

**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN**



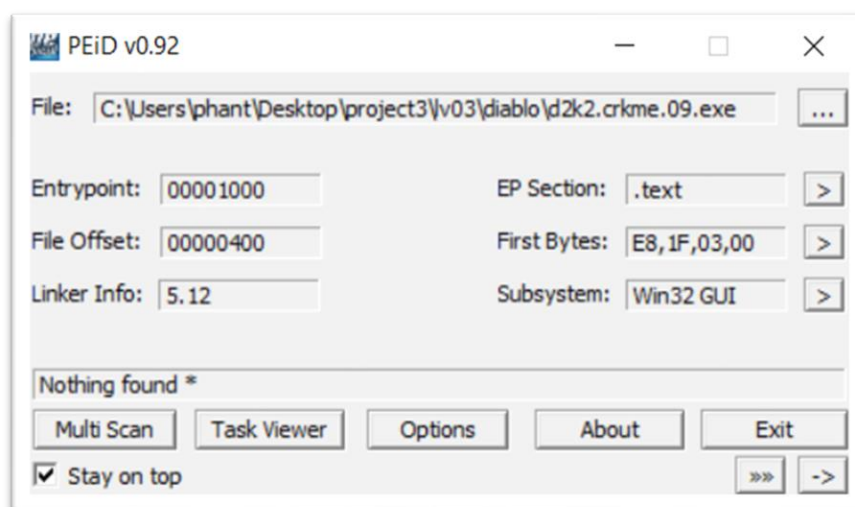
**BÁO CÁO ĐỒ ÁN
MÔN HỌC: KIẾN TRÚC MÁY TÍNH VÀ HỢP NGỮ
ĐỀ TÀI: CRACK PHẦN MỀM**

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:

```
1 INT_PTR __stdcall DialogFunc(HWND hDlg, UINT a2, WPARAM a3, LPARAM a4)
2 {
3     CHAR v5; // [esp+Ch] [ebp-100h]
4     CHAR String; // [esp+8Ch] [ebp-80h]
5
6     if ( a2 != 272 )
7     {
8         if ( a2 == 273 )
9         {
10             if ( (_WORD)a3 == 103 )
11             {
12                 GetDlgItemTextA(hDlg, 101, &String, 128); // Get username
13                 GetDlgItemTextA(hDlg, 102, &v5, 128); // Get serial
14                 handle(hDlg, &String, &v5); // Take username, serial then handle it
15             }
16         }
17         else if ( a2 == 16 )
18         {
19             EndDialog(hDlg, 0);
20         }
21     }
22     return 0;
23 }
```

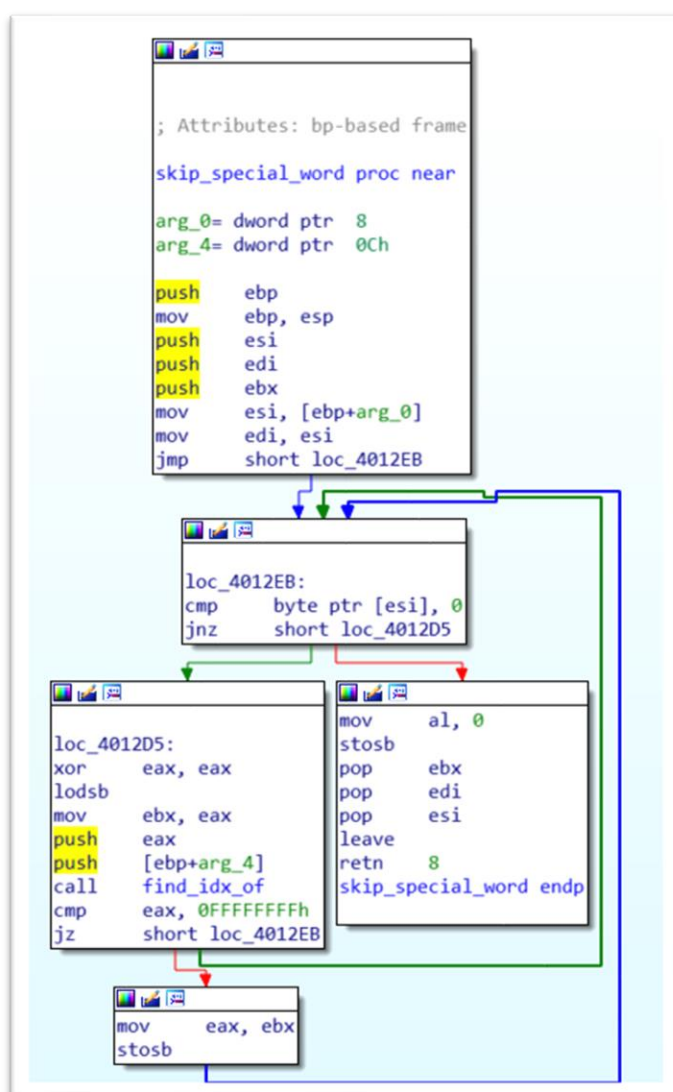
Dùng Resources Hacker ta nhận xét được:

- 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 !!!

```
12| skip_special_word(lpString, (int)a0123456789abcd);
13| v3 = strlenA(lpString);
14| if ( v3 )
15| {
16|     v8 = v3;
17|     if ( strlenA(a3) )
18|     {
19|         multiply_by_4_strlen(&String, v8);
20|         encode_name((int)lpString, (int)&v10, &String);
21|         v5 = &v10;
22|         v6 = (char *)a3;
23|         v7 = lpString;
24|         while ( v8 )
25|         {
26|             if ( (unsigned __int8)check_serial_and_encoded_name(*(_BYTE *)v5, *v6, &String) != *v7 )
27|             {
28|                 MessageBoxA(hWnd, aWrongSerial, Caption, 0x30u);
29|                 return 0;
30|             }
31|             --v8;
32|             v5 = (int *)((char *)v5 + 1);
33|             ++v6;
34|             ++v7;
35|         }
36|         MessageBoxA(hWnd, aSerialIsOk, aHeyCracker_0, 0x40u);
```

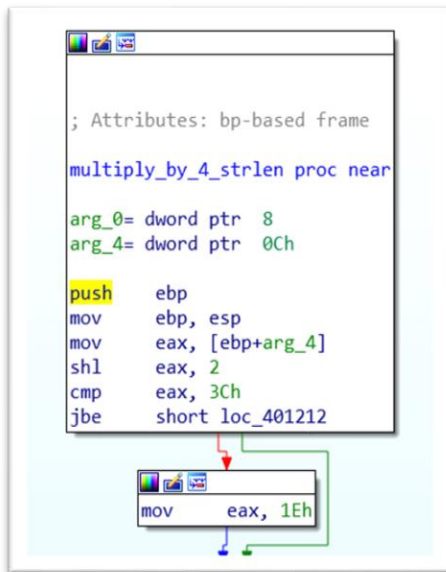


Đầu tiên hàm handle gọi hàm skip_special_word và truyền vào biến name

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

VD: user_name^^ → username

Sau khi hàm `skip_special_word` trả về, hàm `handle` tiếp tục thực hiện và gọi hàm `multiply_by_4_strlen`



