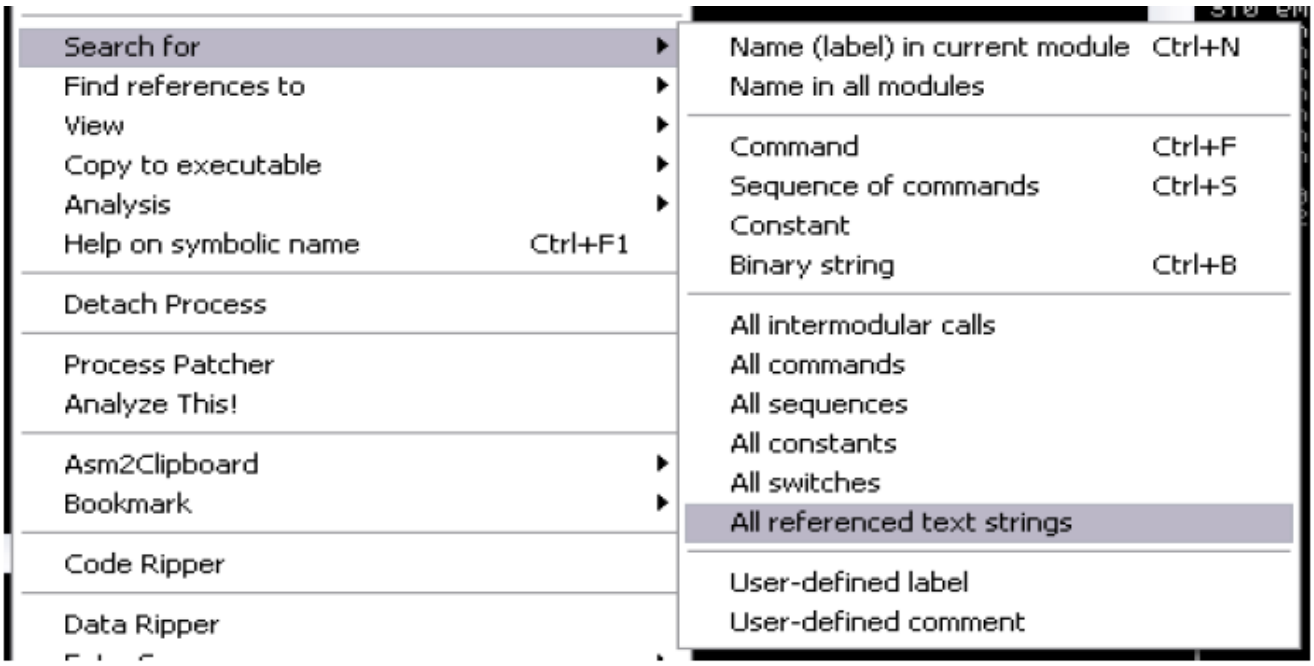


Kế tiếp chúng ta sẽ chuyển sang giai đoạn là dịch ngược và debug chương trình để có được nhiều hơn những thông tin mà chúng ta cần về cách chương trình làm việc.

1. Cách tiếp cận thứ 1

a) Tìm chuỗi Badboy

– Click chuột phải tại 1 vị trí bất kỳ trong cửa sổ disassemler, chọn *Search for* → *All referenced text strings*



– Như bạn thấy, chúng ta dễ dàng tìm thấy chuỗi Badboy từ messagebox phản hồi của chương trình. Bằng cách double click vào chuỗi này, bạn sẽ được OllyDBG đưa đến đoạn code xử lý việc hiển thị messagebox ở trên:

```
00456F96 PUSH Example_.00456FA8      ASCII ".:ish:."
00456F9B PUSH Example_.00456FB0      ASCII "The Serial number you entered is too short or too long"
00456FA8 ASCII ".:ish:.",0
00456FB0 ASCII "The Serial numbe"
00456FC0 ASCII "r you entered is"
00456FD0 ASCII " too short or to"
00456FE0 ASCII "o long",0
00456FEA PUSH Example_.00456FFC      ASCII ".:ish:."
00456FEF PUSH Example_.00457004      ASCII "The Serial number you entered is not valid"
00456FFC ASCII ".:ish:.",0
00457004 ASCII "The Serial numbe"
00457014 ASCII "r you entered is"
00457024 ASCII " not valid",0
00457032 PUSH Example_.00457044      ASCII ".:ish:."
00457037 PUSH Example_.0045704C      ASCII "Thank You for registering."
```

