**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN**

**KHOA CÔNG NGHỆ THÔNG TIN**

**٭٭٭٭٭**

****

**BÁO CÁO ĐỒ ÁN**

**MÔN HỌC: KIẾN TRÚC MÁY TÍNH VÀ HỢP NGỮ**

**ĐỀ TÀI: CRACK PHẦN MỀM**

LỚP : 18CNTN

GIÁO VIÊN HƯỚNG DẪN : Phạm Tuấn Sơn

SINH VIÊN THỰC HIỆN : 18120015 – Trần Duy Đạt

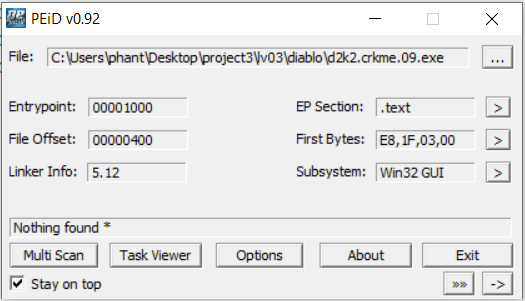
18120019 – Nguyễn Hoàng Dũng

18120020 – Phan Thái Dương

*Thành phố Hồ Chí Minh, ngày 17 tháng 7 năm 2020*

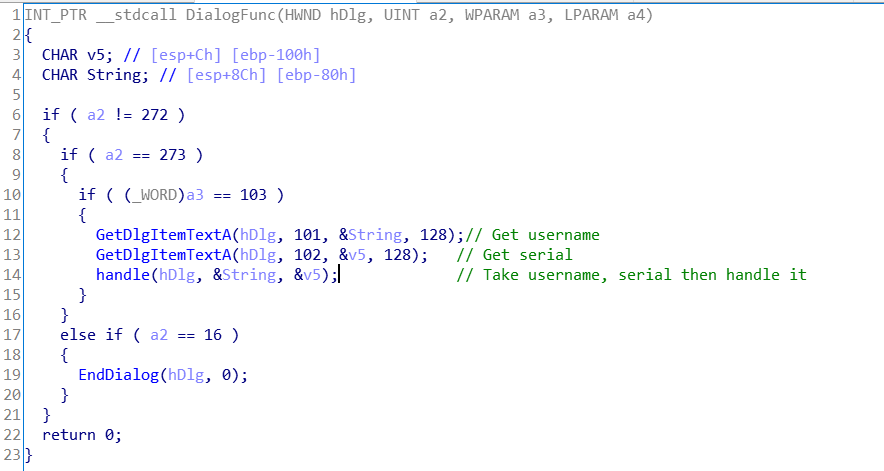
Solution Diablo

Sử dụng PEiD để xem chương trình đã bị pack lại chưa?



* Nothing found nên không cần unpack gì cả

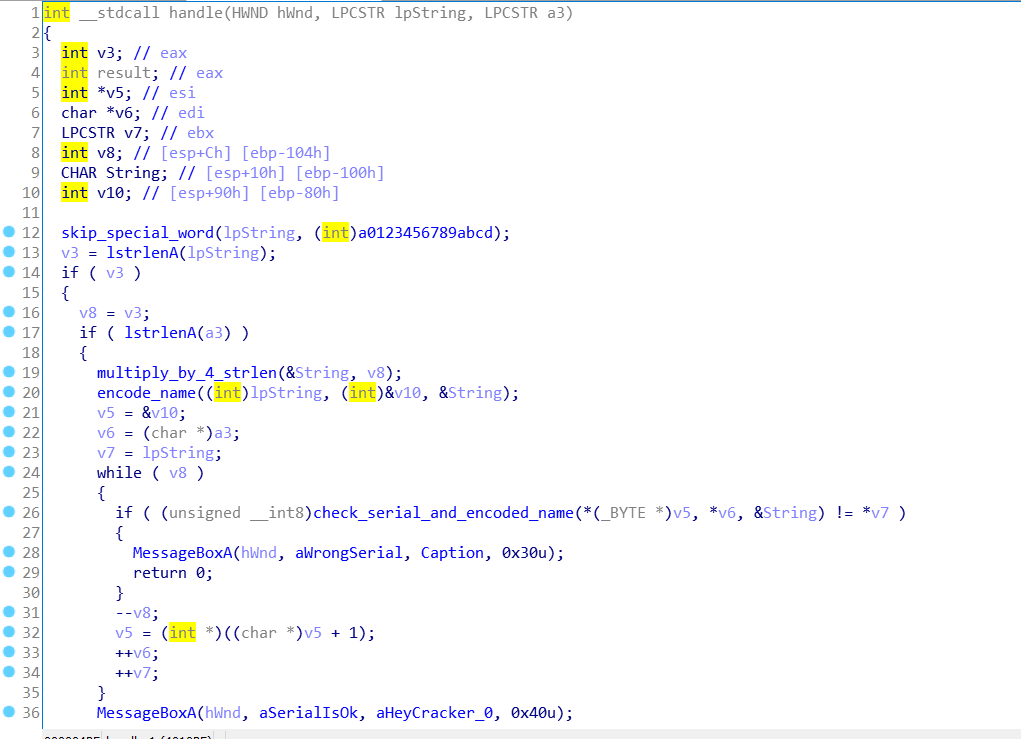
Sử dụng IDA Pro để phân tích, nhận thấy chương trình sẽ gọi hàm DialogFunc:

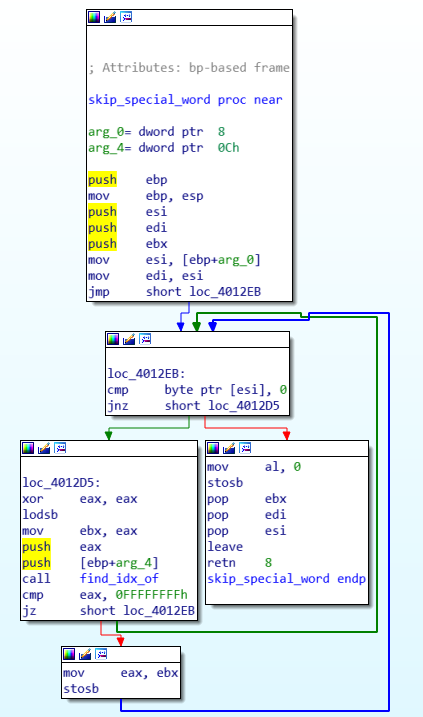


Dùng Resources Hacker ta nhận xét được:

* GetDlgItemTextA: Nhận giá trị name khi người dùng nhập vào box name
* GetDlgItemTextB: Nhận giá trị serial khi người dùng nhập vào box serial

Tiếp tục hàm sẽ truyền serial, name vào hàm handle khi ta click vào nút Check

Chúng ta cùng vào xem hàm handle xử lý như thế nào nhé !!!