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

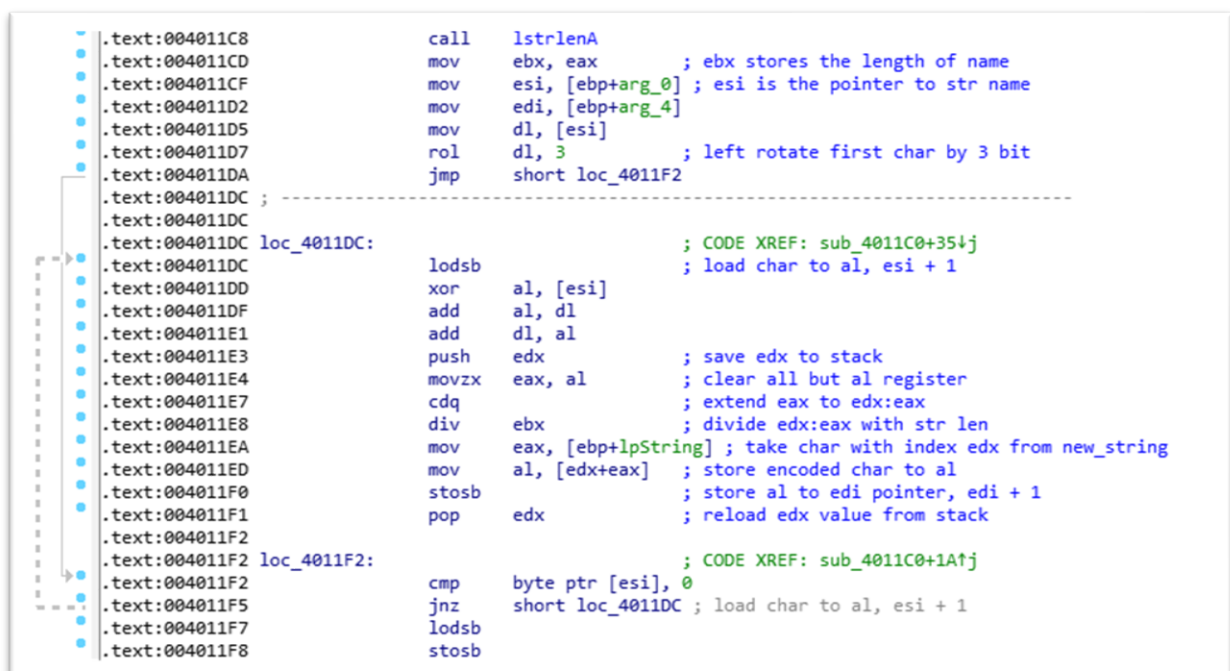
Nếu vượt quá thì sẽ trả về giá trị mặc định là 30.
VD: `abcd` → `strlen` = 4 → return 16

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 gọi hàm `encode`. Nội dung của hàm `encode` được chú thích như sau:



Hàm này sử dụng các phép tính để mã hóa các ký tự trong name thành 1 ký 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:

```
string encryptUsername(string username, int startPositionInBaseString)
{
    int DL = int(ROL(username[0], 3));
    string res = "";
    for (int i = 0; i < username.length(); i++)
    {
        int AL = username[i] ^ username[i + 1];
        AL = (AL + DL) & 0xFF;
        DL = (AL + DL) & 0xFF;
        int offset = AL % baselen;
        char ch = getCharFromBaseString(startPositionInBaseString, offset);
        res.push_back(ch);
    }
    return res;
}
```

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_encoded_name:

```
1 int __stdcall check_serial_and_encoded_name(char a1, char a2, char *a3)
2 {
3     char v4; // [esp+Ch] [ebp-88h]
4     int v5; // [esp+90h] [ebp-4h]
5
6     v5 = find_idx_of((int)a3, a1);
7     cut_and_cat(a3, &v4, v5);
8     return (unsigned __int8)a3[find_idx_of((int)&v4, a2)];
9 }
```

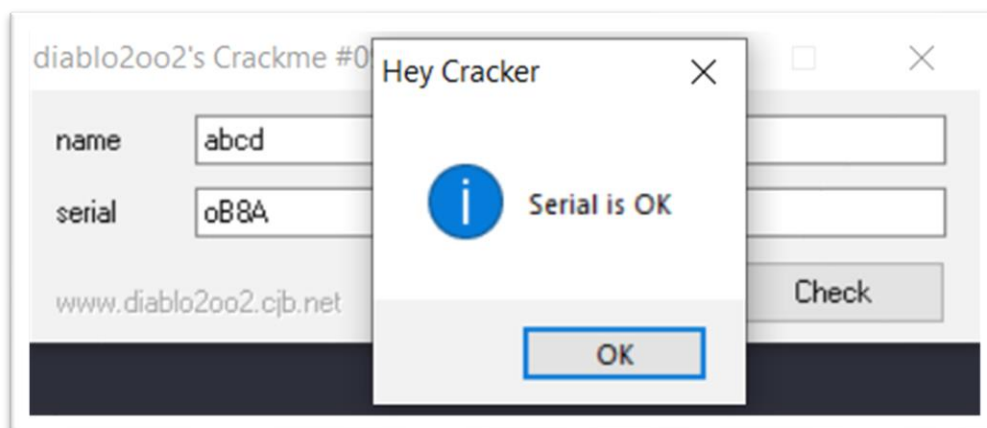
Vậy, bây giờ đơn giản chúng ta làm ngược lại các bước trên để tạo được keygen

```
string getSerial(string username, string encryptedUname, int startPosInBaseString)
{
    string res = "";
    for (int i = 0; i < username.length(); i++)
    {
        int offs = getOffsetFromPos(startPosInBaseString, username[i]);
        int pos2 = getOffsetFromPos(0, encryptedUname[i]);
        res.push_back(getCharFromBaseString(pos2, offs));
    }
    return res;
}
```

Chạy thử chương trình:

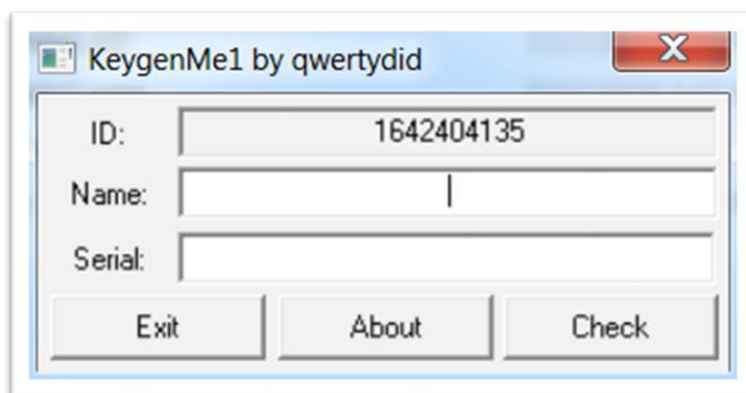
```
PS C:\Users\phant\Desktop> ^C
PS C:\Users\phant\Desktop> cd "c:\Users\phant\Desktop\"
Input username: abcd
Your username: abcd
Your Serial: oB8A
```

Vậy là đã hoàn thành !

