```
title : AFCTF RE 部分出题报告
author : HNU-listennter
review :  CSU-吹雪
Date : 2018-5-23
```

AFCTF RE1到8题 出题/解题报告

RE1:欢迎光临

出题人:CSU-吹雪

题目描述：签到题

flag : afctf{w41c0me a1l y0u 9uys}

出题说明

这是一个签到级别的RE题目，需要选手了解IDA(基本静态逆向工具)，WinHex(以此为代表的二进制文件编辑工具)的基本功能和使用方法，在exe文件中直接提取出相关的字符串信息即可。同时值得一提的是对于一些敏感字符串的定位非常有助于我们在逆向过程中寻找关键的函数位置，希望大家能够熟练运用。源代码如下:

```c
#include <stdio.h>
#include <stdlib.h>

char flag[] = "afctf{w41c0me a1l y0u 9uys}\0";

int main() {
    char *f = (char *)malloc(0x20);
    memcpy(f, flag, 28);
    printf("Welcome !!!\n");
    return 0;
}
```

解题说明

- 你可以用做杂项的办法做这道题，比如说你可以用WinHex之类的二进制文件编辑工具打开exe文件，使用搜索字符串的功能，搜索`afctf{`来直接找到flag。

- 相似的Ubuntu等Linux系统提供的strings命令来查找文件中可见的字符串。相关信息可以参考strings命令
- 使用IDA分析文件，使用展示文件中字符串的方式(shift+f12)来搜索字符串得到flag
- 有能力的同学可以尝试写一个python的脚本用于提取文件中的全部可见字符

使用上述方法的任意一种都可以找到题目的flag.

RE2：Caesar

出题人：CSU-吹雪

题目描述：凯撒

flag : afctf{c14ssic_cae5ar}

出题说明

本题旨在让大家了解经典的替换加密算法——Caesar加密 的工作原理和逆向方法，需要选手会使用IDA的F5插件功能和一些最基本的编程能力，文件的源代码如下:

```
#include <stdio.h>
unsigned char enc_flag[] = "\x6c\x71\x6e\x7f\x71\x86\x6e\x3c\x3f\x7e\x7e
\x74\x6e\x6a\x6e\x6c\x70\x40\x6c\x7d\x88";
int main() {
    unsigned char input_flag[50];
    int i, j;
    printf("Input flag:\n");
    scanf("%s", input_flag);
    for(i = 1; i <= 16; i++) {
        j = 0;
        while(enc_flag[j]) {
            if(input_flag[j] + i != enc_flag[j]) {
                goto L;
            }
            j++;
        }
        printf("You are right.\n");
        return 0;
        L: continue;
    }
    printf("You are wrong.\n");
    return 0;
}
```

ps : 这种Caesar加密的方式没有直接写出密钥，所以需要大家写一个爆破的脚本来计算

解题说明

在查壳和运行下查看基本的exe文件信息以后，放进IDA中，由于有明显的`Input flag:`等字符串信息，很容易就能找到主要的函数流程(具体方法可以参考上一题的说明)，在识别出这个题目是Caesar加密以后找到加密以后的密文结果，在出题说明中已经提到过，源代码中没有直接写出加密使用的密钥，因此要写脚本爆破。献上我自己写的脚本

```
#! -*-coding : utf-8 -*-
enc_flag = [0x6c,0x71,0x6e,0x7f,0x71,0x86,0x6e,0x3c,0x3f,0x7e,0x7e,0x74,0
x6e,0x6a,0x6e,0x6c,0x70,0x40,0x6c,0x7d,0x88]

for i in range(0,16) :
    output = ''
    for c in enc_flag:
        output += chr(c - i)
    print output;
```

RE3：隐写术

出题说明

出题人:CSU-吹雪

题目描述：隐写术

flag : afctf{simple_picture_encryption}

本题旨在让大家了解最简单的文件加密形式，在本题中涉及到了一般CTF中不太会使用的文件读取技术，将文件读取到内存中以后,做一系列的加密操作，然后输入形成新的加密文件。同时此题在使用gcc编译以后添加了UPX壳，因此需要大家使用UPX的脱壳机或者手动脱壳以后才能对文件做基本的分析，UPX官方加脱壳机在群里有，入门导论也介绍过如何使用.程序源代码如下:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    unsigned char *buf, c;
    unsigned int i = 0;
    buf = (unsigned char *)malloc(0x200000);
    FILE *infile, *outfile;
    if((infile = fopen("flag.jpg", "rb")) == NULL) {
        printf("Cannot find \'flag.jpg\'\n");
        exit(1);
    }
    while(!feof(infile)) {
```