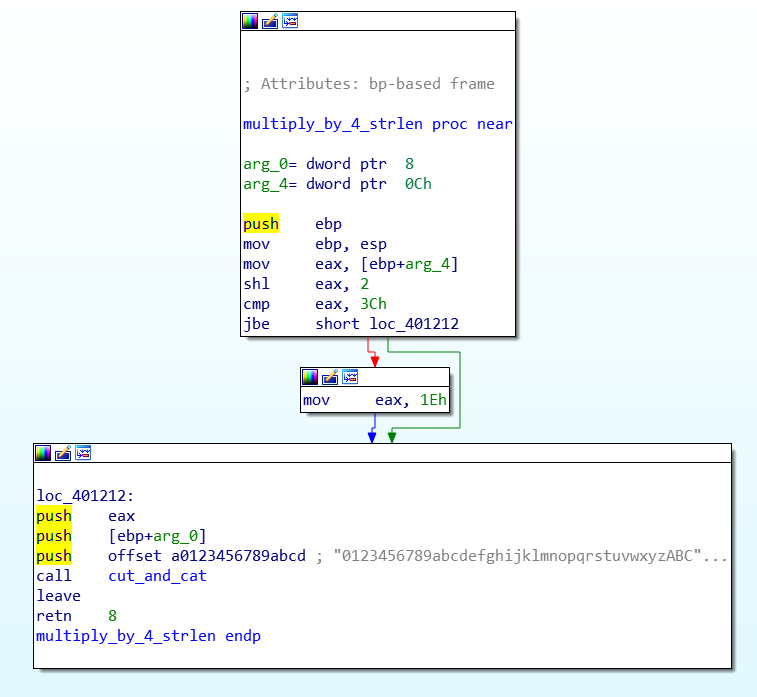


Đầu tiên hàm handle gọi hàm skip\_special\_word và truyền vào biến name

Hàm skip\_special\_word: đầu vào chuỗi name và trả về chuỗi đã loại bỏ các ký tự đặc biệt (chỉ nhận các kí tự a – z, A – Z, 0 – 9)

VD: user\_name^^ 🡪 username

Sau khi hàm skip\_special\_word trả về, hàm handle tiếp túc thực hiện và gọi hàm multiply\_by\_4\_strlen

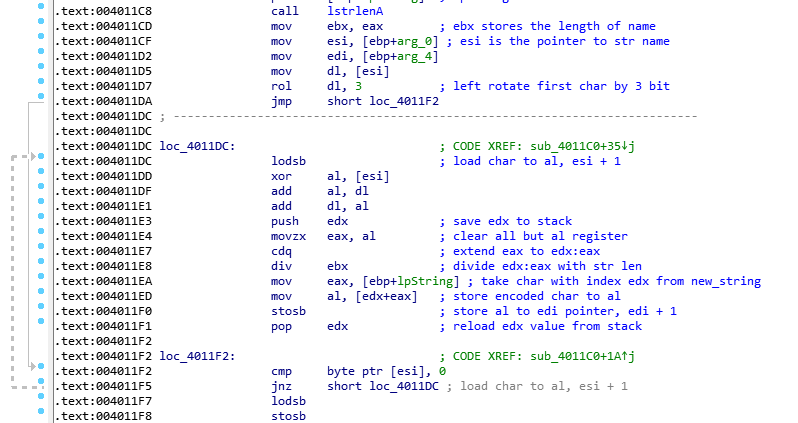
Nhiệm vụ của hàm là trả về giá trị của strlen \* 4 với strlen là độ dài chuỗi name được nhập.

Nếu vượt quá thì sẽ trả về giá trị mặc định là 30.  
VD: abcd 🡪 strlen = 4 🡪 return 16

Sau đó, hàm multiply sử dụng giá trị vừa tìm được để gọi đến hàm cut\_and\_cat và tạo ra một chuỗi mới new\_string đã được rotate dựa vào base\_string = ‘01234…abc…xyz…ABC…XYZ’

VD: multiply\_by\_4\_strlen = 16

🡪 new\_string = ‘ghijk….XYZ012…789abcdef’

Sau khi hàm cut\_and\_cat được chạy xong, hàm handle tiếp tục thực hiện và gặp hàm encode. Nội dung của hàm encode được chú thích như sau:

Hàm này sử dụng các phép tính để mã hóa các kí tự trong name thành 1 kí tự mới thuộc new\_string có index là giá trị vừa tính được.

VD: name = ‘abcd’, startPosition = 16 (tính ở hàm multiply\_by\_4\_strlen)

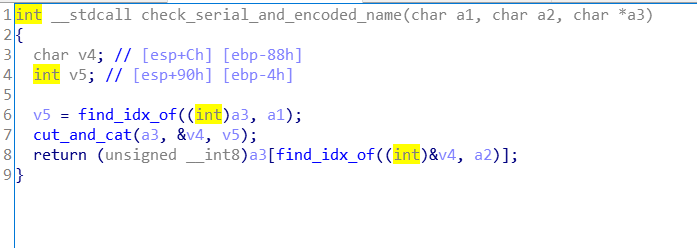
🡪 Return ‘uGcD’

Minh họa thuật toán mã hóa name bằng cplusplus:



Cuối cùng, dựa vào serial và chuỗi đã được mã hóa ở trên, hàm check\_serial and\_encoded\_name sẽ kiểm tra key và name có đúng hay chưa?

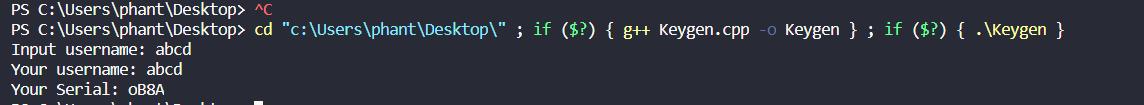
Hàm check\_serial\_and\_endcoded\_name:



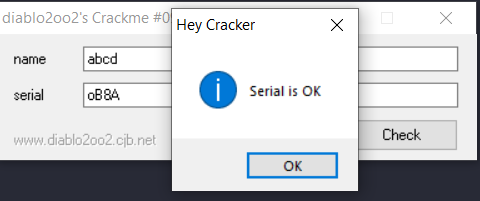
Vậy, bây giờ đơn giản chúng ta làm ngược lại các bước trên để tạo được keygen



Chạy thử chương trình:

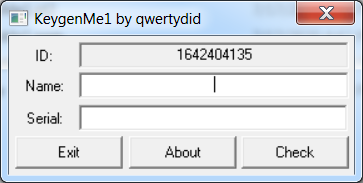


Vậy là đã hoàn thành !

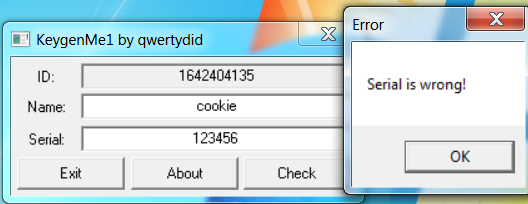


Solution KeygenMe1

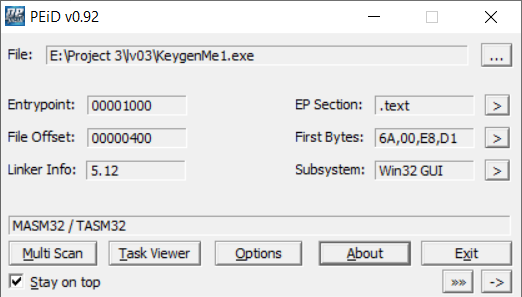
Mở chương trình lên:



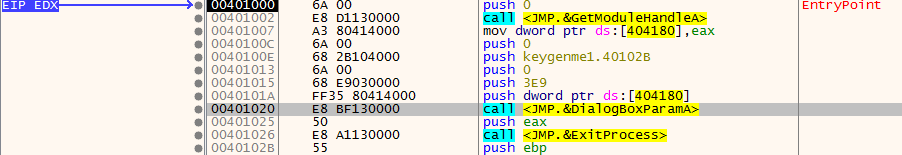
Chương trình tạo ra 1 DialogBox với ID có sẵn và cần phải nhập Name và Serial. Nhập tùy ý vào ví dụ Name là “cookie”, Serial là “123456”. Sau đó nhấn Check thì sẽ có 1 Message box hiện lên:



Tạm thời tắt đi và dùng PEiD để kiểm tra:

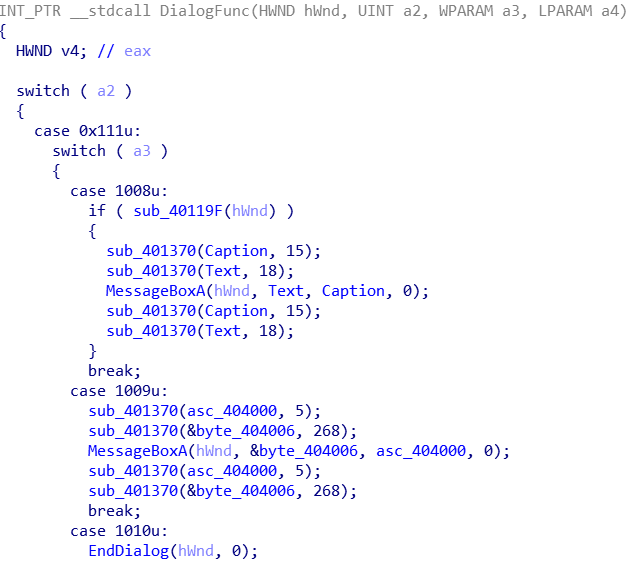


Ta thấy chương trình được code bằng MASM32.

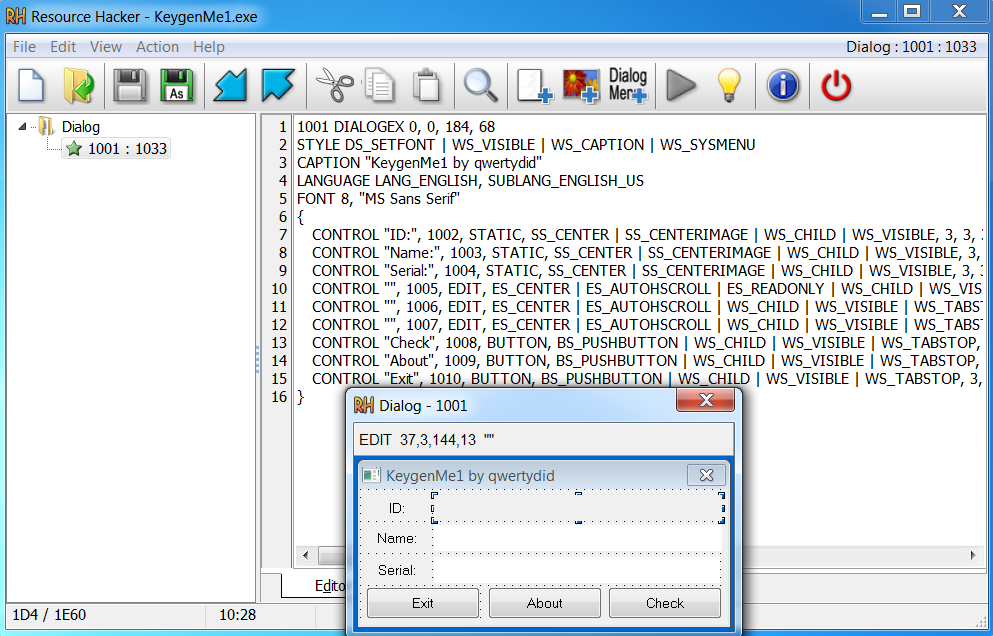
Mở chương trình trong x32 debug:

Ta thấy nó gọi hàm DialogBoxParamA với tham số lpDialogFunc là hàm tại 0040102B. Ta dùng IDA Pro để mở và decompile để xem hàm DialogFunc này.

Ta có thể lập tức thấy ngay có 1 switch case xử lý dựa theo message a2 gồm có: WM\_INITDIALOG (0x0110), WM\_COMMAND (0x0111) và WM\_CLOSE (0x0010). Tham khảo thêm tại: [link](https://docs.microsoft.com/en-us/windows/win32/api/winuser/nc-winuser-dlgproc)

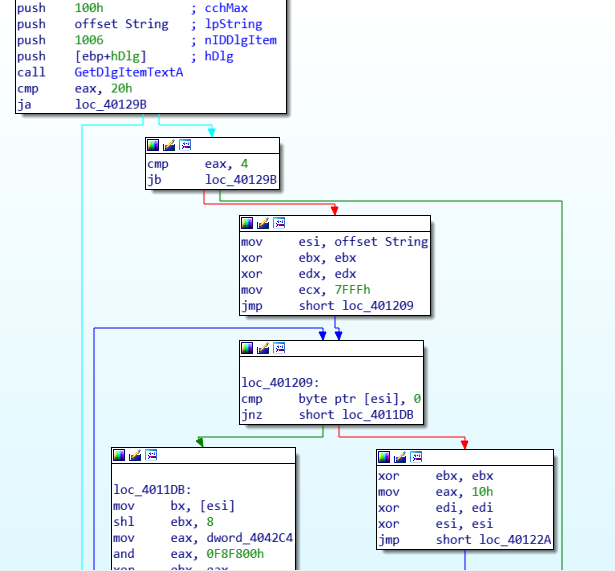


Tạm thời chỉ chú ý đến message 0x111 vì nó sẽ chứa các bước xử lý khi ta nhấn nút Check. Dùng Resource Hacker để xem ID các thành phần của DialogBox:

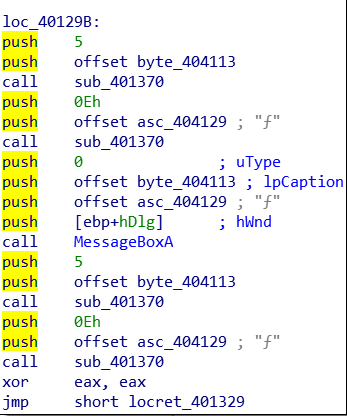


Ta cần chú ý đến case 1008. Khi nhấn nút Check, hàm tại 0040119F sẽ được gọi và kiểm tra kết quả trả ra, nếu True thì sẽ xử lí tiếp còn False thì sẽ không xử lí. Ta có thế đoán hàm này chính là hàm kiểm tra Name và Serial nhập vào.

