

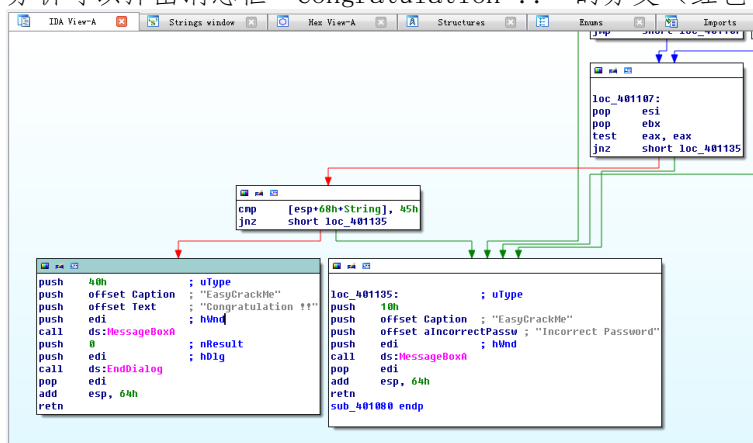
reversing.kr-逆向挑战_笔记

目录

1	Easy CrackMe	2
2	Easy UnpackMe	5
3	Easy Keygen	6
4	Easy ELF	9
5	Position	12
6	Replace	17

1 Easy CrackMe

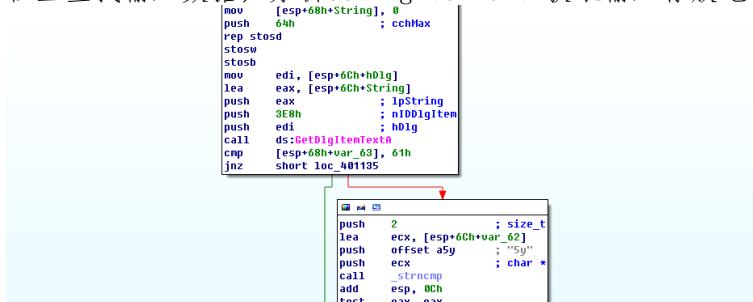
1. 运行程序，随意输入，弹窗提示：“Incorrect Password”。
2. IDA打开查看函数列表确认没有加壳，直接静态分析。
3. (a) Shift + F12 打开Strings窗口，找到字符串“Incorrect Password”，
(b) 双击 转到字符串定义，
(c) 双击 交叉引用（DATA XREF）的函数名，跳转到对应函数定义，
(d) 空格 切换为Graph View分析函数功能
4. 分析可以弹出消息框“Congratulation !!”的分支（红色箭头）：



[esp+68h+String]期望值为E（45h是字符E的ASCII值）

```
cmp     [esp+68h+String], 45h
jnz     short loc_401135
```

往上查找输入数据，分析GetDlgItemTextA获取输入存放地址：



[esp+68h+String]为存放输入的地址（注意：push 64h ; cchMax影响esp的值）

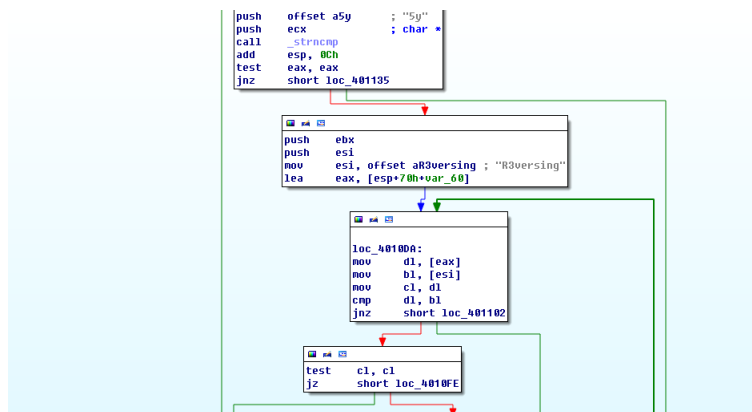
```
lea     eax, [esp+6Ch+String]
push    eax                ; lpString
```

继续分析，比较输入和期望值 第一步：[esp+68h+var_63]期望值为a（61h是字符a的ASCII值）

```
cmp     [esp+68h+var_63], 61h
jnz     short loc_401135
```

第二步：[esp+6Ch+var_62]期望值为“5y”（注意：push 2 ; size_t影响esp的值）

```
lea     ecx, [esp+6Ch+var_62]
push    offset a5y        ; "5y"
push    ecx                ; char *
call    _strncmp
```



第三步：[esp+70h+var_60]期望值为“R3versing”（注意：push ebx、push esi影响esp的值）
循环不是很难分析，期望的分支是

```
xor     eax, eax
jmp     short loc_401107
```

而非

```
sbb     eax, eax
sbb     eax, 0FFFFFFFFh
```

```
; int __cdecl sub_401080(HWND hDlg)
sub_401080 proc near

String= byte ptr -64h
var_63= byte ptr -63h
var_62= byte ptr -62h
var_60= byte ptr -60h
hDlg= dword ptr 4
```

