

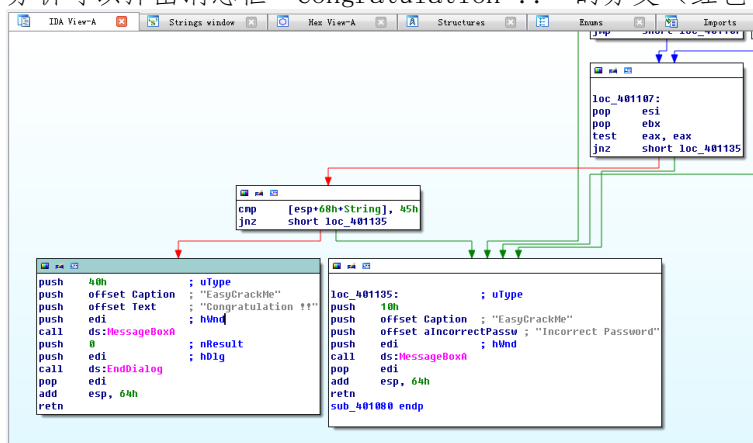
# reversing.kr-逆向挑战\_笔记

## 目录

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

## 1 Easy CrackMe

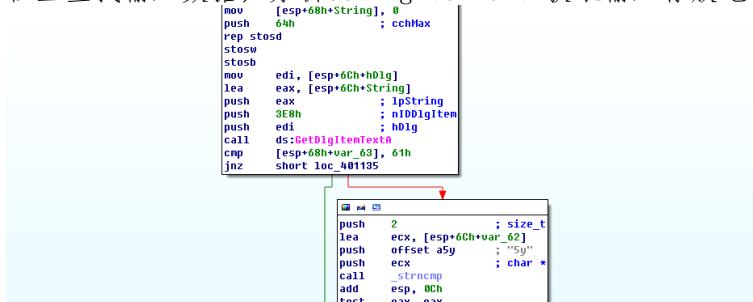
1. 运行程序，随意输入，弹窗提示：“Incorrect Password”。
2. IDA打开查看函数列表确认没有加壳，直接静态分析。
3. (a) Shift + F12 打开Strings窗口，找到字符串“Incorrect Password”，  
(b) 双击 转到字符串定义，  
(c) 双击 交叉引用（DATA XREF）的函数名，跳转到对应函数定义，  
(d) 空格 切换为Graph View分析函数功能
4. 分析可以弹出消息框“Congratulation !!”的分支（红色箭头）：



[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）

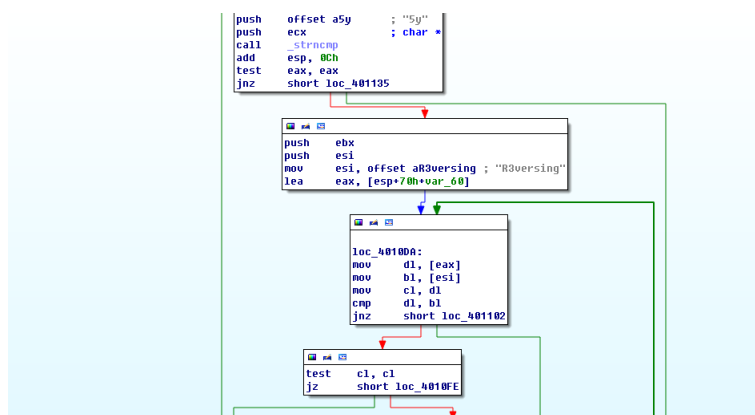
```
lea     eax, [esp+6Ch+String]
push    eax                ; lpString
```