```
        c = getc(infile);
        c ^= 0x73;
        buf[i] = c;
        i++;
    }
    outfile = fopen("flag.jpg.enc", "wb");
    fwrite(buf, 1, i - 1, outfile);
    fclose(infile);
    fclose(outfile);
    return 0;
}
```

解题说明

首先，UPX的脱壳方法:

- 使用UPX脱壳机，可以下载群里面给出的官方版本，也可以参考其他人的脱壳机:https://down.52pojie.cn/Tools/Unpackers/
- 相关比较详细的脱壳教程，大家可以在吾爱破解中搜索Ximo脱壳教程比较适合新手，如果大家有兴趣，欢迎加listennter的QQ(602792777)一起学习交流(脱壳萌新一枚)

脱壳结束以后，在通过`flag.jpg.enc`等字符串信息找到主要的功能函数，然后就可以分析出程序的加密工作原理，加密其实就是简单的把原图片异或了0x73，由于异或算法加密密钥与解密密钥相同，加密算法与解密算法相同，一个最好的办法就是把`flag.jpg.enc`改成`flag.jpg`，然后直接运行程序，再把新出来的`flag.jpg.enc`改成`flag.jpg`就可以打开看到手写有flag的图片了，这种方法可以不用编程。

当然写解密脚本解该题也是可以的，输出以后是一个图片文件，打开以后就是flag了。解密脚本如下:

```
#! -*- coding:utf-8 -*-

fp = open('flag.jpg.enc','rb')
input = fp.read()
output = ''
fp.close()

for c in input :
    output += chr(ord(c) ^ 0x73)

fp = open('flag.jpg','wb')
fp.write(output)
fp.close()
```

RE4：简单编码

出题人:HNU-listennter

题目描述：一个简单的python逆向

flag : afctf{CaN_Uu_Br0ak_1T}

出题说明

此题部分取自2017HCTF竞赛中的RE1最后一部分，旨在让大家明白能够使用最基本的爆破技术，通过每个字符各个击破的方式解密出flag，由于题目中出现了'&'和'|'运算等不太容易逆向的运算，因此想要找到加密函数的逆运算就不太容易，Okay，加密源代码如下(已经提供给选手们)；

```
#! -*- coding:utf-8 -*-
flag = ''
f = open('flag.txt','r')
input = f.readline()
assert input.startswith('afctf{')
assert input.endswith('}')
flag = input[6:21]
f.close()
print input + flag

def enc1(word, i):
        word = ord(word) ^ 0x76 ^ 0xAD
        temp1 = (word & 0xAA) >> 1
        temp2 = 2 * word & 0xAA
        word = temp1 | temp2
        return word

def enc2(word, i):
        word = ord(word) ^ 0x76 ^ 0xBE
        temp1 = (word & 0xCC) >> 2
        temp2 = 4 * word & 0xCC
        word = temp1 | temp2
        return word


def enc3(word,i):
        word = ord(word) ^ 0x76 ^ 0xEF
        temp1 = (word & 0xF0) >> 4
        temp2 = 16 * word & 0xF0
        word = temp1 | temp2
        return word

output = ''

for i in range(5):
    output += chr(enc1(flag[i],i))
```

```
for i in range(5):
    output += chr(enc2(flag[i+5],i))

for i in range(5):
    output += chr(enc3(flag[i+10],i))

f = open('output','w')
f.write(output)
f.close()
```

解题说明

用二进制编辑器找到output文件中的二进制的密文，或者可以使用python直接读取进来进行相关操作。

爆破可以直接使用题目中给出的加密函数，最后，还是献上解密的脚本

最后想说一下，爆破有一个最基本的条件，就是穷举难度不能太大，像这道题，加密是针对于每个字符单独加密的，其他字符的变化对该字符加密的结果没有任何影响，所以穷举难度是【可显字符数 * 15】而不是【可显字符数 ^ 15】，因此具备爆破的条件