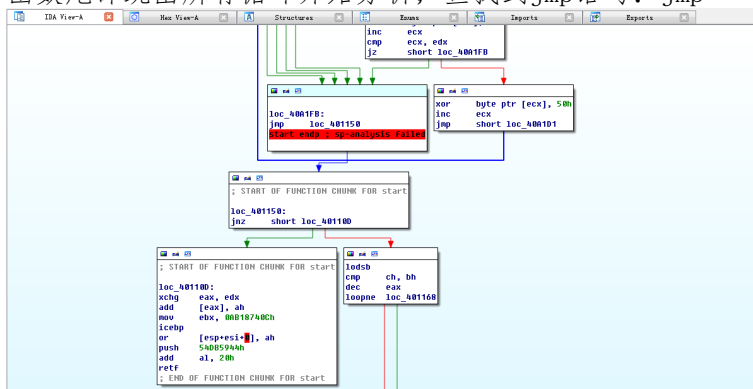
最后，根据变量定义，将4部分期待字符（串）连接在一起：

```
String= byte ptr -64h    ;    'E'
var_63= byte ptr -63     ;    'a'
var_62= byte ptr -62h    ;    '5y'
var_60= byte ptr -60h    ;    'R3versing'

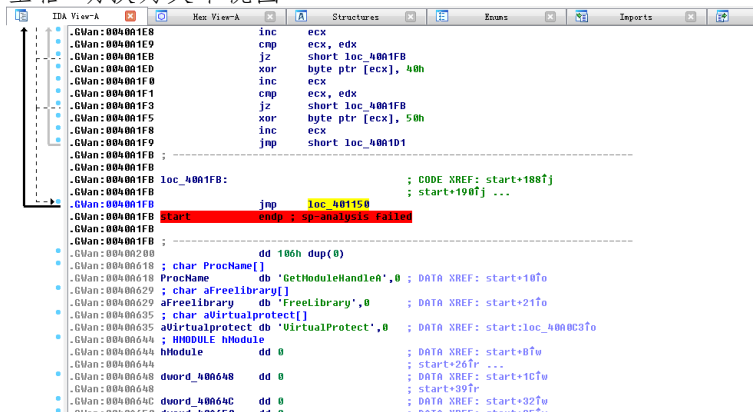
flag :  "Ea5yR3versing"
```

2 Easy UnpackMe

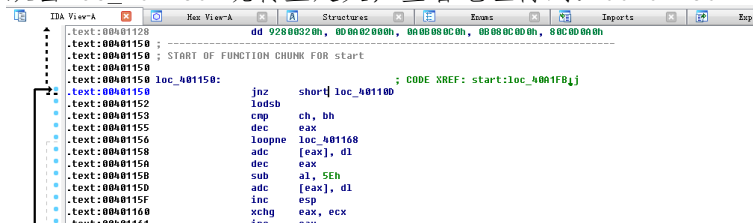
1. IDA直接打开，查看函数列表，只有一个start函数
2. 双击 查看函数 ，开始分析
3. 空格 切换为Graph视图
4. 函数尾部跳出所有循环开始分析，查找到jmp语句：jmp loc_401150



空格 切换为文本视图



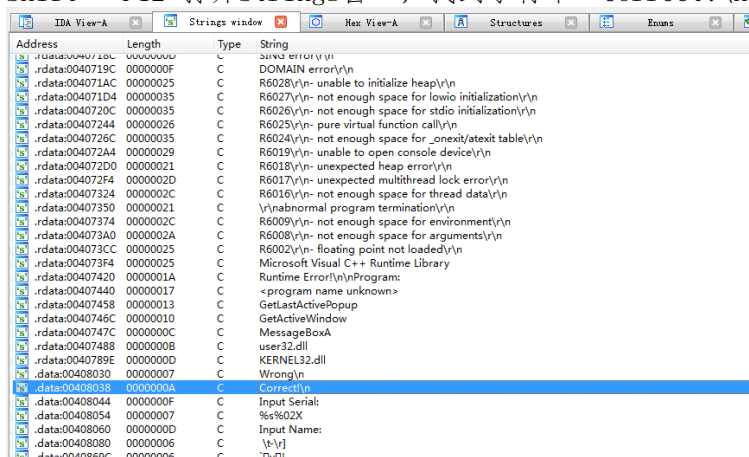
双击loc_401150 跳转至定义，查看地址得到：00401150



flag : “00401150”

3 Easy Keygen

1. 运行程序，随意输入，程序结束。
2. IDA打开查看函数列表确认没有加壳，直接静态分析。
3. (a) Shift + F12 打开Strings窗口，找到字符串“Correct!\n”，

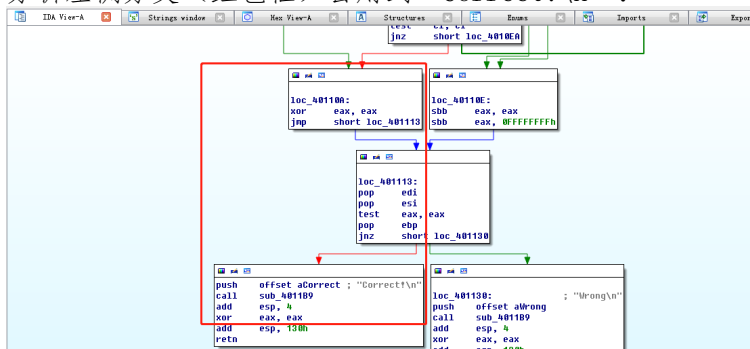


(b) 双击 转到字符串定义，

(c) 双击 交叉引用（DATA XREF）的函数名，跳转到对应函数定义，

(d) 空格 切换为Graph View分析函数功能

4. 分析左侧分支（红色框）会用到“Correct!\n”：



分析输入存储位置：

```

push    offset aInputName ; "Input Name: "
rep stosd
stosb
mov     [esp+140h+var_130], 10h
mov     [esp+140h+var_12F], 20h
mov     [esp+140h+var_12E], 30h
call    sub_401109
add     esp, 4
lea     eax, [esp+13Ch+var_12C]
push    eax
push    offset aS          ; "%s"
call    _scanf
add     esp, 4
lea     eax, [esp+13Ch+var_12C]