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
```

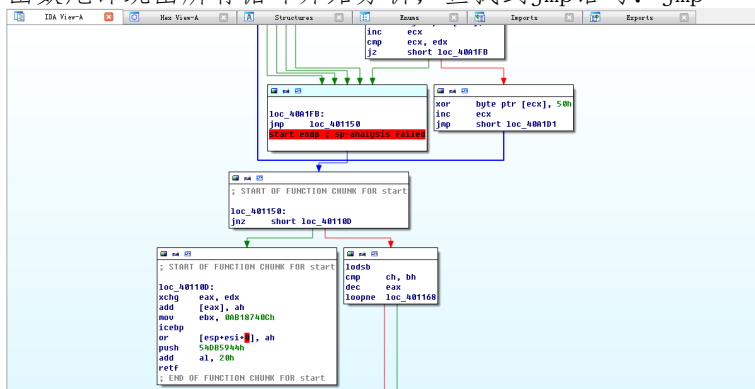
最后，根据变量定义，将4部分期待字符（串）连接在一起：

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

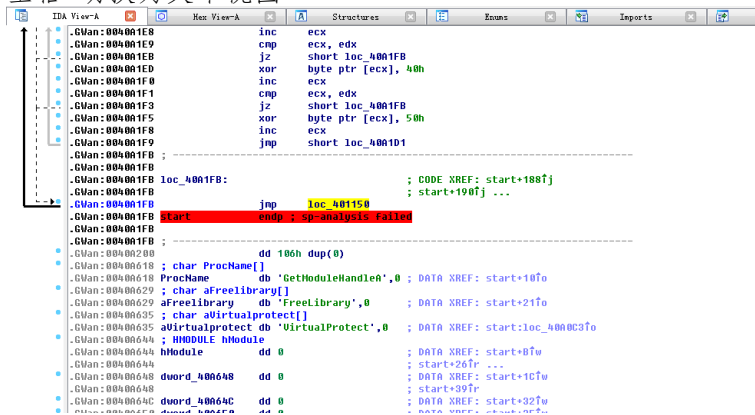
flag :    "Ea5yR3versing"
```

## 2 Easy UnpackMe

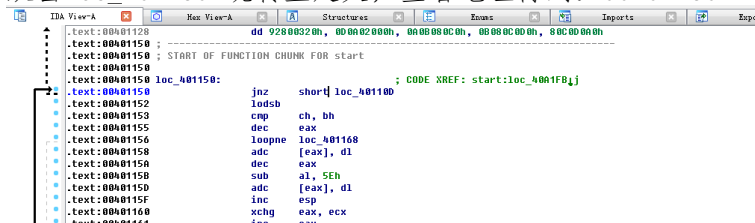
1. IDA直接打开，查看函数列表，只有一个start函数
2. 双击 查看函数 ，开始分析
3. 空格 切换为Graph视图
4. 函数尾部跳出所有循环开始分析，查找到jmp语句：jmp loc\_401150