```
#! -*- coding : utf-8 -*-
import string
f = open('output','rb')
enc_flag = f.read()
f.close()
output = 'afctf{'

def enc1(word, i):
        word = ord(word) ^ 0x76 ^ 0xAD
        temp1 = (word & 0xAA) >> 1
        temp2 = 2 * word & 0xAA
        word = temp1 | temp2
        return word

def enc2(word, i):
        word = ord(word) ^ 0x76 ^ 0xBE
        temp1 = (word & 0xCC) >> 2
        temp2 = 4 * word & 0xCC
        word = temp1 | temp2
        return word


def enc3(word,i):
        word = ord(word) ^ 0x76 ^ 0xEF
        temp1 = (word & 0xF0) >> 4
        temp2 = 16 * word & 0xF0
        word = temp1 | temp2
        return word
```

```
for i in range(0,5):
    for word1 in string.printable:
        res1 = enc1(word1, i)
        if(res1 == ord(enc_flag[i])):
            print chr(ord(word1))
            output += chr(ord(word1))

for i in range(0,5):
    for word1 in string.printable:
        res1 = enc2(word1, i)
        if(res1 == ord(enc_flag[5+i])):
            print chr(ord(word1))
            output += chr(ord(word1))

for i in range(0,5):
    for word1 in string.printable:
        res1 = enc3(word1, i)
        if(res1 == ord(enc_flag[10+i])):
            print chr(ord(word1))
            output += chr(ord(word1))

print output + '}'
```

RE5：吃豆子

出题人：CSU-吹雪

题目描述：吃豆子游戏

flag：afctf{kkljllkkkkhhk}

出题说明

此题是一道比较经典的简单逆向题目，相信大家都玩过贪吃蛇或者走迷宫等小游戏，这一系列的题目就是要选手们在逆向的过程中玩这些小游戏，除了本题的吃豆子游戏以外，还有数独等游戏，此题考验选手们的代码阅读理解能力，选手需要根据 `8 * i + j` 这种线索反应出这不是一个一维数组，而是一个8 * 8矩阵，当能够判断出这是一个吃豆子(走迷宫)游戏以后，基本就已经做出来了，源代码如下:

```
#include <stdio.h>
#include <string.h>

char layout[] = "*********B*******_*___***___*_*******_*******_*****___**
***O****";
```

```c
int judge(char *flag);

int main() {
    char flag[50];
    printf("Flag ??\n");
    scanf("%s", flag);
    if (judge(flag))
        printf("False.\n");
    else
        printf("True.\n");

    return 0;
}

int judge(char *flag) {
    char sub_str[10];
    int i, j, k, len;
    strncpy(sub_str, flag, 6);
    sub_str[6] = '\0';
    len = strlen(flag);
    if (strcmp(sub_str, "afctf{") || flag[len - 1] != '}')
        return -1;
    for(i = 0; i < 8; i++)
        for(j = 0; j < 8; j++)
            if(layout[8 * i + j] == 'B')
                goto LABEL1;
    LABEL1:
    for(k = 6; k < len - 1; k++) {
        switch (flag[k]) {
            case 'h':
                layout[8 * i + j] = '*';
                j--;
                break;
            case 'l':
                layout[8 * i + j] = '*';
                j++;
                break;
            case 'j':
                layout[8 * i + j] = '*';
                i--;
                break;
            case 'k':
                layout[8 * i + j] = '*';
                i++;
                break;
            default:
                return -1;
        }
        if(layout[8 * i + j] == '*') {
            return -1;
        }
        if(layout[8 * i + j] == 'O') {
```

```
            return 0;
        }
    }
}
```

解题说明

通过运行程序可以发现程序中存在`flag ??`的字符串，通过IDA的搜索字符串的功能回溯到游戏的运行函数以后，很容易就能发现此时程序加载了一个全是`*,0,_`组成的字符串，在理解下面的代码以后能理解游戏是如何运行的(可能需要一点时间来理解这部分的代码，刚开始分析的时候可能略有些难度，希望大家放平心态慢慢来)