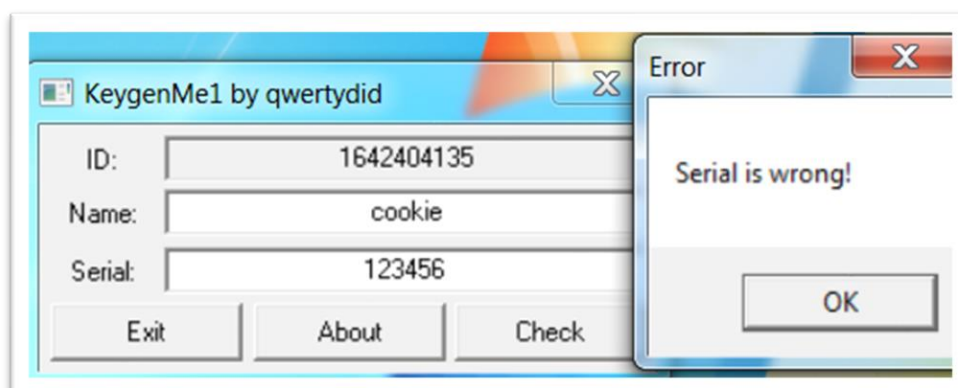


Solution KeygenMe1

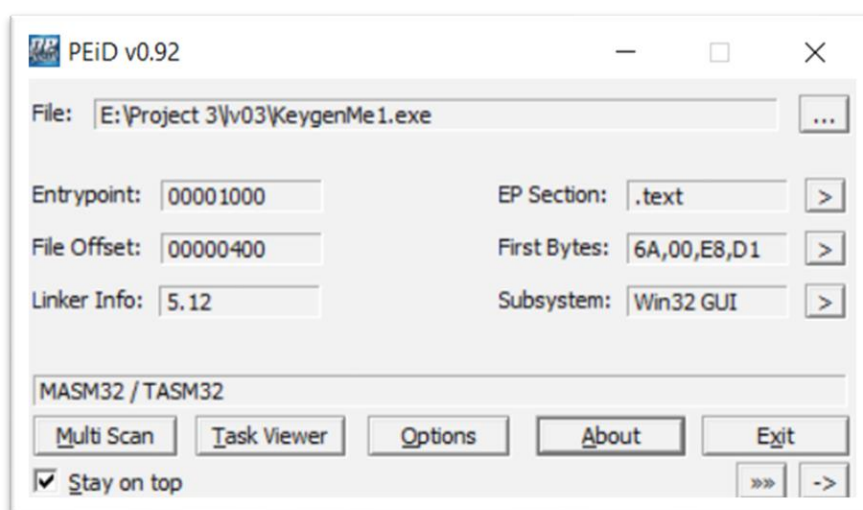
Mở chương trình lên:



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:

EIP EDX	→	00401000	6A 00	push 0	EntryPoint
		00401002	E8 D1130000	call <JMP.&GetModuleHandleA>	
		00401007	A3 80414000	mov dword ptr ds:[404180],eax	
		0040100C	6A 00	push 0	
		0040100E	68 2B104000	push keygenme1.40102B	
		00401013	6A 00	push 0	
		00401015	68 E9030000	push 3E9	
		0040101A	FF35 80414000	push dword ptr ds:[404180]	
		00401020	E8 BF130000	call <JMP.&DialogBoxParamA>	
		00401025	50	push eax	
		00401026	E8 A1130000	call <JMP.&ExitProcess>	
		0040102B	55	push ebp	

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.

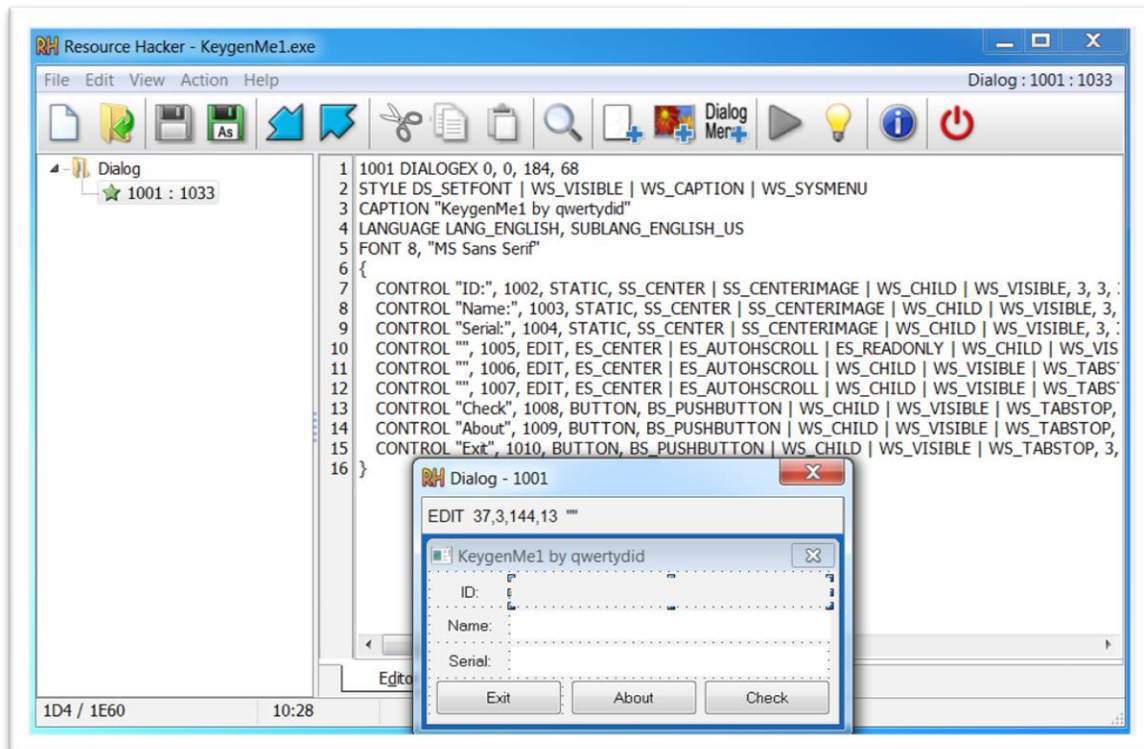
Ta có thể lập tức thấy ngay có 1 switch case xử lý dựa theo message a2 gồm có: WM_INITDIALOG (0x0110), WM_COMMAND (0x0111) và WM_CLOSE (0x0010). Tham khảo thêm tại: [link](#)

```
NT_PTR __stdcall DialogFunc(HWND hWnd, UINT a2, WPARAM a3, LPARAM a4)

HWND v4; // eax

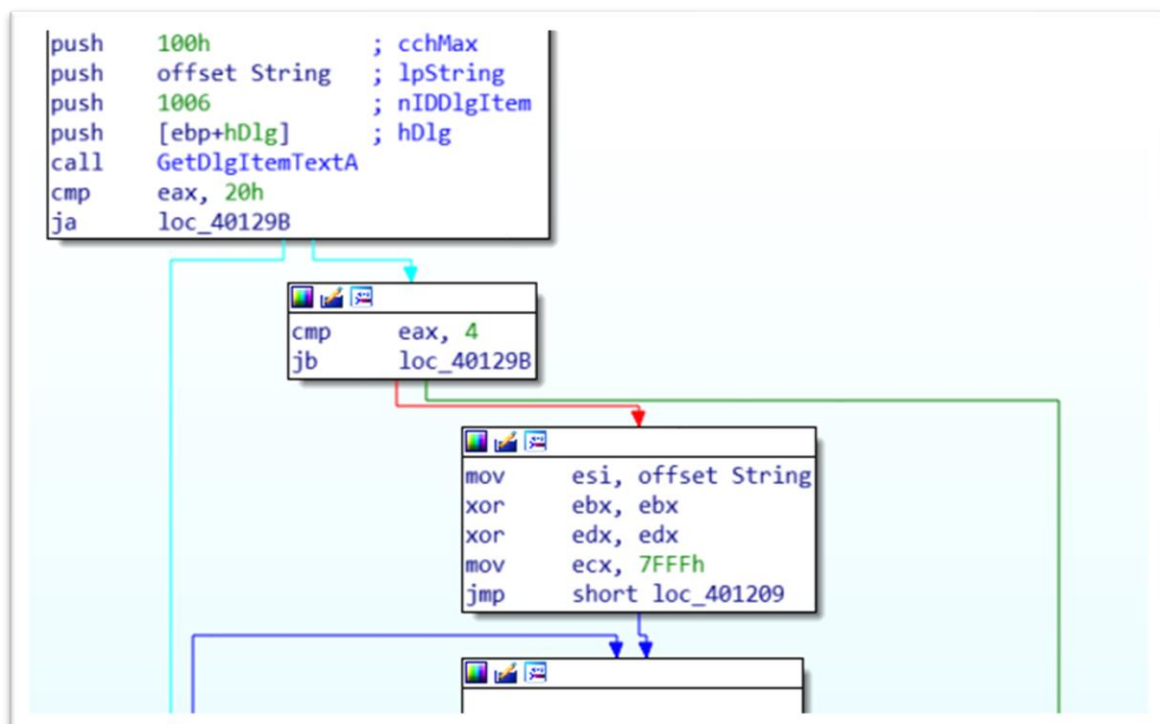
switch ( a2 )
{
    case 0x111u:
        switch ( a3 )
        {
            case 1008u:
                if ( sub_40119F(hWnd) )
                {
                    sub_401370(Caption, 15);
                    sub_401370(Text, 18);
                    MessageBoxA(hWnd, Text, Caption, 0);
                    sub_401370(Caption, 15);
                    sub_401370(Text, 18);
                }
                break;
            case 1009u:
                sub_401370(asc_404000, 5);
                sub_401370(&byte_404006, 268);
                MessageBoxA(hWnd, &byte_404006, asc_404000, 0);
                sub_401370(asc_404000, 5);
                sub_401370(&byte_404006, 268);
                break;
            case 1010u:
                EndDialog(hWnd, 0);
        }
    }
}
```