空格 切换为文本视图



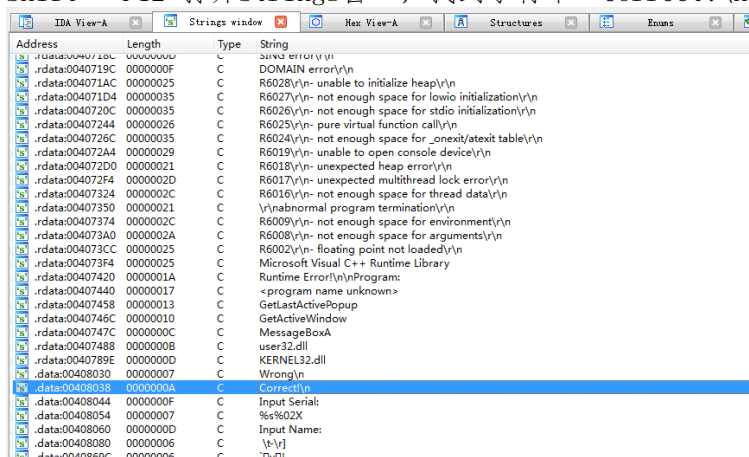
双击loc\_401150 跳转至定义，查看地址得到：00401150



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”，

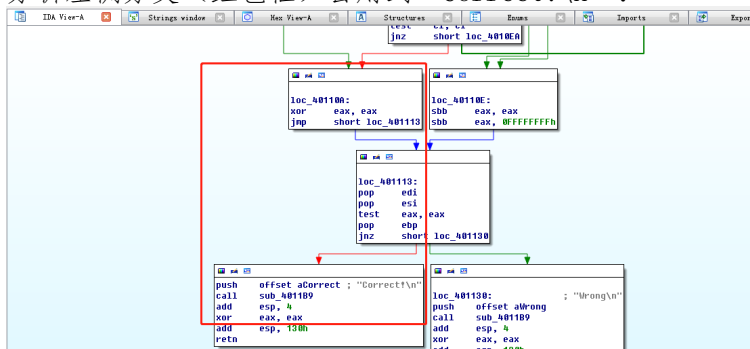


(b) 双击 转到字符串定义，

(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

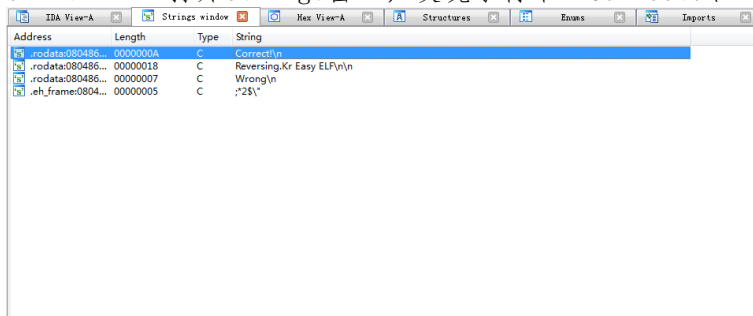
```
#include <stdio.h>
int main()
{
    int salt[3] = { 0x10, 0x20, 0x30 };
    char serial[8] = { 0x5B, 0x13, 0x49, 0x77, 0x13, 0x5E, 0x13, 0x5E };
    int i = 0;
    for (i = 0; i < 8; i++)
        printf("%c", serial[i] ^ salt[i % 3]);
    printf("\n");
}
```

flag : “K3yg3nm3”

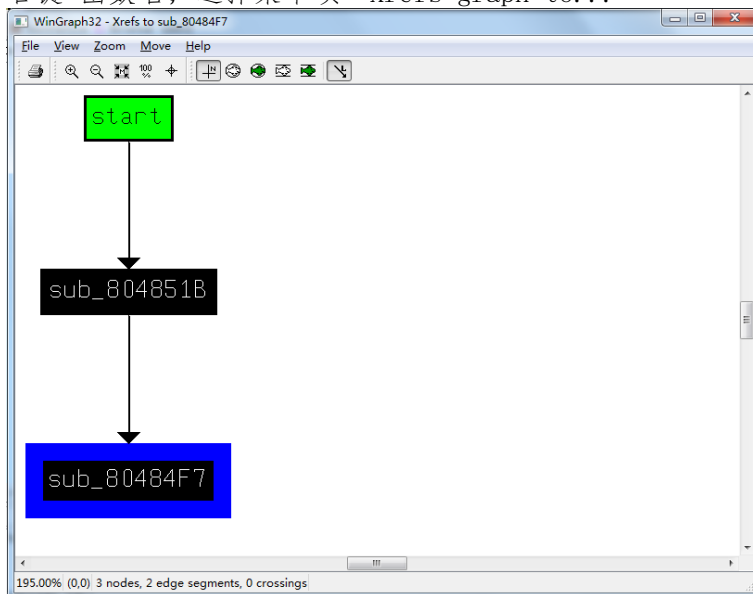


## 4 Easy ELF

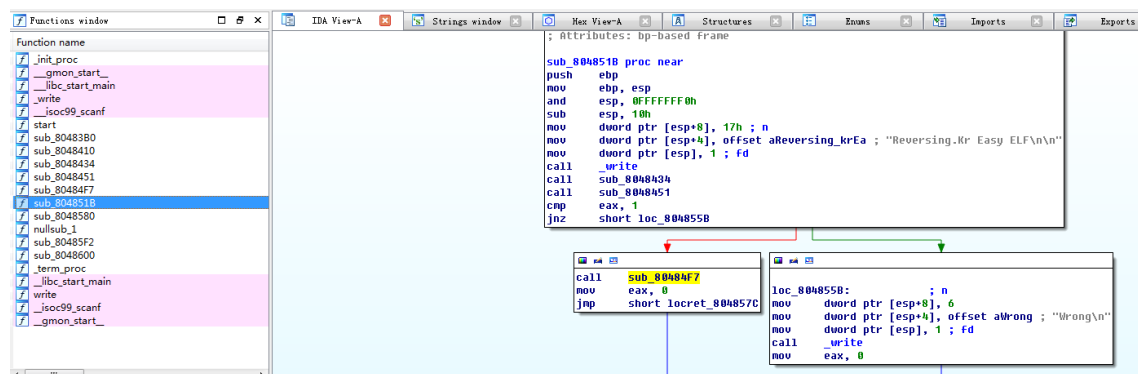
1. IDA打开查看函数列表确认没有加壳，直接静态分析。
2. Shift + F12 打开Strings窗口，发现字符串“Correct!\n”和“Wrong\n”



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

```

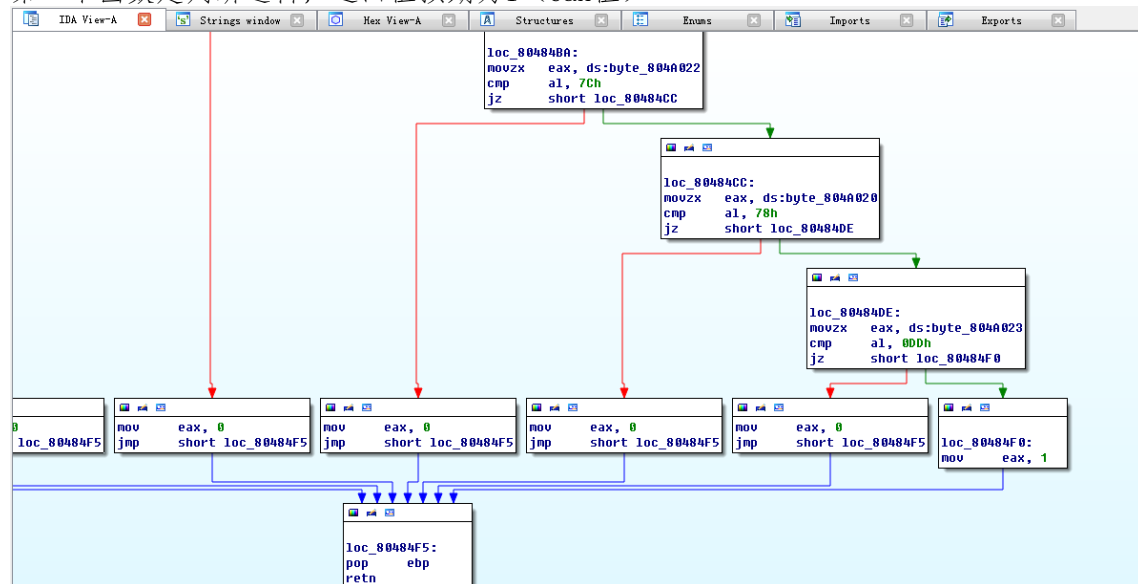
预期eax值为1，才能走到正确的分支

```

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）

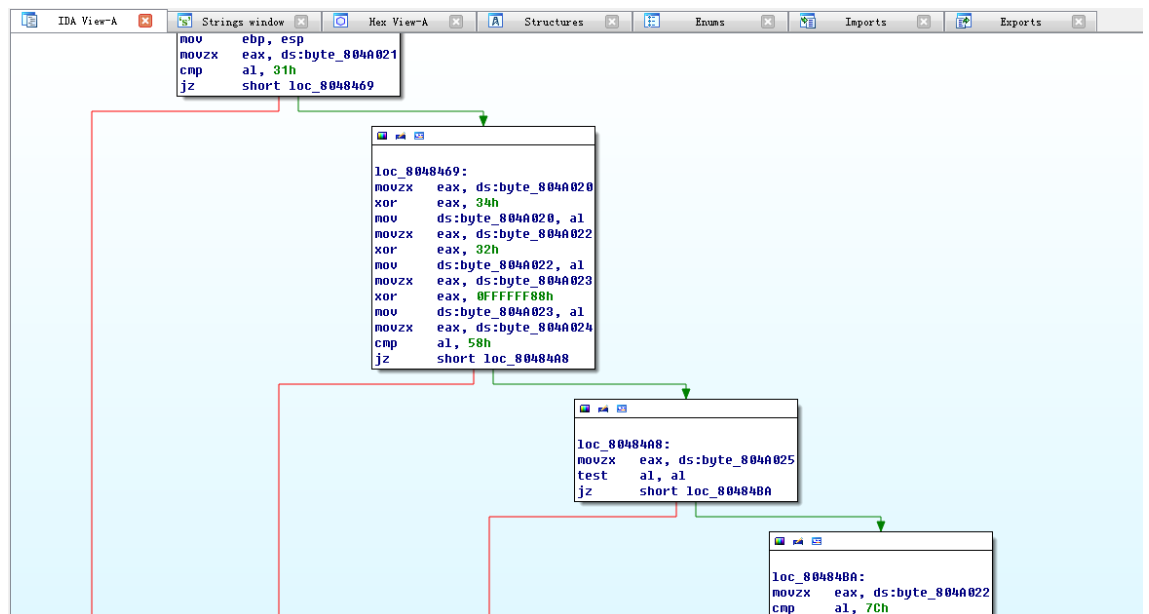


沿右侧绿色箭头指向，从后往前分析：

byte\_804A023预期值是0DDh

byte\_804A020预期值是78h

byte\_804A022预期值是7Ch



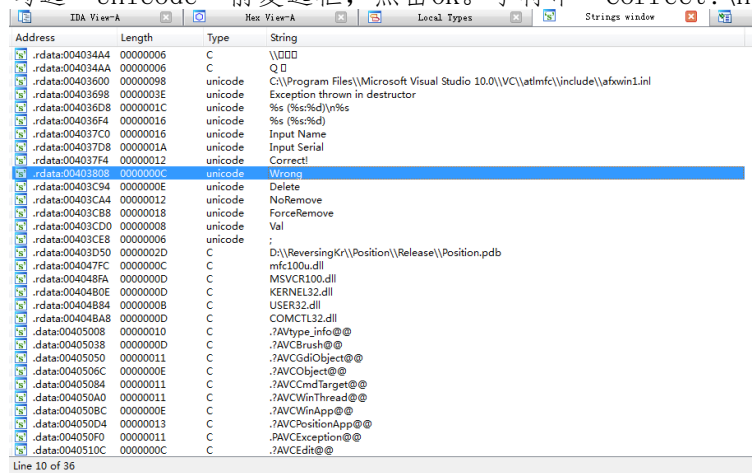
byte\_804A025预期值是00h  
 byte\_804A024预期值是58h  
 byte\_804A023转换为其值与0FFFFFFF88h进行异或的结果  
 byte\_804A022转换为其值与32h进行异或的结果  
 byte\_804A020转换为其值与34h进行异或的结果  
 byte\_804A021预期值是31h  
 整理一下分析结果：  
 byte\_804A020 异或 34h 结果预期值是78h  
 byte\_804A021预期值是31h  
 byte\_804A022 异或 32h 结果预期值是7Ch  
 byte\_804A023 异或 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");
}
```

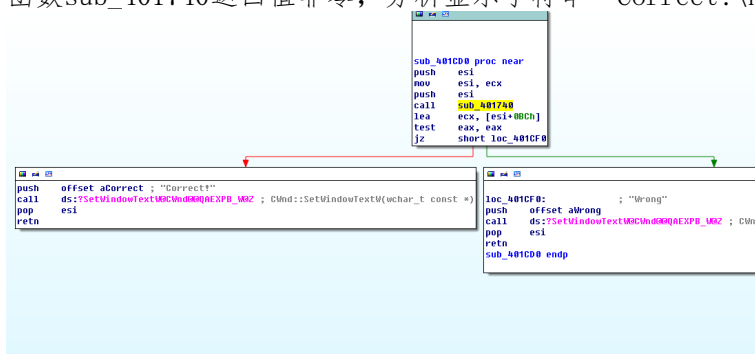
flag : "LINUX"

## 5 Position

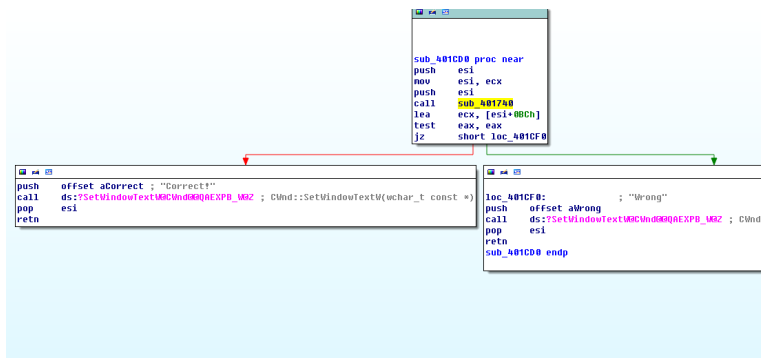
1. IDA打开查看函数列表确认没有加壳，直接静态分析。
2. (a) Shift + F12 打开Strings窗口，查找字符串“Correct!\n”和“Wrong\n”  
(b) 若无字符串“Correct!\n”，右键任一字符串，选择菜单项“Setup”、、  
(c) 勾选“Unicode”前复选框，点击OK。字符串“Correct!\n”出现



- (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：



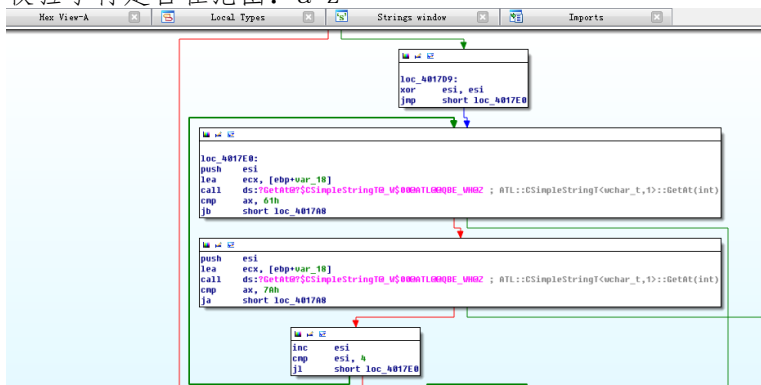
校验长度是否为4:

```

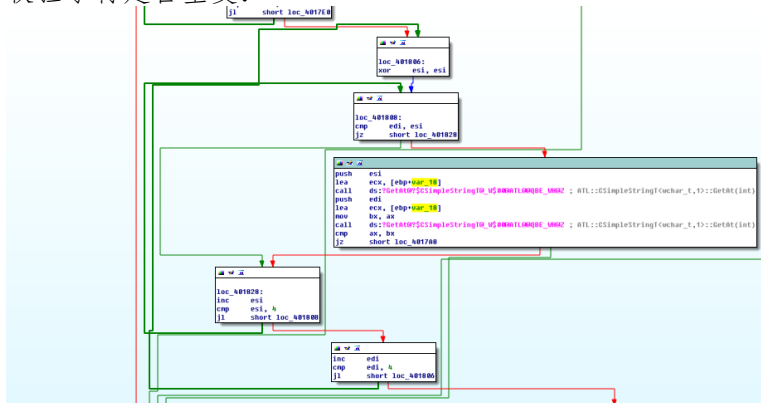
mov     ecx, [ebp+var_18]
cmp     dword ptr [ecx-0Ch], 4

```

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 '-' :





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”