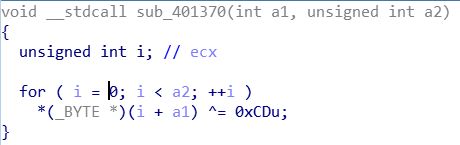
Ta vào trong hàm này để xem:



Đầu tiên hàm sẽ gọi GetDlgItemTextA để lấy giá trị từ vùng có ID 1006, chính là Name và lưu giá trị Name vào vùng nhớ String. Hàm sẽ kiểm tra độ dài của Name có nằm trong khoảng từ 4 – 32 hay không, nếu không thì sẽ xử lí như sau:

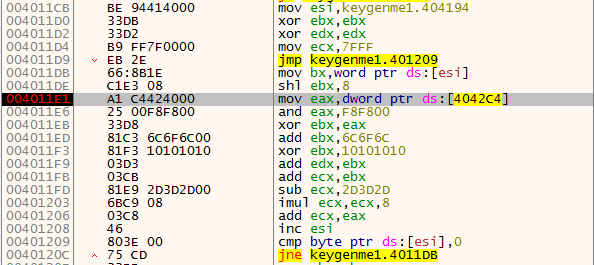


Có thể hiểu là hàm sẽ hiển thị một MessageBox có Caption lưu tại byte\_404113 và Text lưu tại asc\_404129. Tuy nhiên trước đó thì nó đã gọi hàm tại địa chỉ 00401370 để xử lí 2 vùng nhớ này, ta có thể decompile để xem chức năng của hàm sub\_401370.



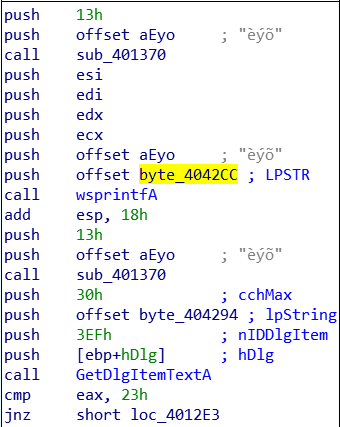
Đại loại là hàm sub\_401370 sẽ xor từng byte trong a2 bytes bắt đầu từ byte a1 với 0xCD. Đây chính là hàm dùng để encode/decode data, nó chỉ decode data khi nào cần đọc và sau đó sẽ lập tức encode lại.

Quay trở lại với hàm sub\_40119F, khi độ dài chuỗi String trong khoảng 4 – 32 thì tiếp theo sẽ là một chuỗi các thao tác gồm 2 vòng lặp để tính toán trên các thanh ghi, ta có thể vào x32dbg và debug bằng cách F8 qua từng câu lệnh, như thế sẽ bắt gặp một câu lệnh cần kiểm tra:



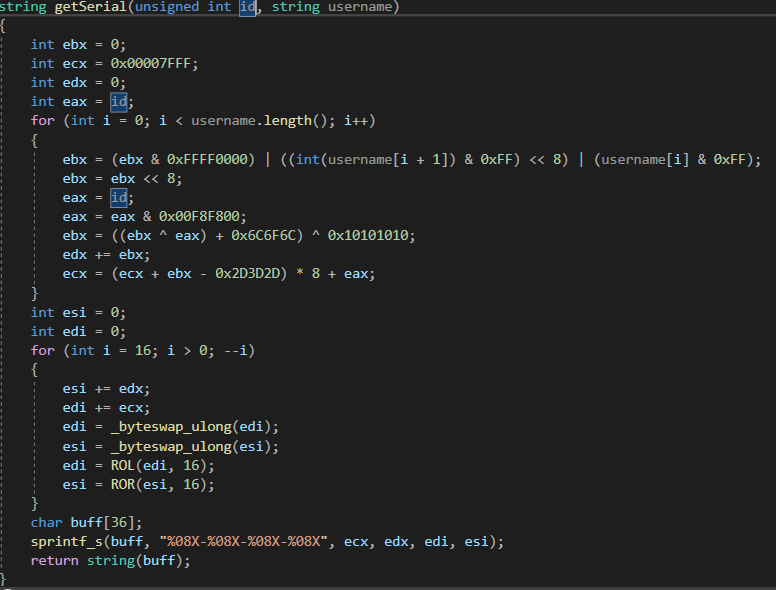
Vùng nhớ 0x004042C4 đang lưu giá trị 0x61E51927. Giá trị này ở đâu mà có, đoán xem? 😊 Đó chính là giá trị hexa mà ID trong DialogBox đang giữ, tạm thời ta cứ bỏ qua việc làm sao để tính được ID này.

Sau 2 vòng lặp tính toán trên các thanh ghi thì sẽ tiếp tục xử lí đoạn code sau:

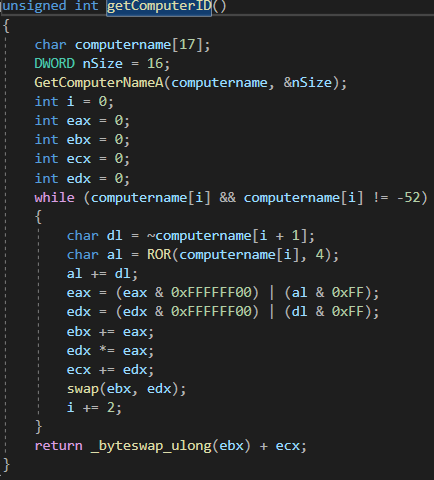
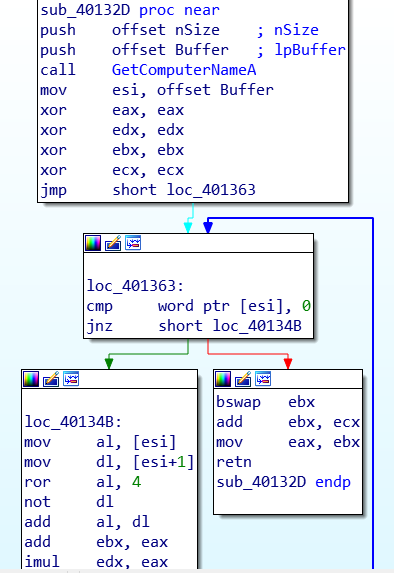


Hàm wsprintfA (tham khảo [link](https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-wsprintfa)) được gọi để ghi giá trị của 4 thanh ghi ecx, edx, edi, esi vào buffer tại byte\_4042CC theo format lưu tại aEyo (0x00404160), format này có thể xem được sau khi được decode bằng hàm sub\_401370, đó là “%08X-%08X-%08X-%08X”. Cuối cùng Serial đúng sẽ được tạo ra tại địa chỉ 0x004042CC, nó sẽ được so sánh với Serial mà ta nhập vào, nếu khác thì hàm 0x0040119F trả về 0, ngược lại trả về 1.