```

[esp+13Ch+var_12C]为输入Name存储地址

```

xor     eax, eax
lea     edi, [esp+13Ch+var_12C]
push    offset aInputSerial ; "Input Serial: "
rep stosd
call    sub_401109
add     esp, 4
lea     edx, [esp+13Ch+var_12C]
push    edx
push    offset aS          ; "%s"
call    _scanf
add     esp, 8
lea     esi, [esp+13Ch+var_C8]
lea     eax, [esp+13Ch+var_12C]

```

[esp+13Ch+var_12C]为输入Serial存储地址，紧接着

```

lea     esi, [esp+13Ch+var_C8]
lea     eax, [esp+13Ch+var_12C]

```

经过分析，后面的代码是在比较两个字符串是否相等，
[esp+13Ch+var_12C]为刚刚输入的Serial，
[esp+13Ch+var_C8]应该就是输入的Name经过某些处理（算法）得到的Serial值
输入Name与输入Serial之间的代码，即为Name转换为Serial的算法，着重分析即可：

```

loc_401077:
cmp     esi, 3
jl      short loc_40107E

```

```

xor     esi, esi

```

```

loc_40107E:
movsx   ecx, [esp+esi+13Ch+var_130]
movsx   edx, [esp+ebp+13Ch+var_12C]
xor     ecx, edx
lea     eax, [esp+13Ch+var_C8]
push    ecx
push    eax
lea     ecx, [esp+140h+var_C8]
push    ecx
push    offset aS02x      ; "%s02x"
push    ecx
call    _printf
add     esp, 10h
inc     ebp
lea     edi, [esp+13Ch+var_12C]
or      ecx, 0FFFFFFFh
xor     eax, eax
inc     esi
repne scasb
not     ecx
dec     ecx
cmp     ebp, ecx
jl      short loc_401077

```

将输入Name的每个字节依次与[esp+esi+13Ch+var_130]存储的3个字节
进行异或操作，并格式化为2位16进制格式存储至[esp+13Ch+var_C8]

```

mov     [esp+140h+var_130], 10h
mov     [esp+140h+var_12F], 20h
mov     [esp+140h+var_12E], 30h

```

[esp+esi+13Ch+var_130]存储的3个字节为：10h、20h、30h

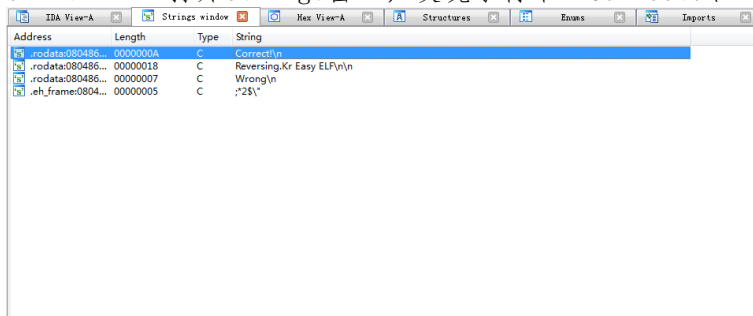
将Serial：“5B134977135E7D13”两两分组，依次分别与10h、20h、30h异或，
然后转换为字符

```
#include <stdio.h>
int main()
{
    int salt[3] = { 0x10, 0x20, 0x30 };
    char serial[8] = { 0x5B, 0x13, 0x49, 0x77, 0x13, 0x5E, 0x13, 0x5B };
    int i = 0;
    for (i = 0; i < 8; i++)
        printf("%c", serial[i] ^ salt[i % 3]);
    printf("\n");
}
```

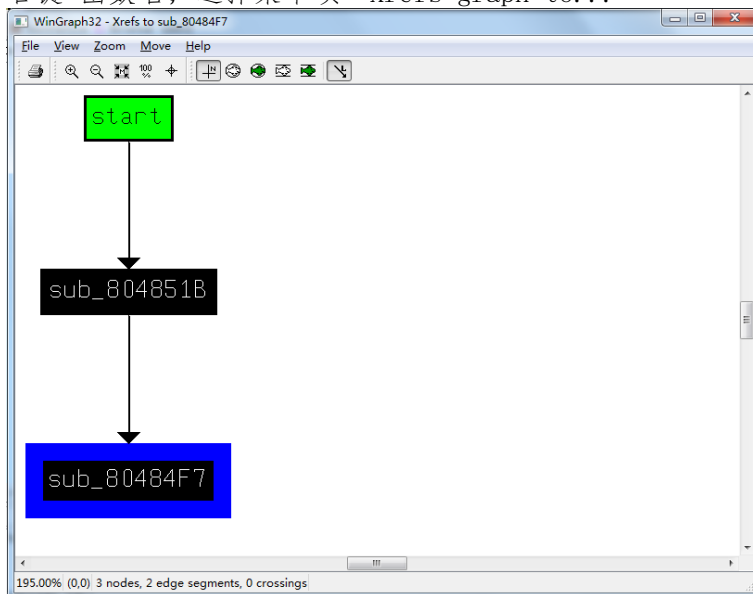
flag : “K3yg3nm3”

4 Easy ELF

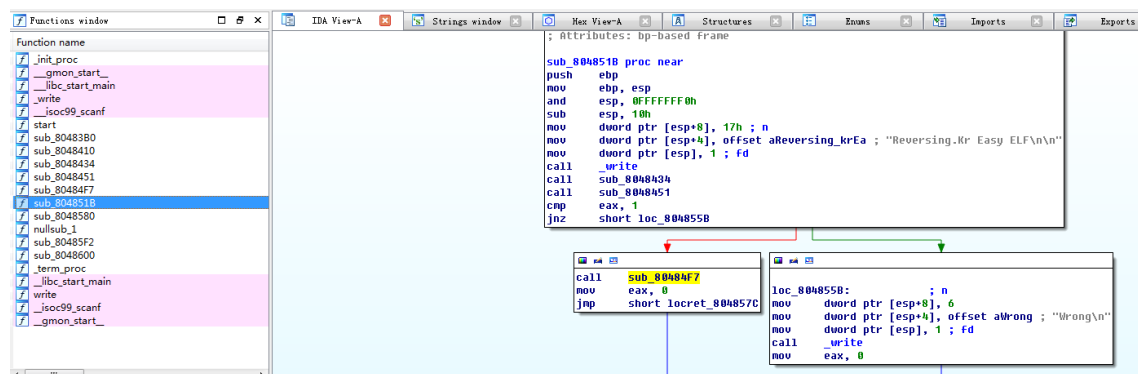
1. IDA打开查看函数列表确认没有加壳，直接静态分析。
2. Shift + F12 打开Strings窗口，发现字符串“Correct!\n”和“Wrong\n”



3. 双击 字符串“Correct!\n”跳转到定义，发现在一个单独的函数内
4. 右键 函数名，选择菜单项“Xrefs graph to...”



在函数窗口双击 函数sub_80484F7的父函数名称sub_804851B，查看定义



```

cmp     eax, 1
jnz     short loc_804855B

```

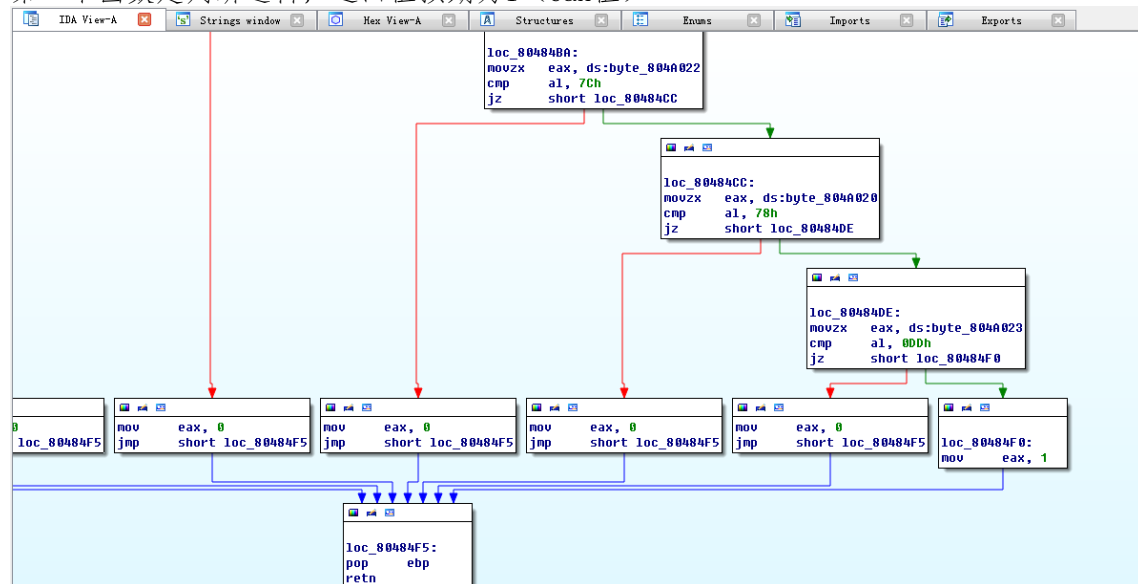
预期eax值为1，才能走到正确的分支

```

call    sub_8048434
call    sub_8048451

```

分析之前两个函数调用，第一个函数简单，是获取输入字符串存储至byte_804A020，第二个函数是判断逻辑，返回值预期为1（eax值）

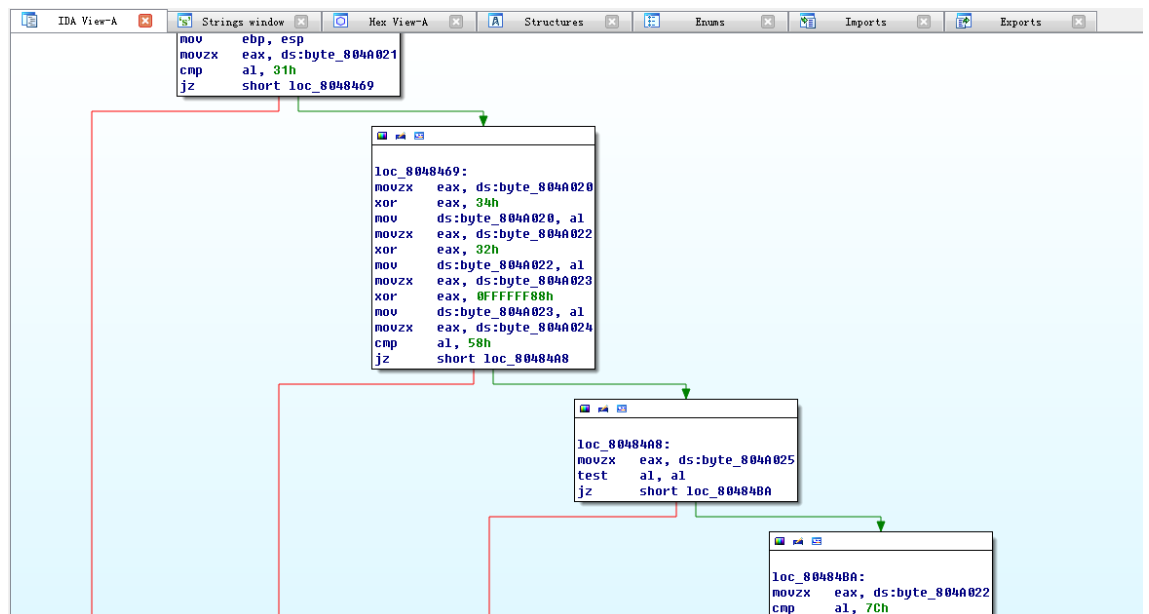


沿右侧绿色箭头指向，从后往前分析：

byte_804A023预期值是0DDh

byte_804A020预期值是78h

byte_804A022预期值是7Ch



byte_804A025预期值是00h

byte_804A024预期值是58h

byte_804A023转换为其值与0FFFFFFF88h进行异或的结果

byte_804A022转换为其值与32h进行异或的结果

byte_804A020转换为其值与34h进行异或的结果

byte_804A021预期值是31h

整理一下分析结果：

byte_804A020 异或 34h 结果预期值是78h

byte_804A021预期值是31h

byte_804A022 异或 32h 结果预期值是7Ch

byte_804A023 异或 88h 结果预期值是0DDh

byte_804A024预期值是58h

byte_804A025预期值是00h

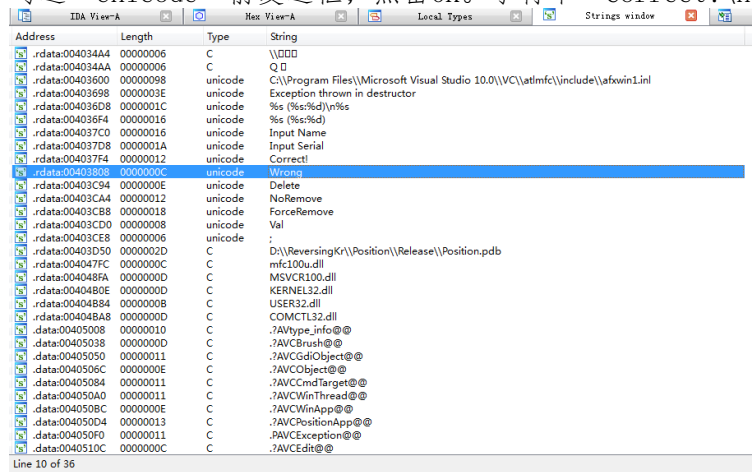
将预期值反过来操作，得出原始输入字符串：

```
#include "stdio.h"
int main()
{
    char salt[5] = {0x34, 0x00, 0x32, 0x88, 0x00};
    char inputs[5] = {0x78, 0x31, 0x7C, 0xDD, 0x58};
    int i = 0;
    for (; i < 5; i++)
        printf("%c", inputs[i] ^ salt[i]);
    printf("\n");
}
```

flag : "L1NUX"

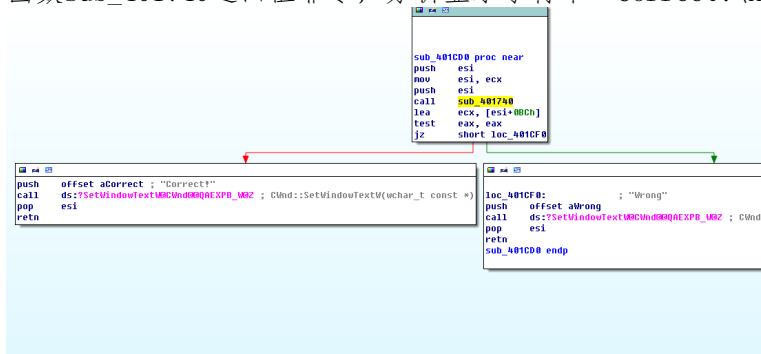
5 Position

1. IDA打开查看函数列表确认没有加壳，直接静态分析。
2. (a) Shift + F12 打开Strings窗口，查找字符串“Correct!\n”和“Wrong\n”
(b) 若无字符串“Correct!\n”，右键任一字符串，选择菜单项“Setup”
(c) 勾选“Unicode”前复选框，点击OK。字符串“Correct!\n”出现



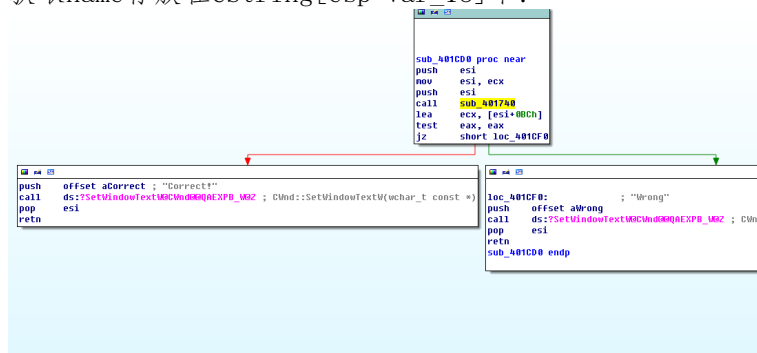
Address	Length	Type	String
.rdata:004034A4	00000006	C	\\DDD
.rdata:004034AA	00000006	C	QD
.rdata:00403600	00000098	unicode	C:\Program Files\Microsoft Visual Studio 10.0\VC\atmf\include\afxwin1.inl
.rdata:00403698	0000003E	unicode	Exception thrown in destructor
.rdata:004036D8	0000001C	unicode	%s (%s;%d)\n%s
.rdata:004036F4	00000016	unicode	%s (%s;%d)
.rdata:004037C0	00000016	unicode	Input Name
.rdata:004037D8	0000001A	unicode	Input Serial
.rdata:004037F4	00000012	unicode	Correct!
.rdata:00403958	00000002	unicode	Wrong!
.rdata:00403C94	0000000E	unicode	Delete
.rdata:00403CA4	00000012	unicode	NoRemove
.rdata:00403CB8	00000018	unicode	ForceRemove
.rdata:00403CD0	00000008	unicode	Val
.rdata:00403CE8	00000006	unicode	:
.rdata:00403D50	0000002D	C	D:\Reversing\K\Position\Release\Position.pdb
.rdata:004047FC	0000000C	C	mfc100u.dll
.rdata:004048FA	0000000D	C	MSVCR100.dll
.rdata:00404B0E	0000000D	C	KERNEL32.dll
.rdata:00404B84	00000008	C	USER32.dll
.rdata:00404BA8	0000000D	C	COMCTL32.dll
.data:00405008	00000010	C	?AVtype_info@@
.data:00405038	0000000D	C	?AVCBrush@@
.data:00405050	00000011	C	?AVCGdiObject@@
.data:0040506C	0000000E	C	?AVCObject@@
.data:00405084	00000011	C	?AVCCmdTarget@@
.data:004050A0	00000011	C	?AVCWinThread@@
.data:004050BC	0000000E	C	?AVCWinApp@@
.data:004050D4	00000013	C	?AVCPositionApp@@
.data:004050F0	00000011	C	?PAVException@@
.data:0040510C	0000000C	C	?AVCEdit@@

- (d) 双击 转到字符串定义，
(e) 双击 交叉引用 (DATA XREF) 的函数名，跳转到对应函数定义，
(f) 空格 切换为Graph View分析函数功能
3. 函数sub_401740返回值非零，分析显示字符串“Correct!\n”分支，



双击 函数sub_401740跳转到定义，分析算法：

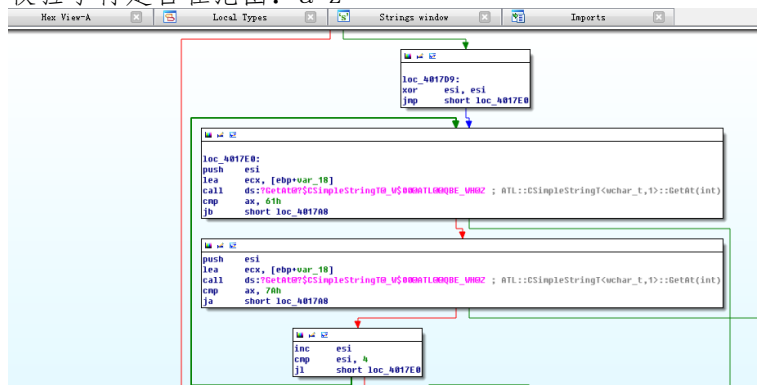
获取name存放在CString[esp+var_18]中:



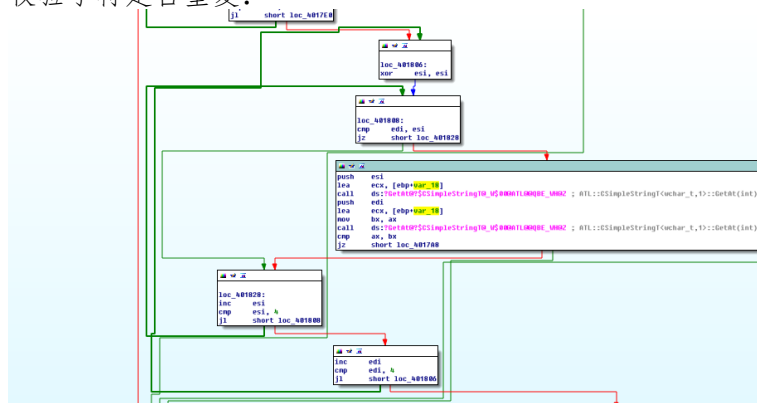
校验长度是否为4:

```
mov     ecx, [ebp+var_18]
cmp     dword ptr [ecx-0Ch], 4
```

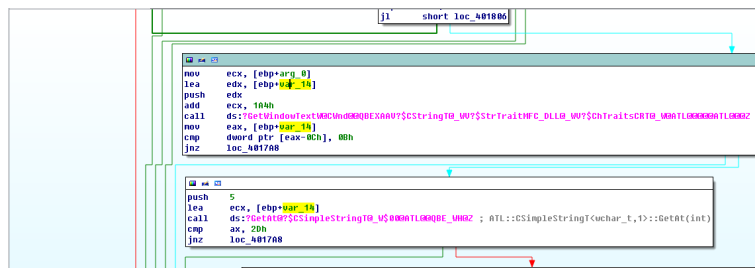
校验字符是否在范围: a-z



校验字符是否重复:



获取serial存放在CString[esp+var_14]中, 判断第5个字符是否 '-' :



关键算法分析：（var_26并未真实存储，而是直接用于计算了）

Name[0] 二进制取1、2、3、4、5位 add 5 存放 var_20 var_1F var_1E var_1D var_1C
Name[1] 二进制取1、2、3、4、5位 add 1 存放 var_28 var_27 var_26 var_25 var_24

var_20 + var_26 转换字符存储 [esp+var_10] CString [0] == Serial[0]
var_1D + var_25 转换字符存储 [esp+var_10] CString [0] == Serial[1]
var_1F + var_24 转换字符存储 [esp+var_10] CString [0] == Serial[2]
var_1E + var_28 转换字符存储 [esp+var_10] CString [0] == Serial[3]
var_1C + var_27 转换字符存储 [esp+var_10] CString [0] == Serial[4]

Name[2] 二进制取1、2、3、4、5位 add 5 存放 var_20 var_1F var_1E var_1D var_1C
Name[3] 二进制取1、2、3、4、5位 add 1 存放 var_28 var_27 var_26 var_25 var_24

var_20 + var_26 转换字符存储 [esp+var_10] CString [0] == Serial[6]
var_1D + var_25 转换字符存储 [esp+var_10] CString [0] == Serial[7]
var_1F + var_24 转换字符存储 [esp+var_10] CString [0] == Serial[8]
var_1E + var_28 转换字符存储 [esp+var_10] CString [0] == Serial[9]
var_1C + var_27 转换字符存储 [esp+var_10] CString [0] == Serial[10]
前两个字符与后两个字符处理一样：

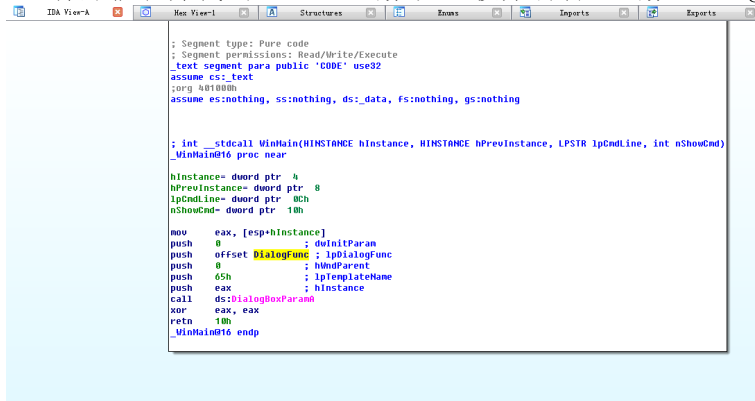
```
#include "stdio.h"
#define LEN 5
int main()
{
    char VAR_20[LEN] = { 0 };
    char VAR_28[LEN] = { 0 };
    int ch_a = 0x61;
    int ch_z = 0x7a;
    int i = ch_a;
    for (; i < ch_z+1; i++)
    {
        int j = ch_a;
        for (; j < ch_z + 1; j++)
        {
            if (i == j)
```


ay	77776
bm	77776
ci	77776
ex	77776
fl	77776
gh	77776
hu	77776
iq	77776
je	77776
ka	77776
lt	77776
mp	77776
nd	77776
~	
~	
~	
~	

有三组name满足条件 (serial=76876-77776, name=***p) : bump、cmp、ftmp
flag : " bump"

6 Replace

1. IDA打开很容易找到WinMain函数，直接分析窗口函数DialogFunc



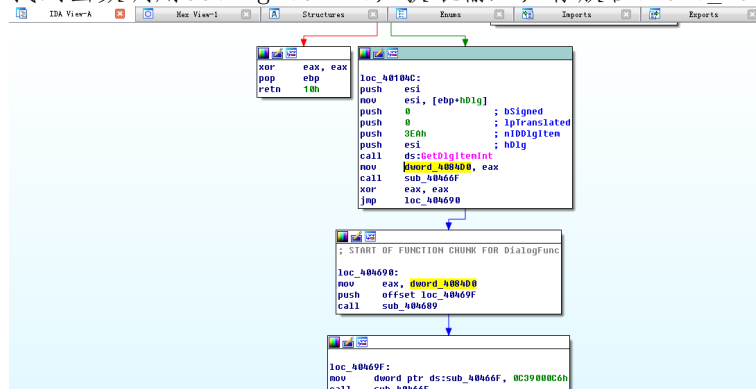
```
; Segment type: Pure code
; Segment permissions: Read/Write/Execute
_text segment para public 'CODE' use32
assume cs:text
org 401000h
assume es:nothing, ss:nothing, ds:data, fs:nothing, gs:nothing

; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
; WinMain@16 proc near

hInstance= dword ptr 4
hPrevInstance= dword ptr 8
lpCmdLine= dword ptr 0Ch
nShowCmd= dword ptr 10h

mov     eax, [esp+hInstance]
push    0                ; defaultParam
push    offset DialogFunc ; lpDialogFunc
push    0                ; hWndParent
push    65h              ; lpTemplateName
push    eax               ; hInstance
call    ds:DialogBoxParam
xor     eax, eax
ret     10h
; WinMain@16 endp
```

2. (a) 找到函数调用GetDlgItemInt，获取输入，存放在dword_4084D0



```
loc_40100C:
xor     eax, eax
pop     ebp
ret     10h

loc_4066F:
push    esi
mov     esi, [ebp+hDlg]
push    0                ; bSigned
push    0                ; lpTranslated
push    3EAh             ; nIDlgitem
push    esi               ; hDlg
mov     dword_4084D0, eax
call    sub_40466F
xor     eax, eax
jmp     loc_40669B

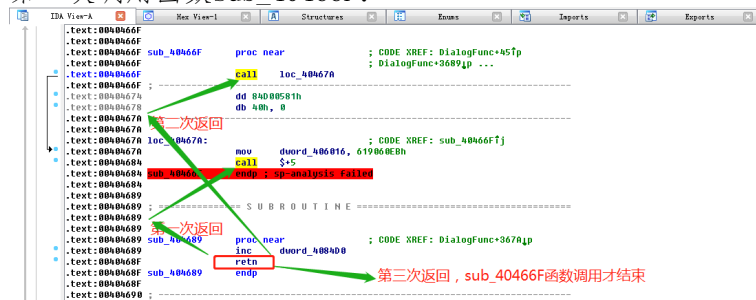
; START OF FUNCTION CHUNK FOR DialogFunc

loc_40669B:
mov     eax, dword_4084D0
push    offset loc_40669F
call    sub_404689

loc_40669F:
mov     dword ptr ds:sub_40466F, 0C39000C6h
call    sub_40466F
```

- (b) 关键函数sub_40466F调用了三次，

- (c) 第一次调用函数sub_40466F:

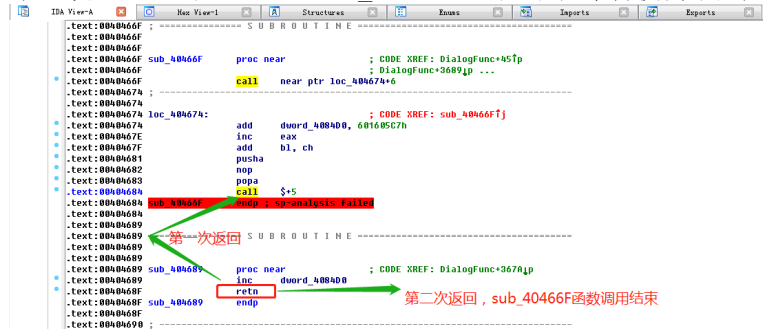


```
.text:0040466F sub_40466F proc near ; CODE XREF: DialogFunc+451p
; DialogFunc+36891p ...
.text:0040466F call loc_404670
.text:0040466F dd 8A00BC51h
.text:0040466F db 40h, 0
.text:00404670 loc_404670: mov dword_406016, 61906BEh
; CODE XREF: sub_40466Fj
.text:00404670 call $+5
; CODE XREF: sub_40466Fj
.text:00404670 jmp $+5
; SUBROUTINE
.text:00404689 sub_404689 proc near ; CODE XREF: DialogFunc+367A1p
; DialogFunc+367A1p
.text:00404689 inc dword_4084D0
.text:0040468F ret
.text:0040468F sub_404689 endp
```

针对dword_4084D0的操作:

```
inc     dword_4084D0
inc     dword_4084D0
```

第二次返回之后: call loc_40467A之后的语句转换为代码 (.text:00404674地址处, 按C



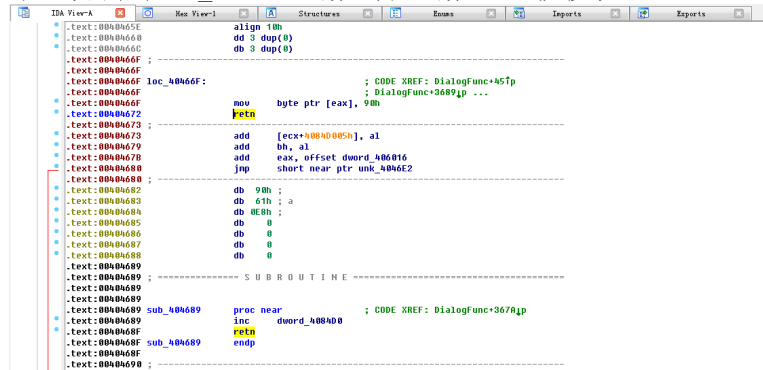
针对dword_4084D0的操作:

```

add     dword_4084D0, 601605C7h
inc     dword_4084D0
inc     dword_4084D0

```

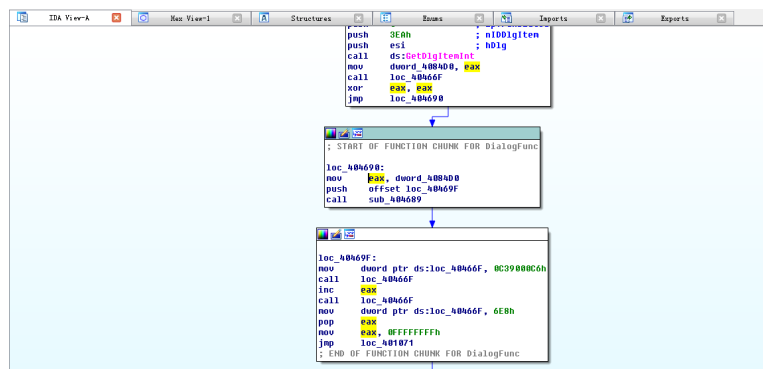
(d) 第二次调用sub_40466F函数时, 函数已经被修改:



```

mov     byte ptr [eax], 90h

```



(e) 紧接着eax加1, 第三次调用sub_40466F函数, 关键就在这里了, 好久之后终于悟到