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 lý tiếp còn False thì sẽ không xử lý. Ta có thể 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 lý như sau:

```
loc_40129B:
push     5
push     offset byte_404113
call     sub_401370
push     0Eh
push     offset asc_404129 ; "f"
call     sub_401370
push     0 ; uType
push     offset byte_404113 ; lpCaption
push     offset asc_404129 ; "f"
push     [ebp+hDlg] ; hWnd
call     MessageBoxA
push     5
push     offset byte_404113
call     sub_401370
push     0Eh
push     offset asc_404129 ; "f"
call     sub_401370
xor      eax, eax
jmp      short locret_401329
```

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 lý 2 vùng nhớ này, ta có thể decompile để xem chức năng của hàm sub_401370.

```
void __stdcall sub_401370(int a1, unsigned int a2)
{
    unsigned int i; // ecx

    for ( i = 0; i < a2; ++i )
        *(_BYTE *)(i + a1) ^= 0xCDu;
}
```

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:

004011CB	BE 94414000	mov esi, keygenme1.404194
004011D0	33DB	xor ebx, ebx
004011D2	33D2	xor edx, edx
004011D4	B9 FF7F0000	mov ecx, 7FFF
004011D9	EB 2E	jmp keygenme1.401209
004011DB	66:8B1E	mov bx, word ptr ds:[esi]
004011DE	C1E3 08	shl ebx, 8
004011E1	A1 C4424000	mov eax, dword ptr ds:[4042C4]
004011E6	25 00F8F800	and eax, F8F800
004011EB	33D8	xor ebx, eax
004011ED	81C3 6C6F6C00	add ebx, 6C6F6C
004011F3	81F3 10101010	xor ebx, 10101010
004011F9	03D3	add edx, ebx
004011FB	03CB	add ecx, ebx
004011FD	81E9 2D3D2D00	sub ecx, 2D3D2D
00401203	6BC9 08	imul ecx, ecx, 8
00401206	03C8	add ecx, eax
00401208	46	inc esi
00401209	803E 00	cmp byte ptr ds:[esi], 0
0040120C	75 CD	jne keygenme1.4011DB

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 lý đoạn code sau:

```

push    13h
push    offset aEyo      ; "ềỹồ"
call    sub_401370
push    esi
push    edi
push    edx
push    ecx
push    offset aEyo      ; "ềỹồ"
push    offset byte_4042CC ; LPSTR
call    wsprintfA
add     esp, 18h
push    13h
push    offset aEyo      ; "ềỹồ"
call    sub_401370
push    30h              ; cchMax
push    offset byte_404294 ; lpString
push    3EFh             ; nIDDlgItem
push    [ebp+hDlg]        ; hDlg
call    GetDlgItemTextA
cmp     eax, 23h
jnz     short loc_4012E3
    
```

Hàm `wsprintfA` (tham khảo [link](#)) 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:

```
string getSerial(unsigned int id, string username)
{
    int ebx = 0;
    int ecx = 0x00007FFF;
    int edx = 0;
    int eax = id;
    for (int i = 0; i < username.length(); i++)
    {
        ebx = (ebx & 0xFFFF0000) | ((int(username[i + 1]) & 0xFF) << 8) | (username[i] & 0xFF);
        ebx = ebx << 8;
        eax = id;
        eax = eax & 0x00F8F800;
        ebx = ((ebx ^ eax) + 0x6C6F6C) ^ 0x10101010;
        edx += ebx;
        ecx = (ecx + ebx - 0x2D3D2D) * 8 + eax;
    }
    int esi = 0;
    int edi = 0;
    for (int i = 16; i > 0; --i)
    {
        esi += edx;
        edi += ecx;
        edi = _byteswap_ulong(edi);
        esi = _byteswap_ulong(esi);
        edi = ROL(edi, 16);
        esi = ROR(esi, 16);
    }
    char buff[36];
    sprintf_s(buff, "%08X-%08X-%08X-%08X", ecx, edx, edi, esi);
    return string(buff);
}
```

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.


```

sub_40132D proc near
push    offset nSize    ; nSize
push    offset Buffer    ; lpBuffer
call    GetComputerNameA
mov     esi, offset Buffer
xor     eax, eax
xor     edx, edx
xor     ebx, ebx
xor     ecx, ecx
jmp     short loc_401363

loc_401363:
cmp     word ptr [esi], 0
jnz     short loc_40134B

loc_40134B:
mov     al, [esi]
mov     dl, [esi+1]
ror     al, 4
not     dl
add     al, dl
add     ebx, eax
imul    edx, eax

bswap   ebx
add     ebx, ecx
mov     eax, ebx
retn
sub_40132D endp
        
```

```

unsigned int getComputerID()
{
    char computername[17];
    DWORD nSize = 16;
    GetComputerNameA(computername, &nSize);
    int i = 0;
    int eax = 0;
    int ebx = 0;
    int ecx = 0;
    int edx = 0;
    while (computername[i] && computername[i] != -52)
    {
        char dl = ~computername[i + 1];
        char al = ROR(computername[i], 4);
        al += dl;
        eax = (eax & 0xFFFFFFFF00) | (al & 0xFF);
        edx = (edx & 0xFFFFFFFF00) | (dl & 0xFF);
        ebx += eax;
        edx *= eax;
        ecx += edx;
        swap(ebx, edx);
        i += 2;
    }
    return _byteswap_ulong(ebx) + ecx;
}
        
```

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

```

E:\Project 3\lv03\Keygenme1\Keygenme1_Keygen\Debug\Keygenme1_Keygen.exe
Input username: cookie
Your serial: B408F438-EE7346E8-E8E66163-0D0E7777
        
```

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

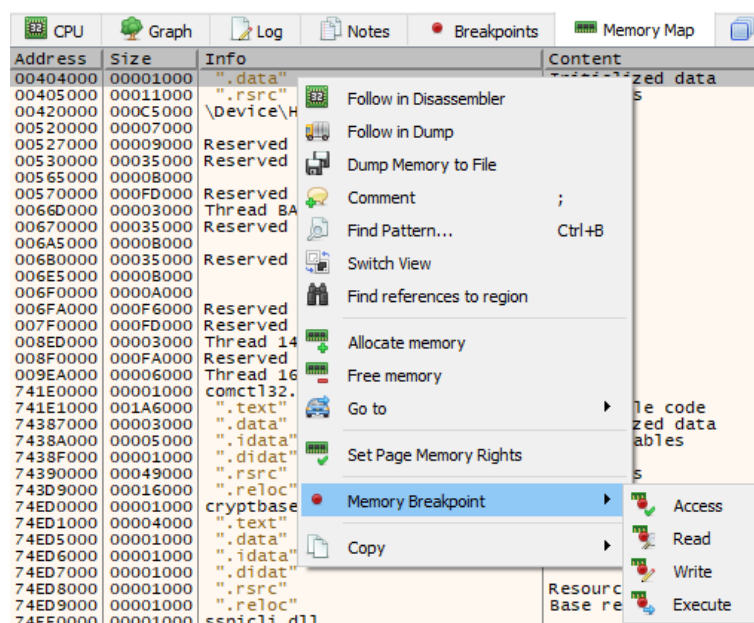
Address	Hex	ASCII
00404000	80 B8 BF B5 BE B9 A8 E3 D0 93 BF A5 A2 B9 B5 A2	..µ%''ãD.¿%¢'µ¢
00404010	F0 9E B5 A7 D0 87 B5 BC BC F0 94 BF BE B5 FC F0	ò.µ\$D.µ%ãD.¿%µüD
00404020	82 B5 A6 B5 A2 A3 B5 A2 F1 F1 F1 D0 99 A4 F0 A3	.µ'µ¢£µ¢ñññD.µD£
00404030	B5 B5 BD A3 F0 A4 B8 B1 A4 F0 A9 BF A5 F0 B8 B1	µµ%£Dµ.±µD@¿%D.±
00404040	A6 B5 F0 A4 BF F0 A7 BF A2 B8 F0 B8 B1 A2 B4 B5	'µDµ¿D\$¿¢%D.±¢'µ
00404050	A2 FC F0 BD B1 A4 B5 F1 F1 F1 D0 87 91 82 9E 99	¢üD%±µµñññD.....
00404060	9E 97 EA F0 84 B8 B5 F0 B1 A0 A0 BC B9 B3 B1 A4	..êD. µD± %'±µ
00404070	B9 BF BE F0 B9 A3 F0 B3 BF A2 A2 A5 A0 A4 B5 B4	'¿D'£D'¿¢¢% µµ'
00404080	F0 B1 BE B4 F0 A4 B8 B5 F0 B5 A8 B5 B3 A5 A4 B9	D±%'Dµ.µDµ'µ*µµ'
00404090	BF BE F0 A7 B9 BC BC F0 B2 B5 F0 A4 B5 A2 BD B9	¿D\$'¿%ãD=µDµµ¢%'
004040A0	BE B1 A4 B5 B4 F0 BE BF A7 F1 D0 00 01 00 08 00	%±µµ' D%¿\$ñD.....

Hình 1

Address	Hex	ASCII
00404000	50 68 6F 65 6E 69 78 33 00 43 6F 75 72 69 65 72	Phoenix3.Courier
00404010	20 4E 65 77 00 57 65 6C 6C 20 44 6F 6E 65 2C 20	New.Well Done,
00404020	52 65 76 65 72 73 65 72 21 21 21 00 49 74 20 73	Reverser!!!.It s
00404030	65 65 6D 73 20 74 68 61 74 20 79 6F 75 20 68 61	eems that you ha
00404040	76 65 20 74 6F 20 77 6F 72 68 20 68 61 72 64 65	ve to work harde
00404050	72 2C 20 6D 61 74 65 21 21 21 00 57 41 52 4E 49	r, mate!!!.WARNI
00404060	4E 47 3A 20 54 68 65 20 61 70 70 6C 69 63 61 74	NG: The applicat
00404070	69 6F 6E 20 69 73 20 63 6F 72 72 75 70 74 65 64	ion is corrupted
00404080	20 61 6E 64 20 74 68 65 20 65 78 65 63 75 74 69	and the executi
00404090	6F 6E 20 77 69 6C 6C 20 62 65 20 74 65 72 6D 69	on will be termi
004040A0	6E 61 74 65 64 20 6E 6F 77 21 00 00 01 00 08 00	nated now!.....

Hình 2

Trong hình 2 data kết thúc tại dấu ! (0x21). Byte kế 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:

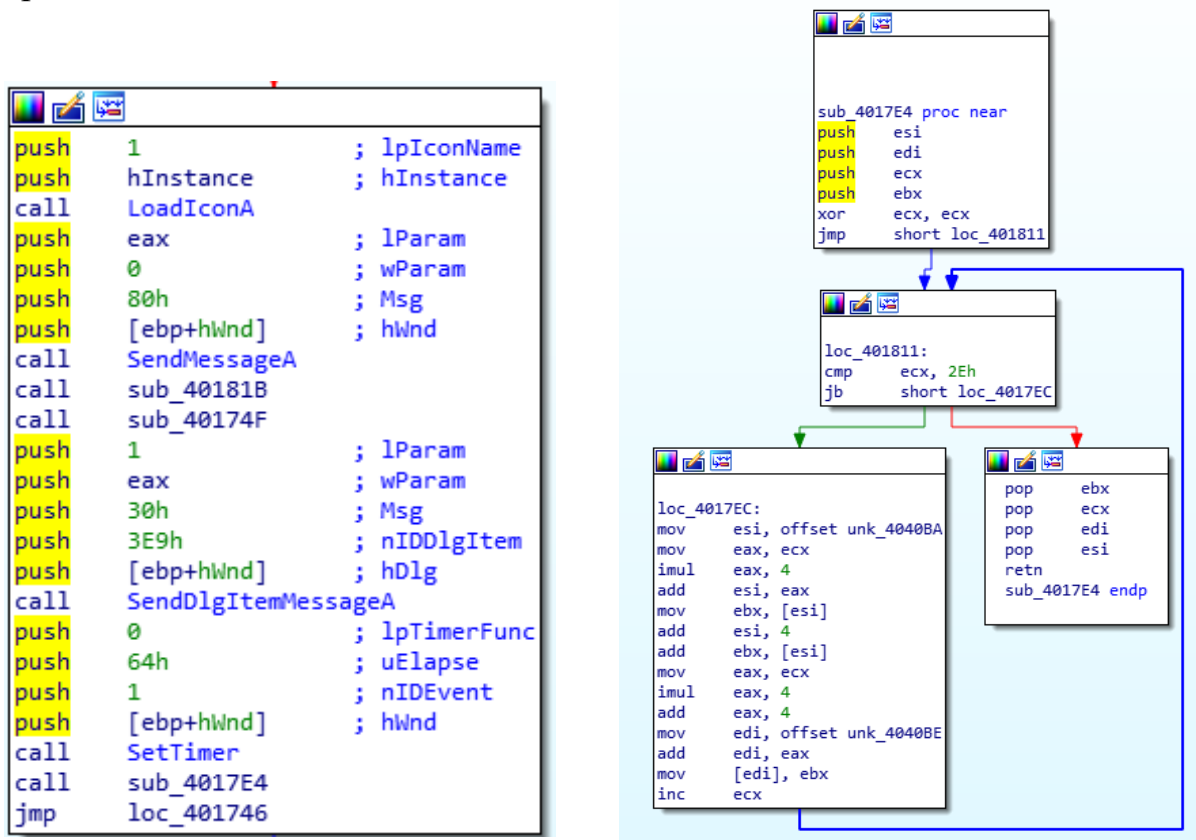
1. Gọi hàm này là **decrypt_data**, địa chỉ gọi đến là **40181B**
2. Thanh ghi ESI trỏ đến địa chỉ .data, thanh ghi EDI trỏ đến vị trí byte D0
3. Vòng lặp thực hiện xor từng byte mà ESI trỏ đến với D0. Nếu ESI = EDI, nghĩa là byte tại EDI là D0 được xor với D0 sẽ cho giá trị 0, trùng với giá trị byte 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

00401818	BE 00404000	mov esi,tryyourkeygen.404000
00401820	BF AA404000	mov edi,tryyourkeygen.4040AA
00401825	EB 04	jmp tryyourkeygen.401828
00401827	8036 D0	xor byte ptr ds:[esi],D0
0040182A	46	inc esi
0040182B	3BF7	cmp esi,edi
0040182D	76 F8	jbe tryyourkeygen.401827
0040182F	C3	ret

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

004015A4	E8 6D030000	call <JMP.&InitCommonControls>	EntryPoint
004015A9	6A 00	push 0	
004015AB	E8 FA020000	call <JMP.&GetModuleHandleA>	
004015B0	A3 64424000	mov dword ptr ds:[404264],eax	
004015B5	6A 00	push 0	
004015B7	68 D4154000	push tryyourkeygen.4015D4	
004015BC	6A 00	push 0	
004015BE	68 E8030000	push 3E8	
004015C3	FF35 64424000	push dword ptr ds:[404264]	
004015C9	E8 F4020000	call <JMP.&DialogBoxParamA>	
004015CE	50	push eax	
004015CF	E8 CA020000	call <JMP.&ExitProcess>	
004015D4	55	push ebp	
004015D5	8BEC	mov ebp,esp	
004015D7	817D 0C 10010000	cmp dword ptr ss:[ebp+C],110	
004015DE	75 51	jne tryyourkeygen.401631	
004015E0	6A 01	push 1	
004015E2	FF35 64424000	push dword ptr ds:[404264]	
004015E8	E8 F3020000	call <JMP.&LoadIconA>	
004015ED	50	push eax	
004015EE	6A 00	push 0	
004015F0	68 80000000	push 80	
004015F5	FF75 08	push dword ptr ss:[ebp+8]	
004015F8	E8 F5020000	call <JMP.&SendMessageA>	
004015FD	E8 19020000	call tryyourkeygen.40181B	
00401602	E8 48010000	call tryyourkeygen.40174F	
00401607	6A 01	push 1	
00401609	50	push eax	

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

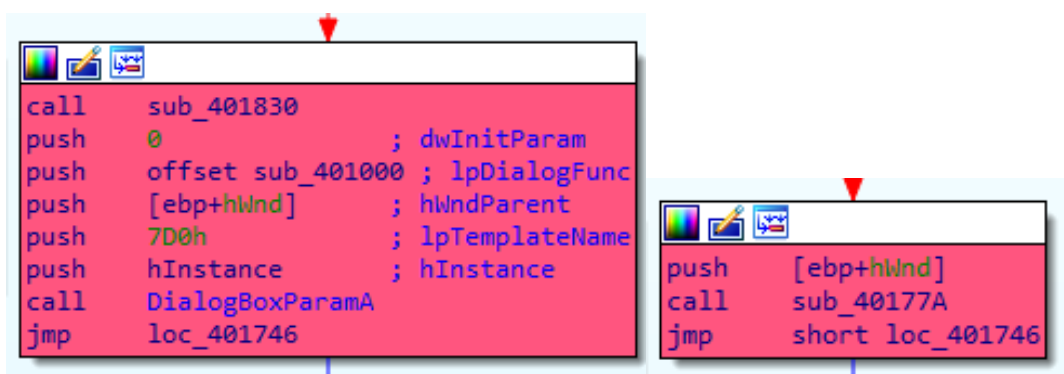


Ngoài việc gọi đến **decrypt_data (40181B)**, hàm này cũng gọi đến các địa chỉ **40174F (1)** và **4017E4 (2)**.

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

```

00401000  55          push ebp
00401001  8BEC        mov ebp,esp
00401003  837D 0C 10  cmp dword ptr ss:[ebp+C],10
00401007  75 0F       jne tryyourkeygen.401018
00401009  6A 00       push 0
0040100B  FF75 08     push dword ptr ss:[ebp+8]
0040100E  E8 B5080000 call <JMP.&EndDialog>
00401013  E9 99000000 jmp tryyourkeygen.401081
00401018  817D 0C 11010000 cmp dword ptr ss:[ebp+C],111
0040101F  0F85 86000000 jne tryyourkeygen.4010AB
00401025  817D 10 D4070000 cmp dword ptr ss:[ebp+10],7D4
0040102C  75 64       jne tryyourkeygen.401092
0040102E  BE 42104000 mov esi,tryyourkeygen.401042
00401033  BF 7C104000 mov edi,tryyourkeygen.40107C
00401038  EB 04       jmp tryyourkeygen.40103E
0040103A  8036 D0     xor byte ptr ds:[esi],D0
0040103D  46         inc esi
0040103E  3BF7       cmp esi,edi
00401040  76 F8       jbe tryyourkeygen.40103A
00401042  2F         das
00401043  A5         movsd
00401044  D838       fdivr st(0),dword ptr ds:[eax]
    
```

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 trở nên 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:

```

00401042  FF75 08     push dword ptr ss:[ebp+8]
00401045  E8 A6000000 call v5_phoenix3.4010F0
0040104A  85C0        test eax,eax
0040104C  74 1A       je v5_phoenix3.401068
0040104E  FF75 08     push dword ptr ss:[ebp+8]
00401051  E8 C7010000 call v5_phoenix3.40121D
00401056  85C0        test eax,eax
00401058  74 0E       je v5_phoenix3.401068
0040105A  6A 40       push 40
0040105C  68 00404000 push v5_phoenix3.404000
00401061  68 15404000 push v5_phoenix3.404015
00401066  EB 0C       jmp v5_phoenix3.401074
00401068  6A 10       push 10
0040106A  68 00404000 push v5_phoenix3.404000
0040106F  68 2C404000 push v5_phoenix3.40402C
00401074  FF75 08     push dword ptr ss:[ebp+8]
00401077  E8 6A080000 call <JMP.&MessageBoxA>
    
```


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.

	<p>Đầu tiên lấy độ dài chuỗi serial và so sánh với 0x1D (độ dài 29).</p> <p>Hàm 4012DD kiểm tra serial chỉ chứa số và dấu – (0x2D)</p> <p>Loop kiểm tra serial có bao gồm 5 bộ 5 chữ số cách nhau dấu – hay không.</p> <p>Hàm 4017E4 tạo sẵn 47 số đầu tiên của dãy fibo trong .data</p> <p>esi giữ địa chỉ name edi giữ địa chỉ giá trị 0x0159A3</p> <p>ebx lưu mã hóa của name</p> <p>mov ebx vào stack (1) edi giữ địa chỉ fibo[21] eax lưu giá trị số fibo</p> <p>esi trở đến 5 digit tiếp theo thuộc serial Hàm 402A46 lấy giá trị số nguyên của esi</p> <p>[ebp-8] là giá trị từ (1)</p>
<pre> 0040110A 6A 1F push 1F 0040110C 68 A6444000 push v5_phoenix3.4044A6 00401111 68 D2070000 push 7D2 00401116 FF75 08 push dword ptr ss:[ebp+8] 00401119 E8 B6070000 call <JMP.&GetDlgItemTextA> 0040111E 83F8 1D cmp eax,1D 00401121 74 07 je v5_phoenix3.40112A 00401123 33C0 xor eax,eax 00401125 E9 D8000000 jmp v5_phoenix3.401205 0040112A E8 AE010000 call v5_phoenix3.4012DD 0040112F 08C0 or eax,eax 00401131 75 05 jne v5_phoenix3.401138 00401133 E9 CD000000 jmp v5_phoenix3.401205 00401138 33C9 xor ecx,ecx 0040113A BE A6444000 mov esi,v5_phoenix3.4044A6 0040113F 4E dec esi 00401140 EB 14 jmp v5_phoenix3.401156 00401142 83C6 06 add esi,6 00401145 A006 mov al,byte ptr ds:[esi] 00401147 3C 2D cmp al,2D 00401149 74 07 je v5_phoenix3.401152 0040114B 33C0 xor eax,eax 0040114D E9 B3000000 jmp v5_phoenix3.401205 00401152 C606 00 mov byte ptr ds:[esi],0 00401155 41 inc ecx 00401156 83F9 04 cmp ecx,4 00401159 72 E7 jb v5_phoenix3.401142 </pre>	<pre> 00401158 E8 84060000 call v5_phoenix3.4017E4 00401160 6A 20 push 20 00401162 68 A6434000 push v5_phoenix3.4043A6 00401167 68 D1070000 push 7D1 0040116C FF75 08 push dword ptr ss:[ebp+8] 0040116F E8 60070000 call <JMP.&GetDlgItemTextA> 00401174 8945 FC mov dword ptr ss:[ebp-4],eax 00401177 33C9 xor ecx,ecx 00401179 33D2 xor edx,edx 0040117B BB DEC0ADDE mov ebx,DEADCODE 00401180 BE A6434000 mov esi,v5_phoenix3.4043A6 00401185 BF 10014000 mov edi,v5_phoenix3.400110 0040118A EB 22 jmp v5_phoenix3.4011AE 0040118C 8A1431 mov dl,byte ptr ds:[ecx+esi] 0040118F 8AF2 mov dh,dl 00401191 66:88C2 mov ax,dx 00401194 C1E2 10 shl edx,10 00401197 66:88D0 mov dx,ax 0040119A 33DA xor ebx,edx 0040119C 81E3 D0D0FEFE and ebx,FEFED0D0 004011A2 081F or ebx,dword ptr ds:[edi] 004011A4 66:88C3 mov ax,bx 004011A7 C1EB 10 shr ebx,10 004011AA 66:33D8 xor bx,ax 004011AD 41 inc ecx 004011AE 384D FC cmp ecx,dword ptr ss:[ebp-4] 004011B1 72 D9 jb v5_phoenix3.40118C 004011B3 81E3 FFFF0000 and ebx,FFFF 004011B9 895D F8 mov dword ptr ss:[ebp-8],ebx 004011BC 8D3D 0E414000 lea edi,dword ptr ds:[40410E] 004011C7 33C9 xor ecx,ecx 004011C9 EB 28 jmp v5_phoenix3.4011F3 004011CB 8B048F mov eax,dword ptr ds:[edi+ecx*4] 004011CE 33C3 xor eax,ebx 004011D0 50 push eax 004011D1 8DB489 A6444000 lea esi,dword ptr ds:[ecx+ecx*4+4044A6] 004011D8 03F1 add esi,ecx 004011DA 56 push esi 004011DB E8 66180000 call v5_phoenix3.402A46 004011E0 5A pop edx 004011E1 81E2 FFFF0000 and edx,FFFF 004011E7 38C2 cmp eax,edx 004011E9 74 04 je v5_phoenix3.4011EF 004011EB 33C0 xor eax,eax 004011ED EB 16 jmp v5_phoenix3.401205 004011EF 3145 F8 xor dword ptr ss:[ebp-8],eax 004011F2 41 inc ecx 004011F3 83F9 05 cmp ecx,5 004011F6 72 D3 jb v5_phoenix3.4011CB 004011F8 FF75 F8 push dword ptr ss:[ebp-8] 004011FB E8 BAFEFFFF call v5_phoenix3.4010BA 00401200 B8 01000000 mov eax,1 00401205 90 nop </pre>

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 (roi 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 roi vào goodboy. Hàm này nhận tham số đầu tiên trên stack là giá trị tại $[EBP - 8]$

•	0040108A	55	push ebp
•	0040108B	8BEC	mov ebp,esp
•	0040108D	51	push ecx
•	0040108E	53	push ebx
•	0040108F	57	push edi
•	004010C0	33C9	xor ecx,ecx
•	004010C2	BF A6424000	mov edi,v5_phoenix3.4042A6
•	004010C7	8B45 08	mov eax,dword ptr ss:[ebp+8]
•	004010CA	8AD8	mov bl,al
•	004010CC	22DC	and bl,ah
•	004010CE	C1E8 10	shr eax,10
•	004010D1	32D8	xor bl,al
•	004010D3	02DC	add bl,ah
•	004010D5	✓ EB 0A	jmp v5_phoenix3.4010E1
•	004010D7	8AF9	mov bh,cl
•	004010D9	32FB	xor bh,bl
•	004010DB	883F	mov byte ptr ds:[edi],bh
•	004010DD	41	inc ecx
•	004010DE	47	inc edi
•	004010DF	D0C3	rol bl,1
•	004010E1	81F9 00010000	cmp ecx,100
•	004010E7	^ 72 EE	jnb v5_phoenix3.4010D7
•	004010E9	5F	pop edi
•	004010EA	5B	pop ebx
•	004010EB	59	pop ecx
•	004010EC	C9	leave
•	004010ED	C2 0400	ret 4
•	004010F0	55	push ebp
•	004010F1	8BEC	mov ebp,esp
•	004010F3	83C4 F8	add esp,FFFFFFF8

edi trở 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 ...

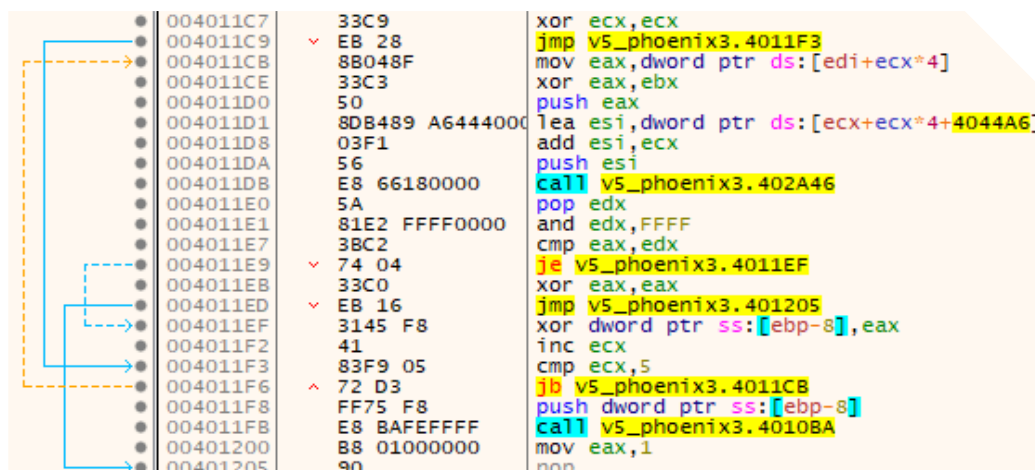
Câu lệnh (!!!) shift right 16 bit. Trong khi đó EAX chỉ có bật 4 byte cuối được bật.

Như vậy câu 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. $Q[1] = Q[0] \wedge \text{fibo}[21] \wedge Q[0]$
2. $Q[2] = Q[1] \wedge \text{fibo}[22] \wedge Q[0]$
3. $Q[3] = Q[2] \wedge \text{fibo}[23] \wedge Q[0]$
4. $Q[4] = Q[3] \wedge \text{fibo}[24] \wedge Q[0]$
5. $Q[5] = Q[4] \wedge \text{fibo}[25] \wedge Q[0]$

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 $f[21] \wedge f[22] \wedge f[23] \wedge f[24] \wedge f[25]$

Nhưng mà bất ngờ hơn là kết quả của 1 đồng 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 trở 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.