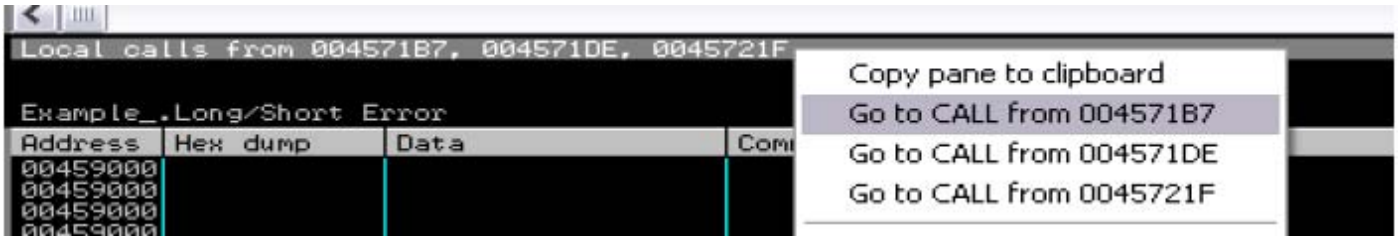
– Mặc dù chương trình Serial Check này được code với độ khó vừa phải, với một người làm việc trong lĩnh vực RCE có một ít kinh nghiệm có thể trace nơi mà hàm check serial được gọi là patch chương trình.

– Bây giờ, chúng ta sẽ set một nhãn tại địa chỉ bắt đầu hàm này (để tiện việc tra cứu sau này) bằng cách click phải → Label

và set một label có tên là Long/Short Error

– Bạn thấy hàm này được gọi ở 3 địa chỉ khác nhau, nếu nhìn kỹ bạn sẽ thấy 3 địa chỉ này gần nhau trong bộ nhớ

– Chúng ta sẽ nhảy tới địa chỉ đầu tiên gọi hàm (***004571B7***) bằng cách click chuột phải → Go to CALL from ***004571B7***



– Bạn đã đến đúng nơi cần đến, đây là đoạn code check serial có hợp lệ hay không?

b) Giải pháp đề nghị

Để tránh việc truy ra các thông tin nhạy cảm của chương trình bằng cách trace các hàm check serial từ việc search chuỗi badboy, một lập trình viên nên làm như sau:

– Lưu trữ chuỗi badboy trong 1 biến toàn cục hoặc tốt hơn là bên trong các mảng và tham chiếu đến nó khi cần thiết.

Mã giả

```
array[] myMsges = {'The Serial number you entered is too short or too long',
                   'The Serial number you entered is not valid',
                   'Thank You for registering.'}
 //Code omitted
 function registrationCheck():
 if(invalid_length) then
    sendMessage(myMsges[0])

 if(invalid_serial) then
    sendMessage(myMsges[1])

 if(valid_serial) then
    sendMessage(myMsges[2])
```

```
//This can be done separately.
//Let's assume that the result of this code will be: 'dkg$2 kF2
gkfoaplk'

string thank_you = 'Thank You for registering'

for(each letter in thank_you) do
add_5_to_ascii_value(letter)
print thank_you


//program serial check
If(valid_serial) then
sendMessage(decrypt('dkg$2 kF2 gkfoaplk'))
```