```
size_t __cdecl sub_40168F(char *Str)
{
  size_t result; // eax@3
  signed int v2; // eax@12
  char Str1; // [sp+16h] [bp-22h]@1
  char v4; // [sp+1Ch] [bp-1Ch]@1
  size_t v5; // [sp+20h] [bp-18h]@1
  int k; // [sp+24h] [bp-14h]@11
  int j; // [sp+28h] [bp-10h]@5
  int i; // [sp+2Ch] [bp-Ch]@4

  strncpy(&Str1, Str, 6u);
  v4 = 0;
  v5 = strlen(Str);
  if ( !strcmp(&Str1, "afctf{") && Str[v5 - 1] == 125 )
  {
    for ( i = 0; i <= 7; ++i )
    {
      for ( j = 0; j <= 7; ++j )
      {
        if ( *(&byte_403020[8 * i] + j) == 66 )
          goto LABEL_11;
      }
    }
LABEL_11:
    for ( k = 6; ; ++k )
    {
      result = v5 - 1;
      if ( (signed int)(v5 - 1) <= k )
        break;
      v2 = Str[k];
      if ( v2 == 106 )
      {
        *(&byte_403020[8 * i--] + j) = 42;
      }
      else if ( v2 > 106 )
```

```
        {
          if ( v2 == 107 )
          {
            *(&byte_403020[8 * i++] + j) = 42;
          }
          else
          {
            if ( v2 != 108 )
              return -1;
            *(&byte_403020[8 * i] + j++) = 42;
          }
        }
        else
        {
          if ( v2 != 104 )
            return -1;
          *(&byte_403020[8 * i] + j--) = 42;
        }
        if ( *(&byte_403020[8 * i] + j) == 42 )
          return -1;
        if ( *(&byte_403020[8 * i] + j) == 79 )
          return 0;
      }
    }
    else
    {
      result = -1;
    }
    return result;
}
```

通过IDA的F5功能可以看见如上的代码，其实大家可以很容易的看出来，这里使用的是二维数组的寻址方式，我们可以按照这种分组方法画出一个迷宫图，然后flag就是走出迷宫的路径喽~

RE6：这个题目不寻常

出题人：CSU-吹雪

题目描述：这个题目不寻常

flag：afctf{_runT1me_D4cr7pt10n_1s_iNt4resT1ng}

出题说明

此题对于有经验的逆向选手毫无难度，只需要在关键函数后面下个断点，然后动态调试执行到此处，跟踪输入的参数即可秒解，但是由于在入门指南里面只讲了静态分析，没有讲动态调试，所以下面我着重讲一下静态分析的方法。

此题对输入flag以后的相应处理代码进行了简单的加密，这是加密壳的一种简单形式。通过对核心代码的加密变换阻止程序被直接逆向。本题旨在让大家了解加密壳的工作原理，需要选手有一定的C语言功底，基本的shellcode思想，和一点点的逆向功底。考虑到大家的C语言可能不太熟练，本题我将简单讲解源代码，源代码如下:

```
#include <stdio.h>
char s[]={68, 152, 244, 66, 146, 253, 1, 214, 84, 233, 57, 17, 17, 17, 25
0, 56, 154, 68, 233, 154, 84, 25, 16, 193, 154, 92, 233, 154, 68, 25, 16,
 219, 30, 167, 11, 154, 68, 233, 156, 91, 238, 154, 68, 25, 16, 219, 30,
167, 3, 32, 203, 153, 1, 146, 124, 233, 16, 146, 108, 233, 17, 110, 192,
129, 146, 213, 1, 74, 76, 210};
char enc[]={97, 20, 102, 9, 123, 26, 97, 19, 102, 8, 92, 109, 0, 101, 58,
 126, 74, 41, 91, 108, 28, 104, 89, 105, 7, 88, 105, 26, 69, 44, 98, 22,
34, 80, 53, 70, 18, 35, 77, 42, 87};
int main(){
    void (*jiemi)(char *);
    char *c;
    jiemi = (void (*)(char *))malloc(70);
    memcpy((char *)jiemi,s,70);
    c = (char *)jiemi;
    for(int i=0;i<70;i++,c++)
        *c ^= 0x11;
    jiemi(enc);
    printf("Good luck !!!");
    return 0;
}
```

此题没有输入和输出，在main函数中首先对jiemi函数做一个异或解密，然后调用jiemi函数跳转过去解密flag。下面详细解释下C语言代码:

- main函数首先声明了一个函数指针 jiemi,其中包括一个 `char *` 类型的参数
- 然后使用malloc函数为jiemi函数分配了70字节的内存空间，并使用memcpy函数将s[]传入jiemi函数的空间中，此时s数组里面其实存放的是一堆加密后的函数机器码，只有先解密这个函数，才知道这个函数是如何怎么解密flag的。
- 之后声明了一个字符类型的指针c，指向jiemi函数的地址，并通过循环异或了0x11进行函数解密；
- 最后，调用jiemi函数，将enc作为参数解密；

然后献上解密后的原函数