Vậy từ các bước tính toán ra một Serial đúng, ta viết hàm getSerial như sau:



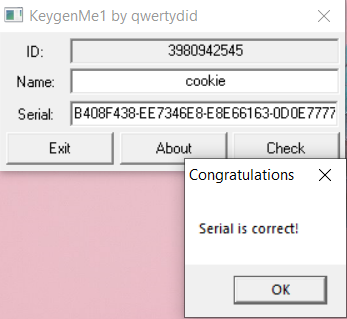
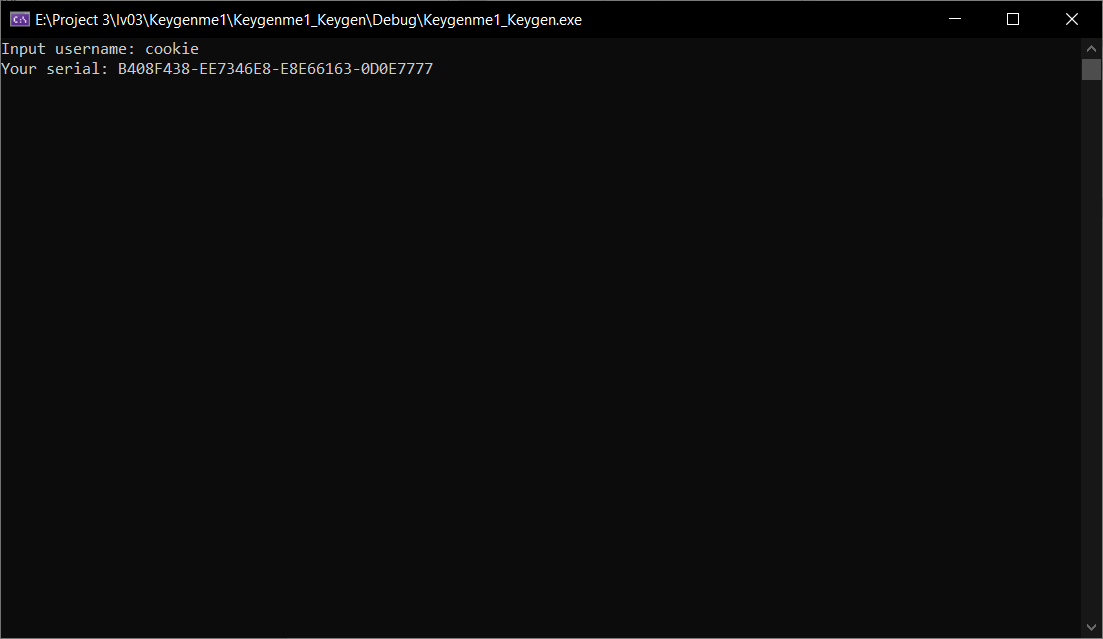
Để tính được giá trị ID thì ta sẽ vào xem hàm sub\_40132D. Hàm này sẽ lấy chuỗi tên của máy tính hiện tại bằng cách gọi hàm GetComputerNameA và tính toán trên chuỗi này để cho ra ID.



Từ đoạn mã bên trái viết thành hàm getComputerID như trên

Kiểm tra Keygen vừa viết (đang test trên 1 máy khác nên ID sẽ khác).

Ta sẽ thử với name: cookie



Thành công !!!

Solution Phoenix3

Tool: PEiD, DIE, x32dbg, IDA, Hex Editor

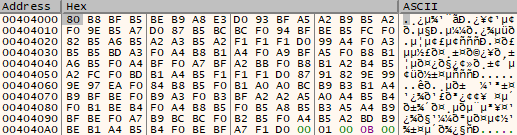
Quan sát ban đầu:

1. Chương trình được viết bằng MASM [x32] (có thể dùng DIE hoặc PeiD)
2. Chạy thử với x32 debug không tìm được goodboy hay badboy để bắt đầu
3. Sử dụng graph view của IDA nhận thấy các hàm gọi nhau rất khó nhìn.
4. Tìm thấy hàm DialogFunc để bắt đầu nhưng khi set breakpoint và debug bằng x32dbg không gọi vào được (có thể do bị loop ở một thread khác)

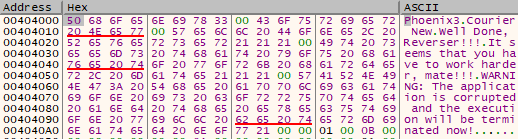
Hướng tiếp cận:

Khi chạy thử lại chương trình bằng x32 debug và quan sát vùng nhớ .data tại địa chỉ 00404000, nhận thấy ban đầu có một đoạn dữ liệu không rõ nghĩa, nhưng sau khi F9 chạy thì đoạn dữ liệu đó hiện lên rõ nghĩa.

* Cần tìm kiếm đoạn code nào đã giải mã được dữ liệu thành bản rõ

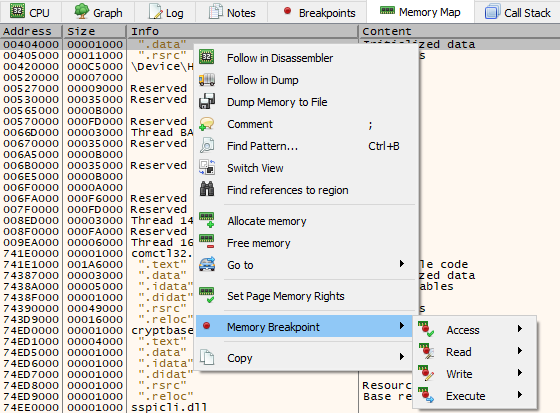


Hình 1



Hình 2

Trong hình 2 data kết thúc tại dấu ! (0x21). Byte kề sau có giá trị 0 và vị trí tương ứng trong hình 1 có giá trị hex là D0. Vị trí này có địa chỉ là 4040A0.



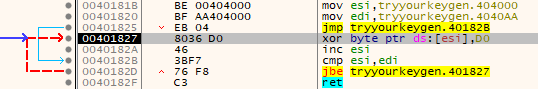
Để tìm kiếm đoạn code nào có truy cập vào vùng dữ liệu này:

Trong Memory Map, click chuột phải vào địa chỉ bắt đầu của .data để set breakpoint trigger đến bất kì dòng code nào xử lý việc đọc và ghi đến vùng dữ liệu này.

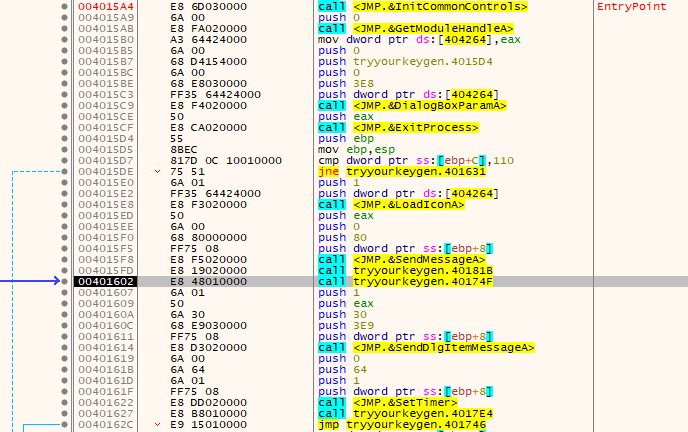
Debug lại chương trình, F9 một vài lần sẽ đến được đoạn code truy cập đến .data

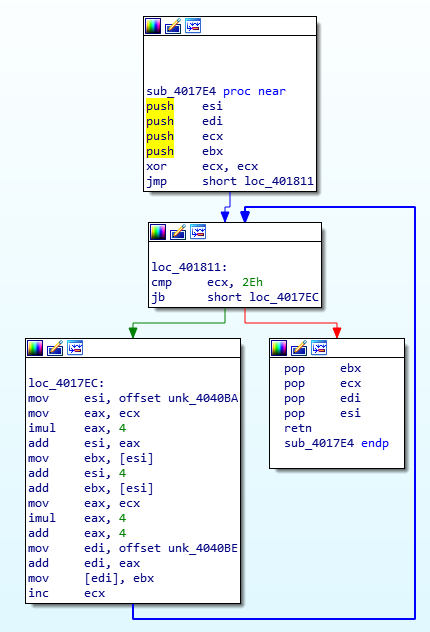
Nhận xét:

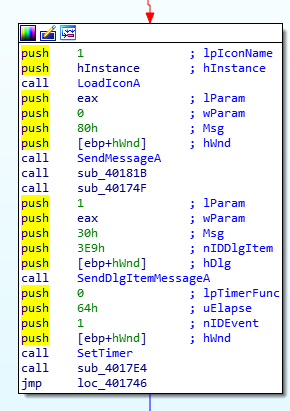
1. Gọi hàm này là **decrypt\_data**, địa chỉ gọi đến là **40181B**
2. Thanh ghi ESI trỏ đến địa chỉ .data, thanh ghi EDI trỏ đến vị trí byte D0
3. Vòng lặp thực hiện xor từng byte mà ESI trỏ đến với D0. Nếu ESI = EDI, nghĩa là byte tại EDI là D0 được xor với D0 sẽ cho giá trị 0, trùng với giá trị byte sau dấu ! nhận được ban đầu khi xem xét vùng dữ liệu sau khi giải mã. Lúc này vòng lặp cũng kết thúc vì điều kiện jbe không thỏa



Sau khi **decrypt\_data** thực hiện xong và quay về nơi gọi hàm, ta đến được một đoạn code như sau:



Đây là một đoạn code thuộc một nhánh điều kiện của hàm DialogFunc mà IDA đã phân tích được:

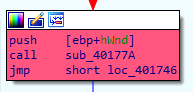
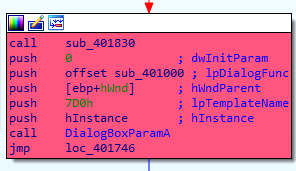


Ngoài việc gọi đến **decrypt\_data** (**40181B**), hàm này cũng gọi đến các địa chỉ **40174F** (1) và **4017E4** (2).

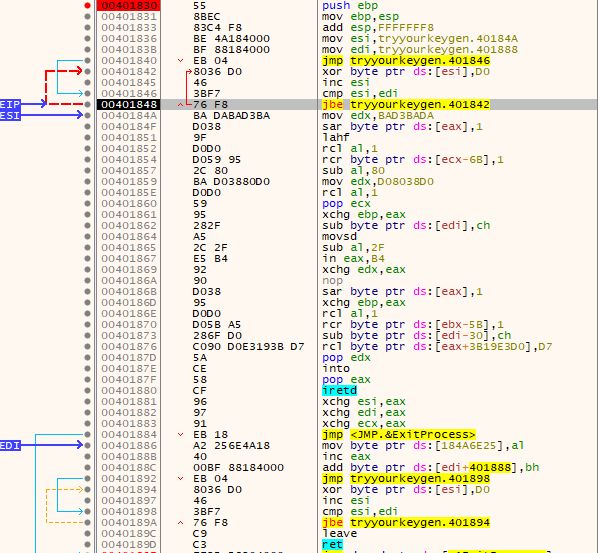
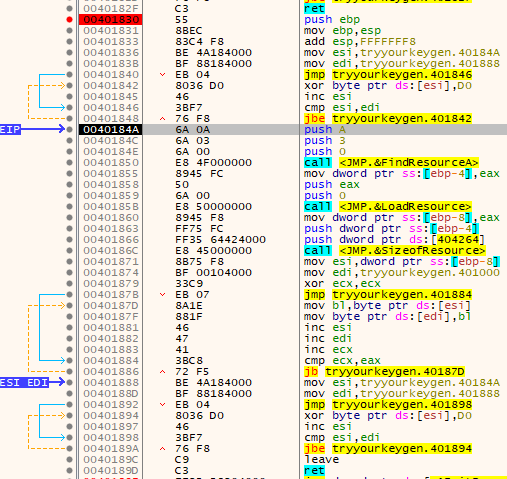
Hàm (1) theo IDA phân tích chỉ gọi lệnh CreateFontA nên ta tạm thời bỏ qua.

Hàm **4017E4** (bên phải) thực hiện 1 vòng lặp với các thao tác đọc và sửa dữ liệu trong vùng nhớ .data, do con trỏ ESI quản lý.

Ta tạm thời note lại 2 hàm này và xem xét các hàm khác thuộc các nhánh điều kiện khác trong DialogFunc.



Các hàm **401830**, **401000**, **40177A** cũng có sự xuất hiện của 1 đoạn lệnh: xor lần lượt các byte (do thanh ghi ESI trỏ đến) với D0 (giống với hàm **decrypt\_data**)

Đặt breakpoint tại **401830** và click nút register để trigger breakpoint:

Hình 3.1

Hình 3.2

Hình 3.1:

* Hai thanh ghi ESI và EDI đang khoanh vùng 1 đoạn code.
* Địa chỉ mà ESI trỏ đến chứa nội dung của 1 câu lệnh.
* Lấy từng byte tại ESI xor với D0, ta sẽ được một list các câu lệnh mới !

Hình 3.2:

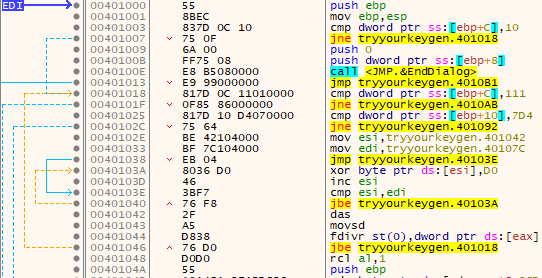
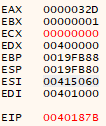
* Các câu lệnh từ 184A đến 1888 đã được giải mã (\*)
* Con trỏ EIP tiếp tục chạy và sẽ thực hiện các câu lệnh đã được giải mã
* Tại 1888, ESI và EDI lại được nạp tiếp địa chỉ đầu và cuối của (\*)
* Khi thực hiện xong, (\*) được mã hóa lại cũng bằng phép xor D0 !

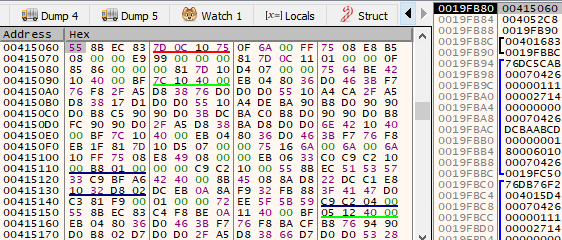
Như vậy, để việc đọc code và debug dễ hơn:

1. Nops 2 đoạn code mã hóa đầu và cuối (mỗi đoạn 7 câu lệnh)
2. Save file có code giải mã (\*) thay thế đoạn code cũ (backup khi reverse)
3. Mặt khác, (\*) cũng là nội dung chính của hàm mà ta cần quan tâm.

Đặt tên hàm này là **onclick\_register**. Sau khi được hiện 1 loạt các lệnh find, load, getsize resource. Ta có thể thấy sự thay đổi của EDI và ESI như sau:

1. EDI giữ địa chỉ 401000, (DialogFunc có push vào stack địa chỉ này)
2. ESI trỏ đến địa chỉ 415060 nằm trong vùng .data
3. EAX chứa size của 1 cái gì đó ta chưa rõ



Hình 3.2: câu lệnh tại 40187B bắt đầu vòng lặp:

Copy các byte từ địa chỉ ESI đến địa chỉ của EDI.