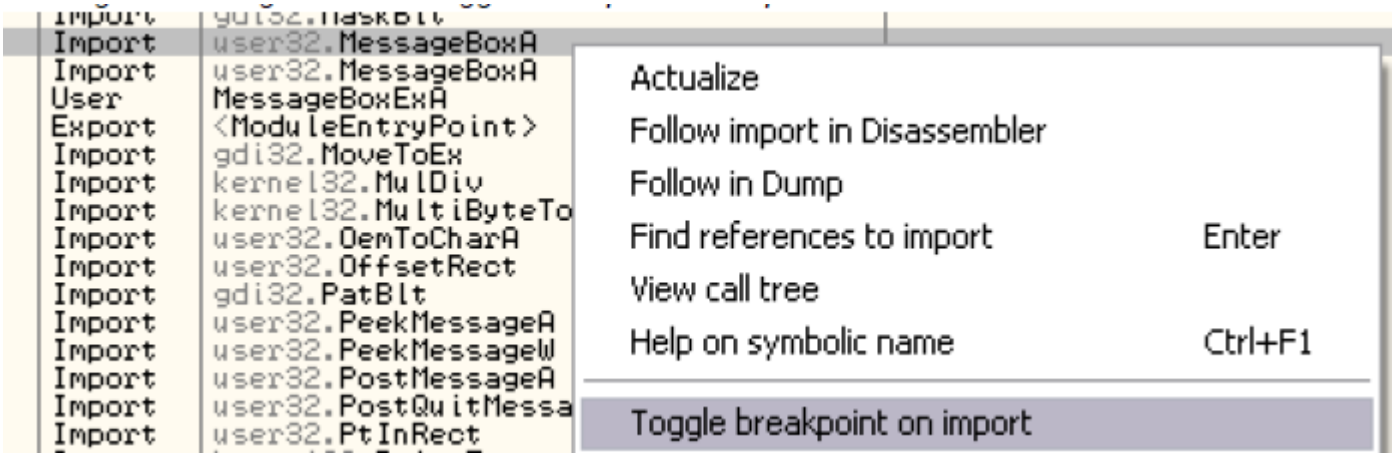
2. Cách tiếp cận thứ 2

a) Đặt breakpoint tại các hàm API

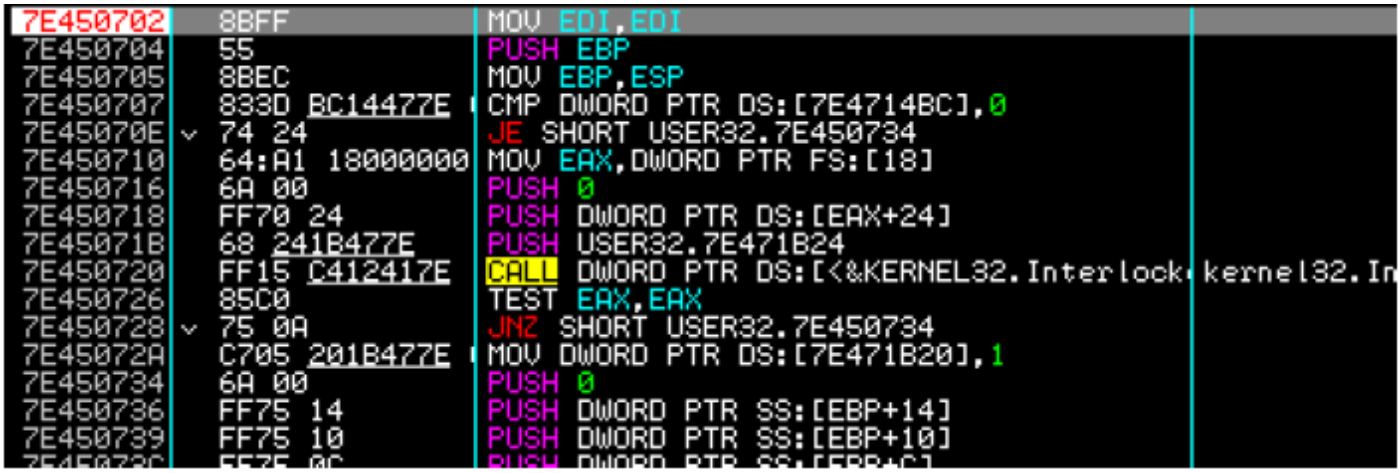
Ở cách tiếp cận thứ 2 này, chúng ta sẽ sử dụng Breakpoint (điểm dừng) tại hàm API MessageBoxA. Một vài chương trình sử dụng hàm **MessageBoxW**, **MessageBoxExA** hoặc **MessageBoxExW.**

– Sử dụng plug-in Command bar của OllyDBG, gõ vào lệnh bp **MessageBoxA** và nhấn Enter:

– Nhấn ALT + E để chuyển qua cửa sổ Executable Modules, chọn file executable (Example.v1.0.exe) và nhấn CTRL + N, tìm hàm MessageBoxA → click phải chọn ***Toggle breakpoint on import***