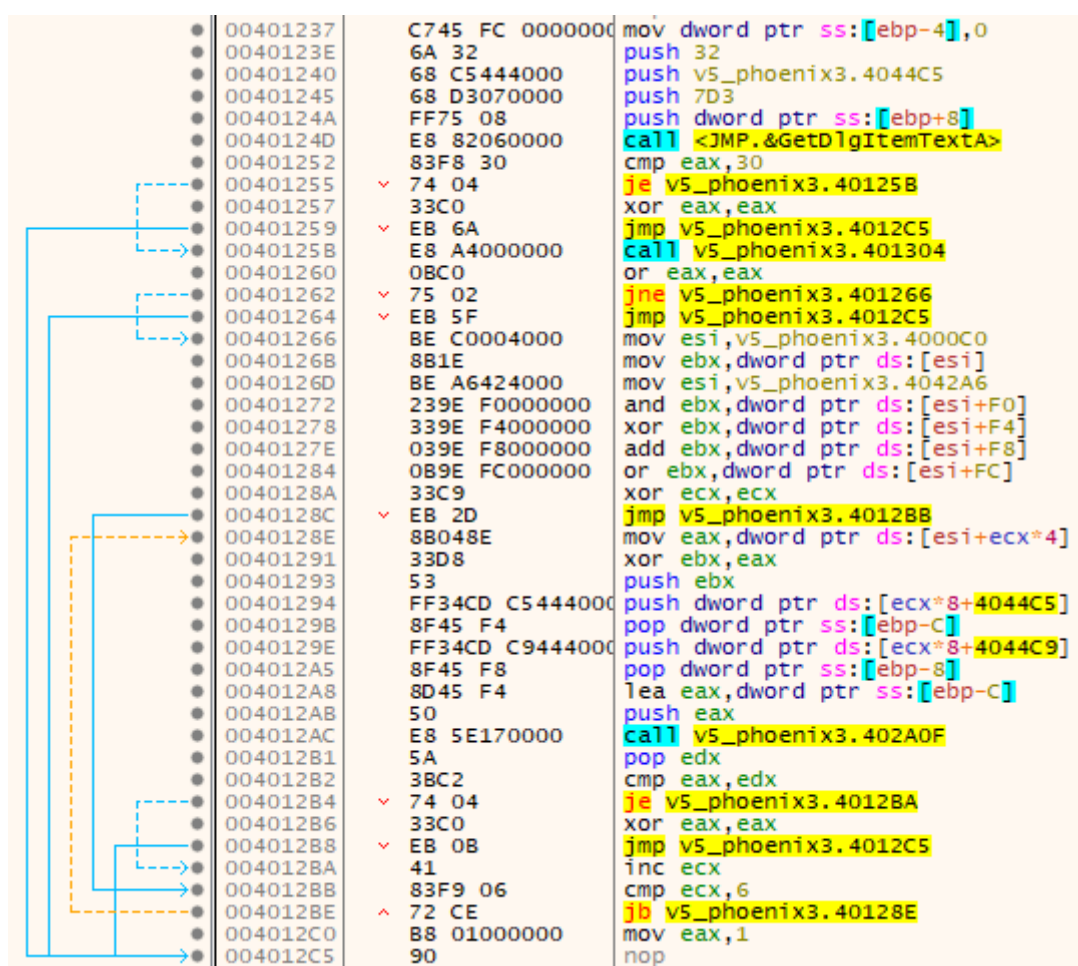
Đầu tiên hàm sẽ lấy chuỗi activation code và lưu tại địa chỉ 4044C5. Ta cần đảm bảo độ dài chuỗi là 48 (0x30) ký tự.

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 0xFFFFF4.



Address	Disassembly	Comment
00401237	C745 FC 00000000	mov dword ptr ss:[ebp-4],0
0040123E	6A 32	push 32
00401240	68 C5444000	push v5_phoenix3.4044C5
00401245	68 D3070000	push 7D3
0040124A	FF75 08	push dword ptr ss:[ebp+8]
0040124D	E8 82060000	call <JMP.&GetDlgItemTextA>
00401252	83F8 30	cmp eax,30
00401255	74 04	je v5_phoenix3.40125B
00401257	33C0	xor eax,eax
00401259	EB 6A	jmp v5_phoenix3.4012C5
0040125B	E8 A4000000	call v5_phoenix3.401304
00401260	0BC0	or eax,eax
00401262	75 02	jne v5_phoenix3.401266
00401264	EB 5F	jmp v5_phoenix3.4012C5
00401266	BE C0040000	mov esi,v5_phoenix3.4000C0
0040126B	8B1E	mov ebx,dword ptr ds:[esi]
0040126D	BE A6424000	mov esi,v5_phoenix3.4042A6
00401272	239E F0000000	and ebx,dword ptr ds:[esi+F0]
00401278	339E F4000000	xor ebx,dword ptr ds:[esi+F4]
0040127E	039E F8000000	add ebx,dword ptr ds:[esi+F8]
00401284	0B9E FC000000	or ebx,dword ptr ds:[esi+FC]
0040128A	33C9	xor ecx,ecx
0040128C	EB 2D	jmp v5_phoenix3.40128B
0040128E	8B048E	mov eax,dword ptr ds:[esi+ecx*4]
00401291	33D8	xor ebx,eax
00401293	53	push ebx
00401294	FF34CD C5444000	push dword ptr ds:[ecx*8+4044C5]
00401298	8F45 F4	pop dword ptr ss:[ebp-C]
0040129E	FF34CD C9444000	push dword ptr ds:[ecx*8+4044C9]
004012A5	8F45 F8	pop dword ptr ss:[ebp-8]
004012A8	8D45 F4	lea eax,dword ptr ss:[ebp-C]
004012AB	50	push eax
004012AC	E8 5E170000	call v5_phoenix3.402A0F
004012B1	5A	pop edx
004012B2	3BC2	cmp eax,edx
004012B4	74 04	je v5_phoenix3.4012BA
004012B6	33C0	xor eax,eax
004012B8	EB 0B	jmp v5_phoenix3.4012C5
004012BA	41	inc ecx
004012BB	83F9 06	cmp ecx,6
004012BE	72 CE	jb v5_phoenix3.40128E
004012C0	B8 01000000	mov eax,1
004012C5	90	nop

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 thước)

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:

FCFD FEFC FBFB FBFB F8F0 F1F2 F0FFFF FFFFC ECEDE EEECFB FBFB FBFB F8

Code tìm serial như sau:

```
print('Input Name:', end = ' ')
name = input()
ebx = 0xdeadC0de
```

```
for ch in name:
    edx = ord(ch)
    edx = (edx << 8) + edx
    edx = (edx << 16) + edx
    ebx = ebx ^ edx
    ebx = ebx & 0xfefed0d0
    ebx = ebx | 0x000159a3
    eax = ebx & 0x0000ffff
    ebx = ebx >> 16
    ebx = ebx ^ eax
```

```
ebx = ebx & 0x0000ffff
sta = ebx
```

```
ebx = ebx & 0x0000ffff
sta = ebx
```

```
fibo = [10946, 17711, 28657, 46368, 75025]
res = ''
```

```
for eax in fibo:
    eax = eax ^ ebx
    eax = eax & 0x0000ffff
    sta = sta ^ eax
    eax = str(eax)
    while len(eax) < 5:
        eax = '0' + eax
    res += eax + '-'
```

```
print('Serial:', res[:-1])
```