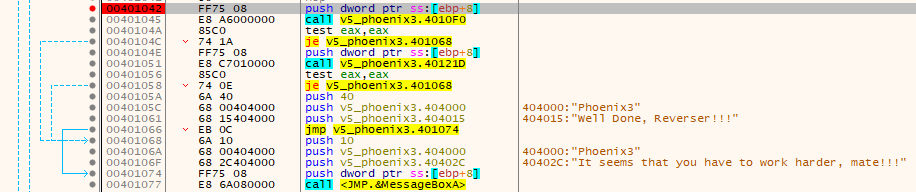
Size cần copy là giá trị của EAX.

các byte do ESI trỏ đến rất giống với các byte trong các câu lệnh tại EDI.

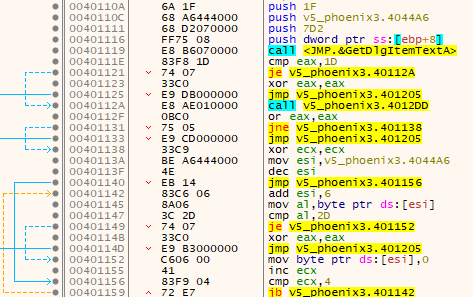
Như vậy, hàm **401830** (**onclick\_register**) được chạy trước khi gọi hàm **401000** và luôn đảm bảo mã hóa cho hàm **401000**.

Vậy ta chỉ cần nops luôn đoạn code từ 40187B -> 401886.

Tiếp tục xem xét hàm **401000**:

Đặt bp MessageBoxA. Sau đó nops hết đoạn code giải mã ở đầu và mã hóa ở cuối hàm. Kết quả như sau:

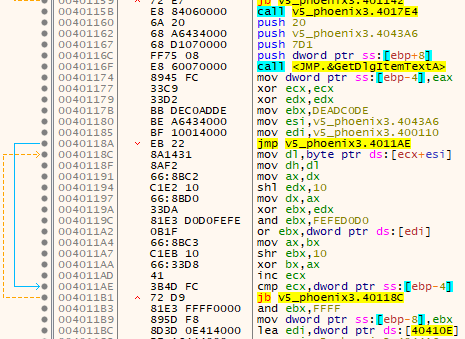
Vậy để đến được goodboy, ta cần phải pass được 2 hàm **4010F0** và **40121D**. Đây là 2 hàm mà ban đầu IDA không phân tích đến được và 2 hàm này cũng đều có code mã hóa và giải mã bằng phép xor với D0.

Xem xét hàm **4010F0**. Ta cũng nops hết các đoạn code giải mã và mã hóa trước.

Đầu tiên lấy độ dài chuỗi serial và so sánh với 0x1D (độ dài 29).

Hàm **4012DD** kiểm tra serial chỉ chứa số và dấu – (0x2D)

Loop kiểm tra serial có bao gồm 5 bộ 5 chữ số cách nhau dấu – hay không.

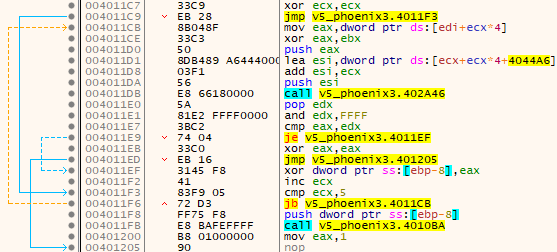
Hàm **4017E4** tạo sẵn 47 số đầu tiên của dãy fibo trong .data

esi giữ địa chỉ name

edi giữ địa chỉ giá trị 0x0159A3

ebx lưu mã hóa của name

mov ebx vào stack (1)

edi giữ địa chỉ fibo[21]

eax lưu giá trị số fibo

esi trỏ đến 5 digit tiếp theo thuộc serial

Hàm **402A46** lấy giá trị số nguyên của esi

[ebp-8] là giá trị từ (1)

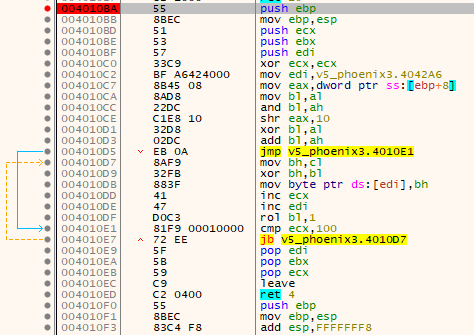
Xem xét giá trị tại [EBP – 8] trong đoạn code trên. Khi được move vào stack tại (1) thì là giá trị của EBX, tức là mã hóa của name sau khi xor 0xFFFF. Sau đó lại được xor với EAX (câu lệnh 4011EF), mà EAX là giá trị số nguyên trả về của 1 bộ 5 digit từ serial được input vào hàm **402A46**.

Ta đang mong muốn EAX = EDX để không phải chạy tiếp lệnh {xor eax, eax} và nhảy thoát khỏi hàm (rơi vào badboy).

Trước đó EDX được pop ra từ stack và and 0xFFFF. Suy ra chỉ có 4 byte cuối của EAX là được bật. Mà ở trên đã khẳng định chỉ có 4 byte cuối của EBX được bật. Vậy nên giá trị tại [EBP – 8] luôn thỏa có chỉ có 4 byte cuối được bật.

Mặt khác, giá trị EDX được pop từ stack chính là giá trị của 1 số fibonacci đã được xor với mã hóa của name tại câu lệnh 4011CE và sau đó push vào stack. Vậy ta đã xác định được cơ chế tính toán serial đúng.

Bây giờ ta cần xem xét hàm **4010BA** được gọi cuối cùng trước khi đến được {mov eax, 1} để rơi vào goodboy. Hàm này nhận tham số đầu tiên trên stack là giá trị tại [EBP – 8]



edi trỏ đến 1 vùng nhớ trống

eax lưu giá trị tham số

1 câu lệnh shift right (!!!)

1 vòng lặp chạy 256 lần.

ASCII cũng có 256 kí tự …

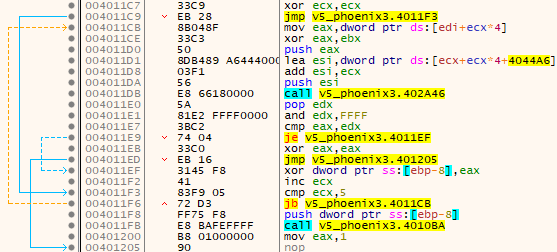
Câu lệnh (!!!) shift right 16 bit. Trong khi đó EAX chỉ có bật 4 byte cuối được bật.

Như vậy cậu lệnh (!!!) reset EAX = 0. Tức là tham số truyền vào không có tác dụng ?

Để ý hai câu lệnh kế trước có thể được viết lại như sau: {and al, ah} {mov bl, al}.

Vậy là vẫn có 1 phần của EAX là AL được lưu vào BL. Tuy nhiên …

Cùng xem lại đoạn lệnh đã tạo ra tham số input cho hàm **4010BA**:



Gọi tham số là Q, chính là giá trị tại [EBP – 8], là mã hóa của name.

Nhắc lại là chúng ta cần EDX = EAX và ở trên đã chỉ ra được rằng với mỗi vòng lặp thì EAX chính là giá trị 1 số fibo xor mã hóa của name, tức là Q.