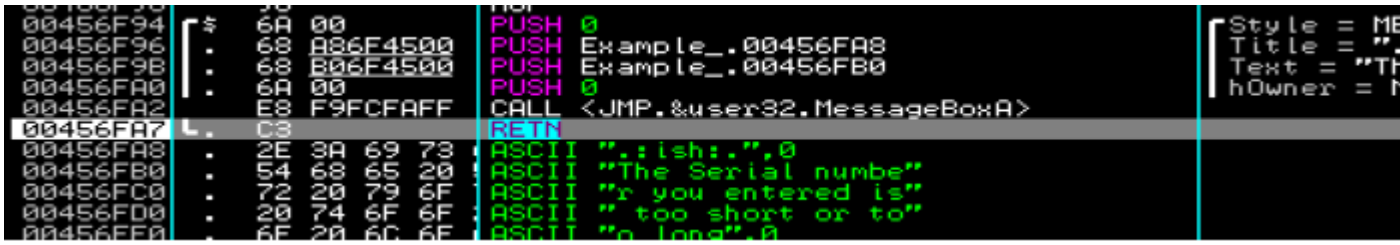


– Run chương trình, nhập dữ liệu sau đó nhấn nút Check, bạn sẽ dừng lại tại MessageBoxA bên trong user32.dll



– Nhấn CTRL + F9 hoặc F8 cho đến khi đến cuối hàm, bạn dừng tại đây:

– Nhấn F8 lần nữa để ra ngoài hàm MessageBoxA này



Bạn thấy chúng ta dừng tại cùng 1 địa chỉ như cách tiếp cận thứ 1 ở bước thứ 3

b) Giải pháp đề nghị

Để tránh việc trace chương trình thông qua cách đặt Breakpoint tại các hàm API , lập trình viên nên giới hạn việc sử dụng các hàm API này, hãy làm cách nào đó để giảm thiểu số hàm API bạn sử dụng trong chương trình, thí dụ để hiển thị một message box, bạn có thể viết 1 hàm riêng, không dùng MessageBoxA

3. Cách tiếp cận thứ 3

a) Trace Stack (thường được gọi là phương pháp reverse dùng Stack)

Đôi khi bạn đã thử qua 2 cách tiếp cận trên, vì kinh nghiệm coding của lập trình viên ngày càng nhiều, nên họ có thể áp dụng các trick đã nêu ở trên giấu đi chuỗi badboy trong stack. Do đó, bạn không thể tìm chuỗi bằng cách thông thường, mà bạn phải tìm nó thông qua phương pháp này – tracing stack. Khi một chỉ thị "CALL <thủ tục>" được thực thi bởi CPU, địa chỉ của chỉ thị tiếp theo sắp được gọi để thực thi sẽ được chứa trong con trỏ EIP và được đẩy (push) vào Stack. Khi một thủ tục đã kết thúc, chỉ thị RETN được gọi thì processor sẽ pop các giá trị mà nó đã push ban nảy ra khỏi stack và trở về địa chỉ của chỉ thị kế sau thủ tục đã được gọi.

Giả sử:

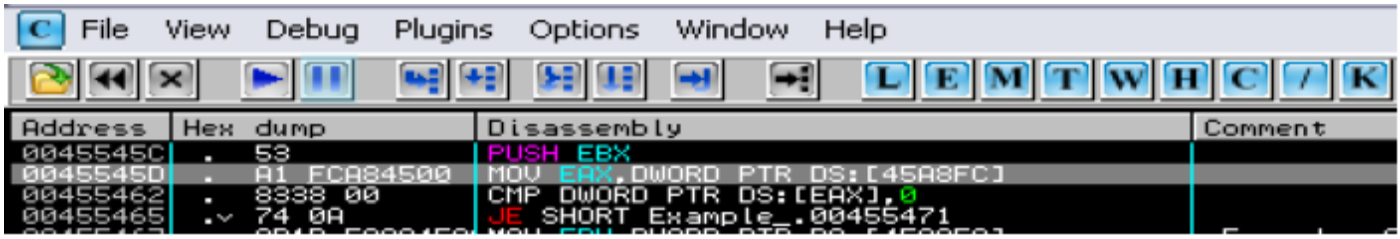
Offset	Opcode
1	PUSH 0
2	CALL 0xF
3	TEST EAX,EAX

Khi CALL 0xF được thực thi, giá trị của offset 3 sẽ được push vào stack

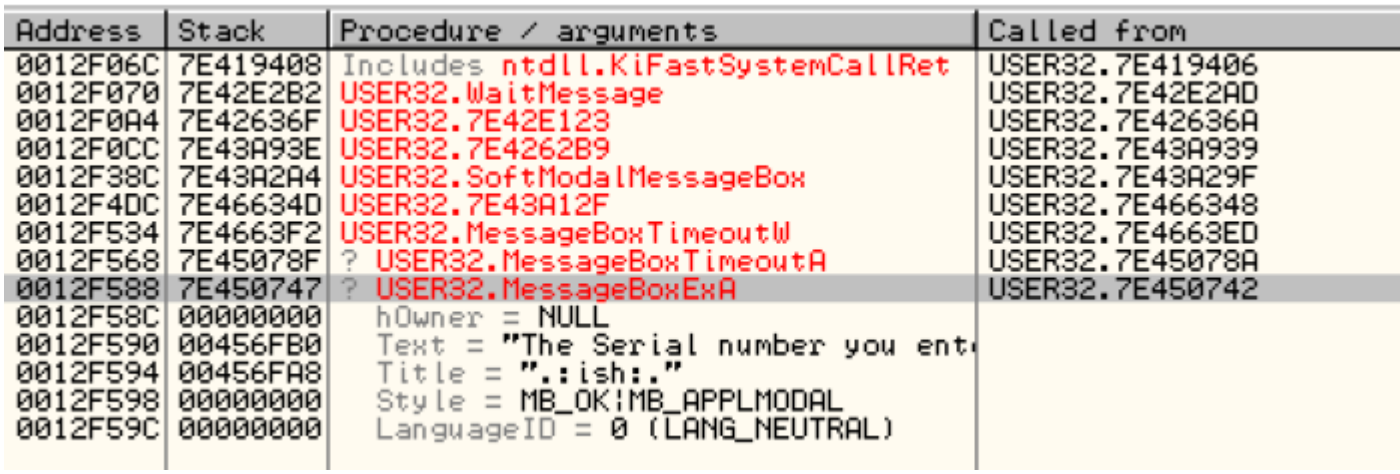
Offset	Opcode
F	MOV EAX,1
10	RETN
11	NOP

Khi RETN được thực thi, giá trị của offset 3 sẽ được pop ra khỏi stack và đặt vào EIP

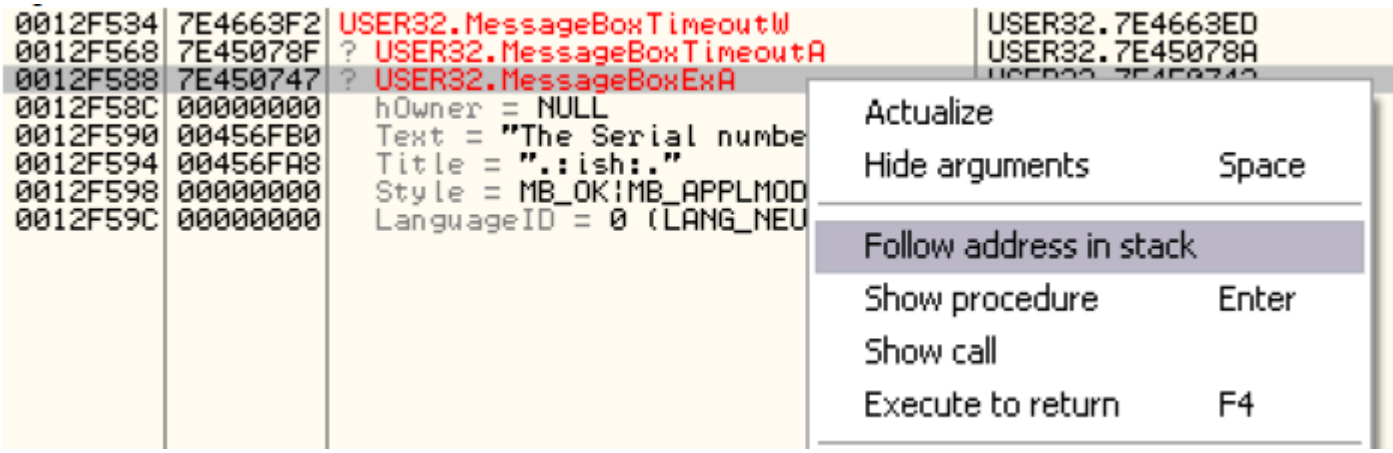
– Run chương trình, nhập dữ liệu sau đó nhấn nút Check, nhấn tiếp nút Pause trên thanh toolbar của OllyDBG, bạn dừng tại đây:



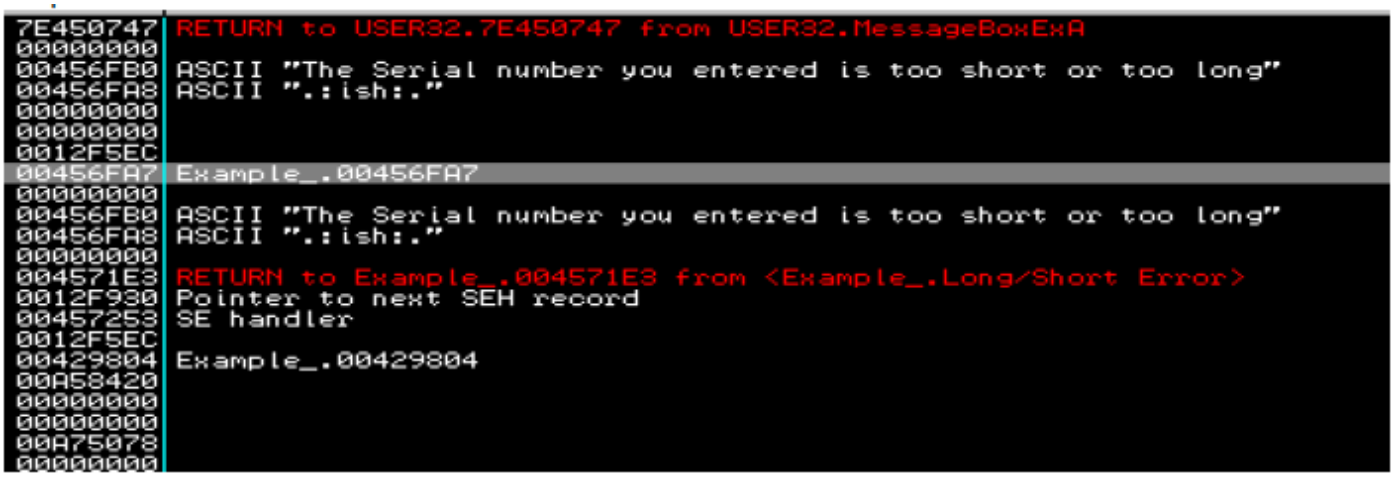
– Mở cửa sổ Call Stack. Nó liệt kế tất cả các hàm được gọi trong chương trình. Và chúng ta chỉ cần quan tâm đến hàm MessageBoxA để trace



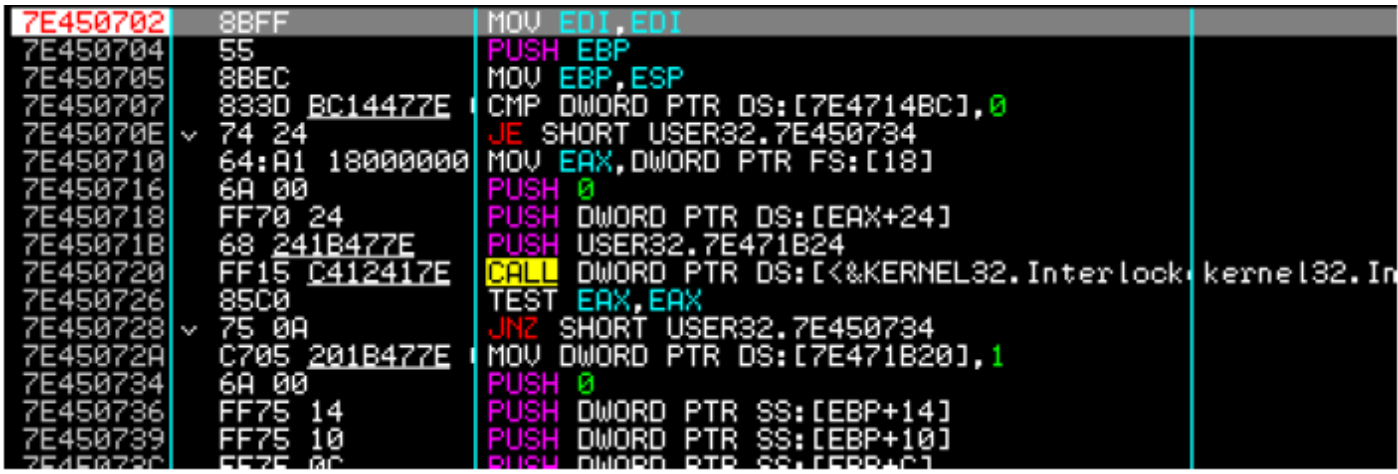
– Điều cần thiết ở đây, là làm thế nào chúng ta biết hàm này được gọi tại địa chỉ nào trong chương trình? Hoàn toàn có thể làm được điều này bằng cách click phải vào dòng chứa hàm MessageBoxA, chọn ***Follow address in stack***



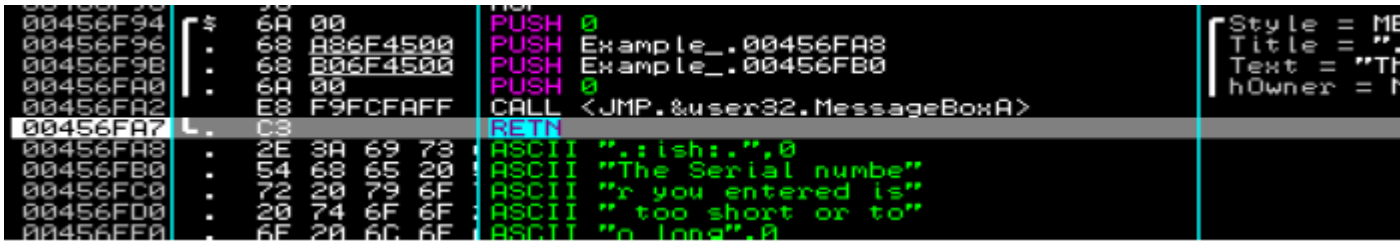
– Bạn sẽ thấy dòng chữ **RETURN to USER32.7E450747 from USER32.MessageBoxExA**



Bạn thấy hàm MessageBoxA nằm trong DLL có tên là USER32.DLL và tại địa chỉ 7E450747. Nếu bạn vẫn chưa hiểu tại sao, thì hãy quan sát code trong USER32.DLL



– Vì thế hàm MessageBoxA này được tìm thấy tại địa chỉ **Example_.00456FA7**



b) Giải pháp đề nghị

Kỹ thuật để tránh việc tracing thông qua stack là một kỹ thuật khó. Có người cho rằng, bạn chỉ cần thay thế tất cả các chỉ thị CALL và RETN thành JMP. Điều này được gọi là **Binary Code Obfuscation**.

Code Obfuscation (CO) là kỹ thuật chuyển đổi code từ dạng mã nhị phân ban đầu của chương trình, thêm vào đó là làm nó rối rắm không thể đọc được và khó bị analyse bởi các disassembly. Mặc dù, kỹ thuật này làm trở ngại những người làm công việc đảo mã, tuy nhiên nó vẫn không bảo vệ được phần mềm, nó chỉ làm sự phân tích mã khó khăn hơn mà thôi.

Ý tưởng cơ bản là kết hợp 2 section .DATA và .CODE lại với nhau. Ngoài ra, obfuscation sẽ thay thế các Opcodes, để tránh việc disassembly và tracing:

Thay thế các lệnh CALL bằng PUSH, POP, RET và JMP. Và sau đó thay thế JMP bằng PUSH và RET.

Original Code	*Obfuscated Code*
PUSH 0	PUSH 0
CALL 7E450747	PUSH EIP + <bytes to next instruction>
	JMP 7E450747
Original Code	*Obfuscated Code*
MOV EBX,1	POP EAX
RETN	JMP EAX
Original Code	*Obfuscated Code*
JMP 00456F94	PUSH 00456F94
CALL 7E450747	RETN

Thay thế các lệnh nhảy rẻ nhánh có điều kiện (JE, JNZ, JL, JG, ...), thêm vào đó cách này có thể làm nản lòng hoặc có thể dẫn họ tới 1 đoạn junk code.

Original Code	*Obfuscated Code*
JMP 00456F94	MOV EAX, 1
	CMP EAX, 0
	JE <**JUNK_CODE**>
	JNE 00456F94

– Để tránh việc tham chiếu trực tiếp địa chỉ của các offset, ví dụ **JMP 00456F94**. Sử dụng một pháp tính toán đơn giản để **obfuscate** và sau đó gọi chúng, ví dụ:

```
MOV EAX, 00456000    ; EAX = 00456000
ADD EAX, 00000F94    ; EAX = 00456F94
JMP EAX              ; JMP 00456F94
```