```
void jiemi(char *s){
    for(int i = 40; i >= 1; i--)
        s[i] ^= s[i - 1];
}
```

总而言之，这道题先通过异或0x11，把解密函数本身先解密出来，再执行解密函数jiemi，把flag解密出来；

解密过程

首先，我们使用脚本将已经加密的函数解密，如果你会IDC脚本的话，其实可以运行脚本直接把s数组解密，然后再按P生成函数，最后F5看到源代码，IDC脚本如下：

```
#include <idc.idc>
static main() {
    auto ea = ScreenEA();    //获得当前光标指向的地址
    while(Byte(ea)) {        //取出地址为ea的一字节数据，若为0，结束循环
        PatchByte(ea, Byte(ea) ^ 0x11); //将数据异或0x11，并写回ea地址
        ea++;
    }
}
```

然后把光标点到s数组的首地址位置，执行脚本就成功解密了。

如果你不会IDC脚本，可以先写一个普通的脚本把解密函数的机器码打出来，再一个一个手动地在Hex View窗口修改字节码，70个字节总体来说还可以接受，只是比前一个方法来说确实要麻烦一些。

RE7：他在搞什么鬼花样

出题人：CSU-吹雪

题目描述：libwinpthread-1.dll这个文件只是为了能够让程序运行，不需要分析

flag：afctf{Mu1ti_THre4d_15_9o0d}

出题说明

此题仍然使用了RE6的思路，然而在本题中，使用了多线程的技术，进一步增加了逆向分析的难度，需要选手在能解出RE6的基础上，理解多线程 ,源代码如下:

```
#include <pthread.h>
#include <windows.h>

unsigned char dest[0x100];
unsigned char *password;
unsigned char flag[] = "\x29\x2c\x2b\x3a\x2c\xc3\x15\x3d\x79\x3a\x31\x27
\x1a\x0e\x38\x2d\x7a\x2a\x27\x79\x7d\x27\x01\x37\x76\x2a\xc5";
unsigned char enc_func[] =
{0x55,0x89,0xE5,0x83,0xEC,0x10,0x8B,0x45,0x08,0x89,0x45,0xFC,0xEB,0x24,0x
8B,0x45,0xFC,0x0F,0xB6,0x00,0x83,0xC0,0x05,0x89,0xC2,0x8B,0x45,0xFC,0x88,0
x10,0x8B,0x45,0xFC,0x0F,0xB6,0x00,0x83,0xC8,0x22,0x89,0xC2,0x8B,0x45,0xFC,0
```

```
x88,0x10,0x83,0x45,0xFC,0x01,0x8B,0x45,0xFC,0x0F,0xB6,0x00,0x84,0xC0,0x75,0
xD2,0x90,0xC9,0xC3}; //63 bytes

void f1() {
    while(!(*dest))
        Sleep(1000);
    dest[0x16] = 0x07;
}

void f2() {
    while(!(*dest))
        Sleep(1000);
    dest[0x26] = 0x41;
}

void f3() {
    while(!(*dest))
        Sleep(1000);
    dest[0x25] = 0xf0;
}

void f4() {
    for(int i = 0; i < 63; i++)
    dest[i] = enc_func[i];
}

void start_routine() {
    pthread_t t1, t2, t3, t4;
    pthread_create(&t1, 0, (void *)f1, 0);
    pthread_create(&t2, 0, (void *)f2, 0);
    pthread_create(&t3, 0, (void *)f3, 0);
    pthread_create(&t4, 0, (void *)f4, 0);
}

int verify() {
    unsigned char *input;
    int i, len_of_flag;
    if (strlen(password) <= 8)
        return 0;
    input = password;
    while(!(*dest))
        Sleep(1000);
    ((void (*)(char *))dest)(input);
    len_of_flag = strlen(flag);
    for(i = 0; i < len_of_flag; i++) {
        if(!flag[i] && len_of_flag - 1 != i)
            return 0;
        if(input[i] != flag[i])
            return 0;
    }
    return 1;
}
```

```
int main() {
    start_routine();
    printf("=== START VERIFYING ===\n\n");
    password = (char *)malloc(0x40);
    printf("[?]PASSWORD\n");
    scanf("%s", password);
    if(verify()) {
        printf("[*]LOGIN SUCCESSFUL\n");
    }
    else {
        printf("[!]INVALID PASSWORD\n");
    }
    return 0;
}
```

在t1-t4四个线程中，t1-t3只是改变了解密函数的部分字节，让解密函数能够正常工作，其实改变之前的代码是这样的：

```
void enc(char *str) {
    char *i;
    for(i = str; *i != '\0'; i++) {
        *i += 5;
        *i |= 0x22;
    }
}
```

很明显，与上一题不同的是，enc_func这个函数本身是可读的，但确是错误的，如果你不细心，忽视了t1到t3这三个线程的细微改动的话，那么分析这段代码就一直对不了。事实上你也会发现`*i |= 22`的“或”算法根本不可逆，这也是一个线索证明这个算法有问题。

通过修改机器码，第一个线程将`*i += 5`变成了`*i += 7`；第二个线程将`*i |= 0x22`变成了`*i |= 0x41`，第三个线程是直接修改了操作码，使得汇编的“or”指令变成了“xor”指令，所以指令变成了`*i ^= 0x41`。因此修改后的函数变为：

```
void enc(char *str) {
    char *i;
    for(i = str; *i != '\0'; i++) {
        *i += 7;
        *i ^= 0x41;
    }
}
```

所以解flag的核心算法就是：

```
void dec(char *enc_flag) {
  char *i;
  for(i = enc_flag,*i != '\0'; i++) {
```

```
        *i ^= 0x41;
        *i -= 7;
        printf("%c",*i);
    }
}
```

解题说明

此题首先要解决一个问题，`dest`到底是个数组还是一个函数？其实`dest`既是数组，也是函数。根据冯·诺依曼体系结构，指令和数据以同等的地位存储在内存中，所以本质上来说，指令也好，数据也好，归根结底都是一堆二进制数据。`dest`在定义的时候是按一个未初始化的数组来定义的，但是在里面存储的值又恰巧符合特定格式可以被识别成为指令，因此我将`dest`强制转换成一个函数指针就变成执行`dest`数组里面存放的指令了。

这个题要彻底理解的话还需要明白“线程同步”这个概念，由于线程是同时进行的，但有时候做件事是有先后的，此时先做完的那个线程有时就需要等待，等待其他线程先把一些事情做完才继续。`Sleep`函数就是用来干这件事的，`Sleep(1000)`表示让该线程休眠1000毫秒。

回到这个题目上来，程序首先在启动了主线程后又启动了四个线程。程序先将`enc_func`里面的内容复制到`dest`数组里面，显然，如果`dest`数组是空的，那么修改`dest`数组的内容，以及调用`dest`数组所反映的函数是没有意义的，因此其他线程在`dest`没有值的时候要休眠等待，等到线程4把数组复制过去了才能继续。

相同的此题也可以使用动态调试的方法，不过此题有多线程的原因，可能动态调试相对困难。

RE8：雕虫小技

出题人：CSU-吹雪

题目描述：雕虫小技

flag：afctf{A5m_1s_N0t_d1ff1cvLT}

出题说明

此题主要考察花指令，花指令通过一些特殊的汇编指令，来干扰IDA的静态分析来干扰选手逆向。更多关于花指令的资料可以点击下面这条链接:

https://www.52pojie.cn/thread-48942-1-1.html

本题的源代码如下:

```
#include <stdio.h>
#include <string.h>
```

```
int enc(char *flag);

char f[30] = "\x16\x66\x36\x47\x66\xb7\x14\x53\xd6\xf5\x13\x37\xf5\xe4\x0
3\x47\xf5\x46\x13\x66\x66\x13\x36\x67\xc4\x45\xd7\x00";

int main() {
    char flag[30];
    int result;
    printf("please input the flag:\n");
    scanf_s("%s", flag, 30);
    result = enc(flag);
    if (!result)
        printf("You are right.\n");
    else
        printf("You are wrong.\n");
    return 0;
}

int enc(char *flag) {
    int len = strlen(flag);
    int i;
    if (len != 27)
        return -1;
    for (i = 0; i < len; i++) {
        __asm {
            mov ebx, [flag]
            mov esi, i
            mov dl, byte ptr[ebx + esi]
            mov dh, dl
            shl dl, 4
            shr dh, 4
            or dl, dh
            mov byte ptr[ebx + esi], dl
        }
    }
    __asm {                          //花指令部分
        xor eax, eax
        test eax, eax
        jz label
        __emit 0e8h
        label:
        xor eax, eax
    }
    return strcmp(flag, f);
}
```

解题说明

首先使用IDA分析程序，发现程序虽然跳向了enc函数的位置，但是IDA却没有把这一部分解析成为一个函数，更不能按F5，而是作为了一个loc的代码段，这说明这一部分解析是存在问题的，再来关注这段loc代码的最后一部分。

```
.text:004010B0 loc_4010B0:                             ; CODE XREF: .tex
t:00401096j
.text:004010B0                 xor     eax, eax
.text:004010B2                 test    eax, eax
.text:004010B4                 jz      short near ptr loc_4010B6+1
.text:004010B6
.text:004010B6 loc_4010B6:                             ; CODE XREF: .tex
t:004010B4j
.text:004010B6                 call    near ptr 30A8D0EEh
.text:004010BB                 add     byte ptr [eax+0], 8Bh
.text:004010BF                 inc     ebp
.text:004010C0                 or      [eax-18h], dl
.text:004010C3                 pop     ecx
```

可以发现这里的call指令跳向了一个根本不存在的地址(IDA中用红色标注)，而在`0x004010b4`位置的跳转指令，并不是跳向了IDA解析中的`0x004010b6`的位置，而是跳向了偏移为1的位置，因此这里并不会像IDA展示的汇编代码这样运行，由于IDA开始解析的位置不正确，导致了后面的汇编指令全都不正确。

此时可以尝试将`0x004010b6`处填充为nop(这是由于`0x004010b4`处的跳转决定的)，然后在使用IDA分析，此时IDA可以将此处识别为正常的代码了，F5以后的伪C如下所示:

```
int __cdecl sub_40105B(char *a1)
{
  int result; // eax@2
  signed int i; // [sp+10h] [bp-4h]@3

  if ( strlen(a1) == 27 )
  {
    for ( i = 0; i < 27; ++i )
      a1[i] = ((unsigned __int8)a1[i] >> 4) | 16 * a1[i];
    result = strcmp(a1, &byte_408030);
  }
  else
  {
    result = -1;
  }
  return result;
}
```

因此算法就是将一个字节的高4位和低4位换个位置，算法可逆，此时再写脚本来就可以找到最后的flag了。

花指令原理

现在我们来简单分析一下这道题目用的花指令的原理，首先要知道，花指令不影响程序运行，但是影响反汇编器的分析，是一个常用的反逆向技术，我们先来分析一下花指令代码：

```
__asm {                          //花指令部分
    xor eax, eax
    test eax, eax
    jz label
    __emit 0e8h
    label:
    xor eax, eax
}
```

这里我们通过`__emit 0e8h`强行插入了一个机器码`E8`，它是一个什么东西呢？它其实就是汇编指令`call`的机器码，因此强行插入的`E8`会把后面对整个机器码的分析全部打乱，使得IDA无法成功把它识别成一个函数。

但为什么在执行程序的时候又能成功运行呢？我们仔细看看前面，首先我们通过`xor eax, eax`将eax寄存器清零，然后我们又通过`test eax, eax`来检测eax是否为0，因此这必然是一个永真条件，虽然后面是个条件跳转，但是无论如何跳转必然实现，跳过了`E8`后程序自然会正常执行了。只是IDA做不到这么智能，没有分析出来。要解决它其实很简单，只要nop掉`E8`就行了，具体方法是在Hex View窗口把`E8`改成`90`。汇编指令`nop`表示什么都不做，常常用来作为某种填充。