Tại loop 1: EAX = fibo[21] xor Q[0] (Lúc này [EBP – 8] vẫn = mã hóa của name)

Tại các loop sau: EAX vẫn là fibo[i] xor Q[0] (Tuy nhiên [EBP – 8] đã có thay đổi)

Mình tạm bỏ qua việc and 0xFFFF tại mỗi loop …

Thử note lại các thao tác trong 1 loop (kí hiệu ^ cho phép xor)

1. Q[1] = Q[0] ^ fibo[21] ^ Q[0]
2. Q[2] = Q[1] ^ fibo[22] ^ Q[0]
3. Q[3] = Q[2] ^ fibo[23] ^ Q[0]
4. Q[4] = Q[3] ^ fibo[24] ^ Q[0]
5. Q[5] = Q[4] ^ fibo[25] ^ Q[0]

Các bạn có nhìn ra được điều gì không …?

Thật dễ dàng! Vì tại Q[5] thì tất cả Q[0] đã bị triệt tiêu !!!

Như vậy: Input cho hàm **4010BA** luôn là f[21] ^ f[22] ^ f[23] ^ f[24] ^ f[25]

Nhưng mà bất ngờ hơn là kết quả của 1 đống xor này lại là 0x1902F. Và khi biểu diễn trên các thanh ghi như EAX thì AL and AH = 0 !!!

Vậy thì toàn bộ đoạn code trước vòng lặp trong hàm **4010BA** đúng thật là vô nghĩa. Và trong vòng lặp đầu tiên có {mov bh, cl} làm cho EBX cũng bị reset = 0. Như vậy toàn bộ vòng lặp 256 lần chỉ có ý nghĩa duy nhất là mỗi lần tăng ECX 1 đơn vị. Sau đó {mov bh, cl} và chuyển giá trị thanh ghi BH đến vùng nhớ EDI đang trỏ đến.

Mặt khác thì 0xFF = 255 nên thanh ghi CL cũng không thể bị overflow.

Nhưng mà tạo bảng ASCII để làm gì?

Chúng ta xem xét đến hàm cần pass tiếp theo tại **40121D**.

Hàm này cũng có chứa đoạn code mã hóa và giải mã. Ta cũng nops tương tự các hàm trước và update lại code mới như hình dưới.

Đầu tiên hàm sẽ lấy chuỗi activation code và lưu tại địa chỉ 4044C5. Ta cần đảm bảo độ dài chuỗi là 48 (0x30) kí tự.

Sau đó hàm lấy một hằng số nào đó tại 4000C0 và lưu vào EBX

Địa chỉ 4042A6 chính là nơi bắt đầu của bảng mã ASCII ta đã tạo sẵn.

Con trỏ ESI tiếp tục lấy một số giá trị dword từ bảng và tính toán trên EBX.

Vì mọi giá trị đến đây là đều hằng số nên ta có thể chạy thử 1 lần và mặc định luôn EBX sẽ được khởi tạo là 0xFFFFFFFC.

Tiếp theo là 1 vòng lặp bắt đầu bằng việc xor EBX với 1 giá trị EAX dword cũng trong phạm vi vùng nhớ của bảng mã ASCII.

Cần ghi nhớ ngay từ lúc vào đầu hàm đã có lệnh {add esp, FFFFFFF4}, tức là tăng stack thêm 12 byte (do stack up ngược nên giảm địa chỉ là tăng kích thuốc)

Từ EBP đến ESP bây giờ đang có 16 byte, tức là 4 dword, tính cả EBP

Quy tắc như sau:

1. push EBX vào stack.
2. Push 4 kí tự đầu tiên vào stack và pop vào [EBP – C] (&EBX + 4)
3. Push 4 kí tự tiếp theo vào stack và pop vào [EBP – 8] (&EBX + 8)
4. EAX lưu địa chỉ stack của giá trị tại [EBP – C]
5. Gọi hàm **402A0F** để đổi bộ 8 kí tự có địa chỉ lưu tại EAX sang số hex
6. Quay trở về và so sánh giá trị với đầu stack

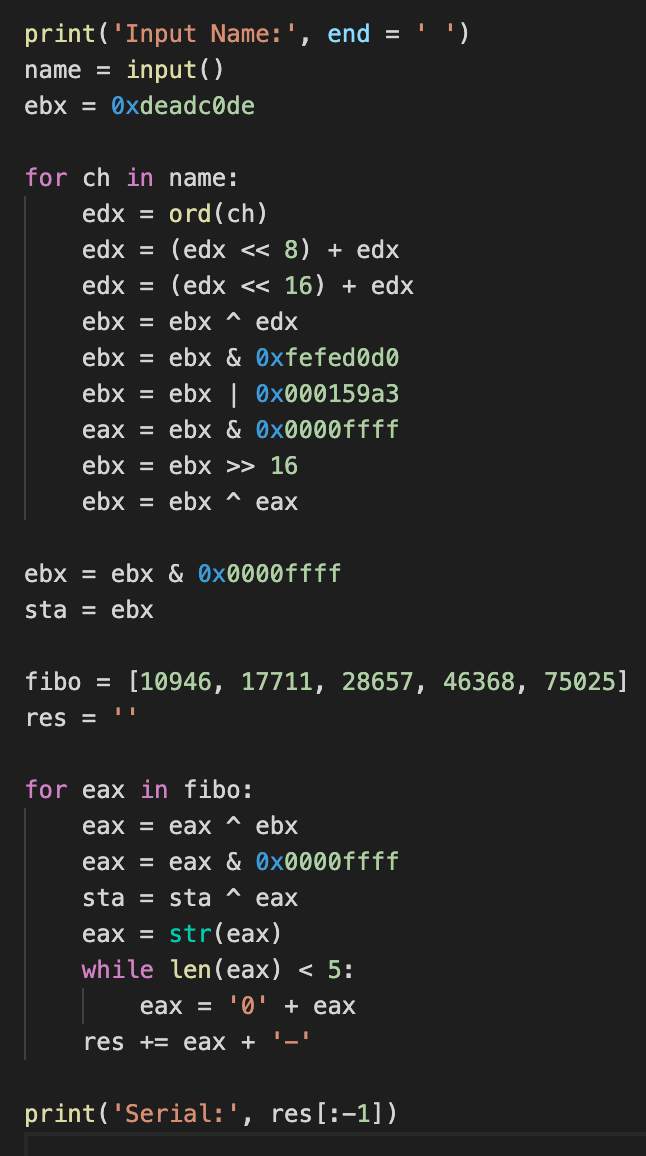
Loop 6 lần quy tắc trên.

Nhận xét: Mỗi lần loop xét 8 kí tự liền kề, 6 lần như vậy là đủ 48 kí tự.

Mặt khác, tuy chỉ lưu địa chỉ của 4 kí tự đầu tiên tại vị trí đang xét (1 kí tự 8 bit). Nhưng khi đổi kí tự về dạng hex thì 1 kí tự là 4 bit nên vừa đủ cho thanh ghi lưu.

Và vì luôn lấy chuỗi tại [EBP – C], nên activation code nhập vào luôn xét theo thứ tự trái sang phải. Vậy mỗi giá trị EBX sau 6 vòng ghép theo thứ tự chính là chuỗi activation code đúng:

FCFDFEFCFBFBFBF8F0F1F2F0FFFFFFFCECEDEEECFBFBFBF8

Code tìm serial như sau:

