作者：　中南大学 信安1502班 申卓祥
QQ：　1554849015

AFCTF 复赛Reverse解题报告

RE1：芝麻开门

题目介绍与预备知识

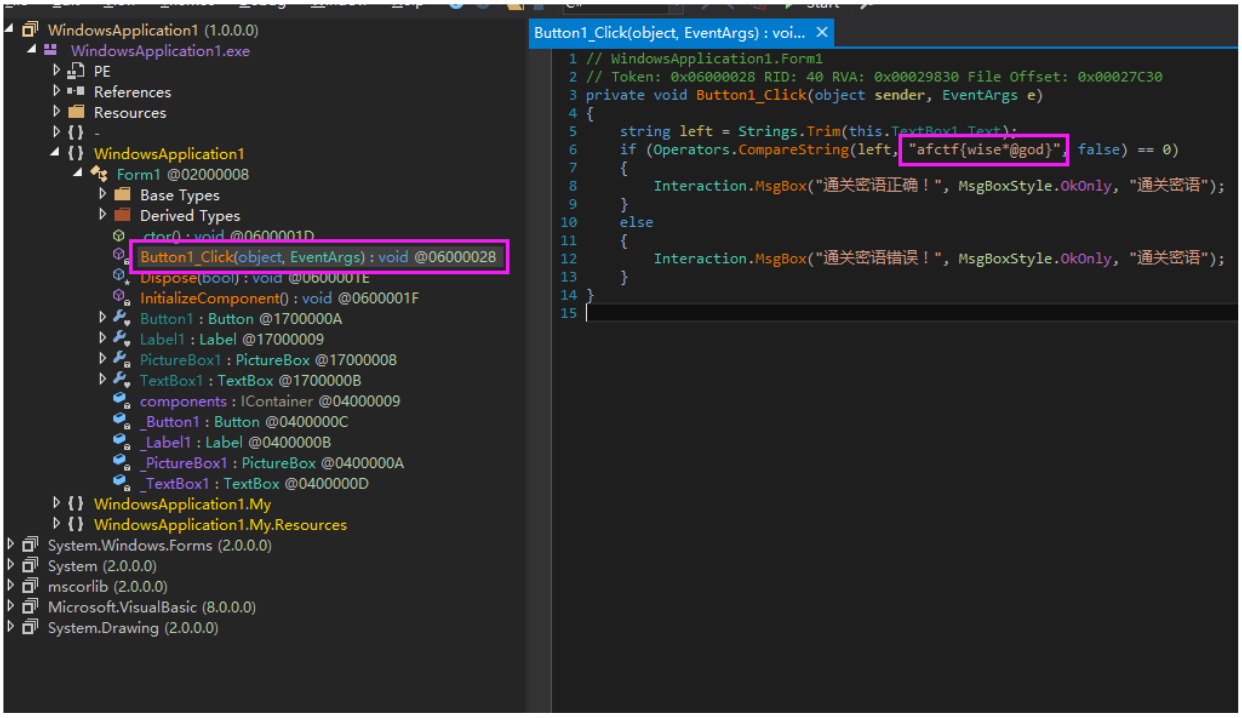
本题的定位是签到题，因此题目难度不大，主要是让大家了解.NET逆向，了解对于.NET逆向来说有比传统的C/C++程序逆向更好的工具。对于.NET逆向来说，用IDA自然是可以的，但是由于.NET逆向的特殊性，我们可以用一个更好的工具`dnSpy`（吾爱破解工具包里面自带），让我们的逆向过程更加轻松。

解题过程

程序的运行模式很常规，在“通关密语”栏输入一段口令，点击“开启”按钮，程序就会进行判断，如果正确就提示正确，如果错误就提示错误。我们可以很自然地推断出：

- 正确的口令即是flag
- 关键判断函数应在“开启”按钮的触发函数里面

因此我们现在打开dnSpy，把程序拖进去，然后仔细找找，找一个跟按钮触发函数比较像的名字，因为这个程序只有一个按钮，而且规模不大，所以很好找：



我们找到了一个名叫`Button1_Click`的函数，随之而来右边的判断逻辑也就清楚了，就是简单的明文对比，因此flag就出来了。

因为这题的flag是以明文形式静态地写进程序里面的，因此有些人用杂项的方法搜索unicode字符串也是可以找到flag的。但是希望大家能够以这道题了解.NET逆向。

flag

`afctf{wise*@god}`

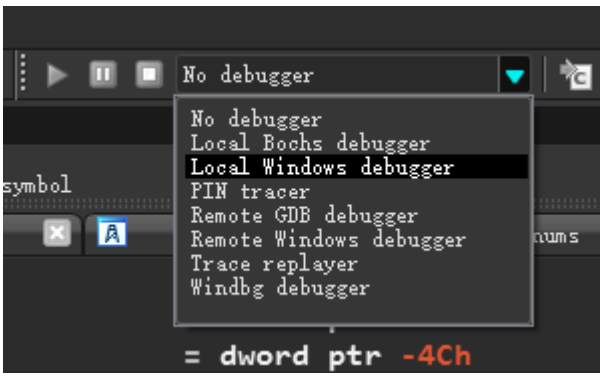
RE2：smc

题目介绍与预备知识

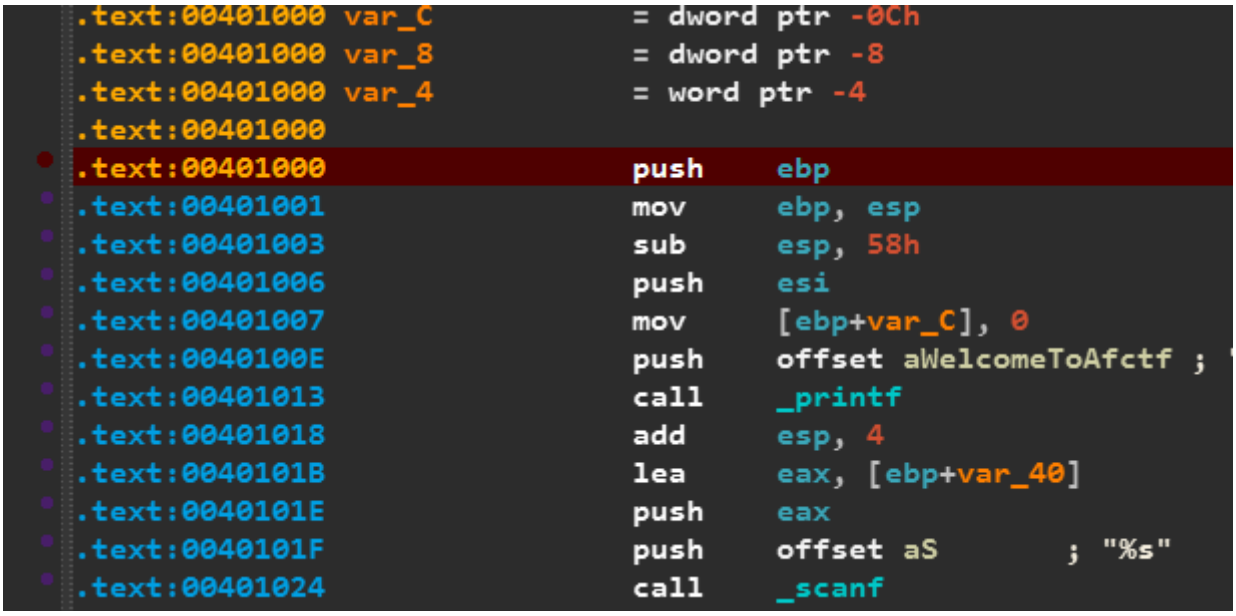
`SMC`(Self-Modifying Code)，意思为“自修改代码”，也就是说，程序执行的代码可能随着程序的运行而改变。因此当我们用IDA做静态分析而不将程序运行起来调试的时候，分析的代码要么就是无法分析，要么就是分析出的逻辑是错误的。而这道题属于SMC中“自解密”的情况，当我们静态分析的时候，程序执行的关键指令被加密了，此时用IDA分析会看到一堆二进制数据，IDA无法将它们解释成机器指令，但当程序运行起来后，程序会先执行一段解密代码，将关键代码中的加密指令还原，再执行解密后的关键代码，因此程序真正运行起来的时候并不会出现异常，其实这种思路跟程序“壳”类似，有兴趣的可以了解一下加壳与脱壳。

既然自解密需要将程序运行起来才能解密，那么我们也就不能只停留在静态分析上面了，接下来我简单讲讲IDA的动态调试功能。

我们先选择一个调试器，在IDA顶部有个下拉栏，我们选择`Local Windows debugger`，如图所示：



点选了之后，左边那个形似播放按钮的三角形会变绿，这时候先别急着点它开始调试，我们先在主函数的第一条指令下一个断点，可以点击指令左边的那个小圆点，也可以选中指令按F2：



这个时候就可以点那个三角形开始调试了，这个时候程序会自动断在刚刚下的那条指令上面。

注意：有的同学可能在点击三角形开启调试后遇到Create Process Fail这个错误，你需要把调试的程序放在一个全英文的路径中，再拖进IDA调试，据我测试，放桌面就可以。

下面列举几个调试常用的快捷键：

- F5：反编译成伪C代码（调试也可以在F5中的伪代码窗口操作）
- F8：单步执行，遇到函数调用不进入函数，直接视为单个指令执行
- F7：单步执行，遇到函数调用会跟进这个函数内部
- F4：执行到此处，选中某条指令按F4，程序会直接执行到该处，通常可用于跳出一个循环，但是要注意，如果程序永远不会执行到那条指令，此时按F4后程序会跑飞。所以在按F4时请务必确保程序一定会执行到那条指令
- F9：执行代码，遇到断点则停，没遇到就不停
- F2：在选中位置下断点

这道题还要求你有一定的汇编功底，因为在调试的过程中很多时候你是要直接面对汇编的，而且这道题有些算法甚至是直接用汇编写的，所以大家在平时一定要把汇编的基础打扎实，不要求熟练编写，但是至少要会读。

解题过程

在主函数第一条指令下断点，开启调试，待断下来后，按F5反编译成伪C代码，得到如下代码：

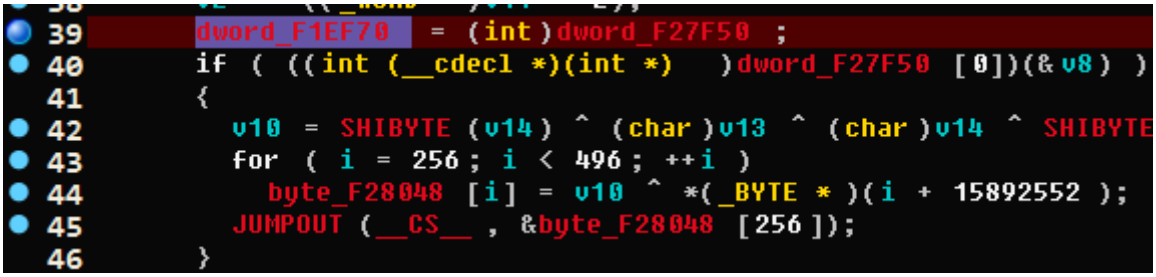
```
int wmain()
{
  char *v0; // eax
  int v1; // ST24_4
  __int16 v2; // ST28_2
  char v4; // [esp+1Ch] [ebp-40h]
  char v5; // [esp+36h] [ebp-26h]
  int v6; // [esp+38h] [ebp-24h]
  __int16 v7; // [esp+3Ch] [ebp-20h]
  int v8; // [esp+40h] [ebp-1Ch]
  __int16 v9; // [esp+44h] [ebp-18h]
  int v10; // [esp+48h] [ebp-14h]
  char *v11; // [esp+4Ch] [ebp-10h]
  int i; // [esp+50h] [ebp-Ch]
  int v13; // [esp+54h] [ebp-8h]
  __int16 v14; // [esp+58h] [ebp-4h]

  i = 0;
  printf("Welcome to AFCTF,I hope you have a good time!\n");
  scanf("%s", &v4);
  if ( strlen(&v4) == 27 )
  {
    if ( !strncmp(&v4, "afctf{", 6u) && v5 == 125 )
    {
      v5 = 0;
      v0 = strtok(&v4, "_");
      v11 = v0;
      v0 += 6;
      v8 = *(_DWORD *)v0;
      v9 = *((_WORD *)v0 + 2);
      v13 = *(_DWORD *)(v11 + 6);
      v14 = *((_WORD *)v11 + 5);
      v11 = strtok(0, "_");
      v6 = *(_DWORD *)v11;
      v7 = *((_WORD *)v11 + 2);
      v11 = strtok(0, "_");
      v1 = *(_DWORD *)v11;
      v2 = *((_WORD *)v11 + 2);
      dword_F1EF70 = (int)dword_F27F50;
      if ( ((int (__cdecl *)(int *))dword_F27F50[0])(&v8) )
      {
        v10 = SHIBYTE(v14) ^ (char)v13 ^ (char)v14 ^ SHIBYTE(v13) ^ SBYTE
2(v13) ^ SBYTE1(v13);
        for ( i = 256; i < 496; ++i )
          byte_F28048[i] = v10 ^ *(_BYTE *)(i + 15892552);
        JUMPOUT(__CS__, &byte_F28048[256]);
      }
      printf("Wrong\n");
    }
```

```
    else
    {
      printf("Wrong\n");
    }
  }
  else
  {
    printf("Wrong\n");
  }
  return 0;
}
```

程序的主要逻辑是，先判断输入的flag符不符合规定的长度和格式，然后把afctf{...}括号中间的内容取出来，以下划线字符为界，切割成三段，且我们可以发现，每一段都由一个`DWORD`类型和一个`WORD`类型组成，共6个字节，所以每段是6个字符，切割的这个操作主要由`strtok`函数完成，这个函数的使用方法大家可以上网查找。接下来是一个if判断，看似好像是执行一段验证函数判断flag符不符合要求，参数就是切割的第一段字串。但是当我们点击进去这个所谓的“函数”时，发现是一大堆无意义的数据：

```
.data:00F27F50 dword_F27F50 dd 0F761C681h, 0E843AEh, 5E000000h,
81FF3357h, 0C0FFh, 8A147F00h, 80253E5Ch
.data:00F27F50                                         ; DATA XREF: _wmai
n+122↑o
.data:00F27F50 dd 5C8873F3h, 0C783253Eh, 65E8EB01h, 5F797361h, 90909090h,
 0F09FF826h
.data:00F27F50 dd 0B5256B9Fh, 0B50F8336h, 0B5128236h, 0B56B8136h, 0B56980
36h, 0B53A8736h
.data:00F27F50 dd 0B54A8636h, 0B53F8B36h, 0B54F8A36h, 0B5618936h, 0B53788
36h, 0B52F8F36h
.data:00F27F50 dd 0B4598E36h, 73739F36h, 36B47373h, 7373739Bh,
0F87A9873h, 0B3F09B36h
.data:00F27F50 dd 9B36FA72h, 759B0EF0h, 3EF83A0Eh, 9B3E707Bh, 0F87ACD7Ch,
 0B3F09B36h
.data:00F27F50 dd 75CDEA71h, 84737373h, 27CD7C8Dh, 99F08366h, 0F8B94061h,
 36707B36h, 0F87BFB9Bh
.data:00F27F50 dd 3E707B3Eh, 62CD7C9Bh, 7C9B36F8h, 8B763FCDh, 7A06A248h,
0F09F26F8h, 26FA72B1h
.data:00F27F50 dd 0F0DB989Fh, 6759F0Eh, 7372CB74h, 71987373h, 0F82DB340h,
 0E3B02E96h
......
```

这就是我们的自解密，我们先断到if判断的前面那行代码，按下F9，中间因为执行了`scanf`函数，故你需要在弹出的终端窗口输入我们要的flag，我们就索性随便输入个`afctf{123456_abcdef_777777}`



接着按F7跟进那个验证函数，IDA会弹出一个提示，点击Yes就行了，接下来会出现这样的汇编代码：

```
.data:00F27F50 add     esi, 43AEF761h
.data:00F27F56 call    $+5
.data:00F27F5B pop     esi
.data:00F27F5C push    edi
.data:00F27F5D xor     edi, edi
.data:00F27F5F
.data:00F27F5F loc_F27F5F:       ; CODE XREF: .data:00F27F75↓j
.data:00F27F5F cmp     edi, 0C0h
.data:00F27F65 jg      short loc_F27F7B
.data:00F27F67 mov     bl, [esi+edi+25h]
.data:00F27F6B xor     bl, 73h
.data:00F27F6E mov     [esi+edi+25h], bl
.data:00F27F72 add     edi, 1
.data:00F27F75 jmp     short loc_F27F5F
.data:00F27F75 ; ---------------------------------------------
.data:00F27F77 db  65h ; e
.data:00F27F78 db  61h ; a
.data:00F27F79 db  73h ; s
.data:00F27F7A db  79h ; y
.data:00F27F7B ; ---------------------------------------------
.data:00F27F7B loc_F27F7B:       ; CODE XREF: .data:00F27F65↑j
.data:00F27F7B pop     edi
.data:00F27F7C nop
.data:00F27F7D nop
.data:00F27F7E nop
.data:00F27F7F nop
.data:00F27F80 db      26h
.data:00F27F80 clc
.data:00F27F82 lahf
......
```

看懂这段代码需要一定的汇编基础，如果你看不懂，建议你先去学习汇编再继续搞懂此题。这里我提几点，首先是`call $+5; pop esi`这两句汇编指令很有代表性，它可以获得`pop esi`这句指令的地址，在这道题中也就是`00F27F5B`，并将其存入esi寄存器中。汇编程序员很多时候需要获得当前指令的地址，就可以通过这种方法获得。至于为什么这两句指令可以用来获得当前指令地址，不妨作为一个小问题留给汇编学习者们。

接下来看`00F27F5F`到`00F27F75`这一段，edi用做计数器，相当于循环控制变量，第一句的`cmp edi, 0C0h`说明此段循环范围为0到0xC0，而`esi + 25h`等于`00F27F80`。因此这段汇编的意思是将`00F27F80`地址开始，长度为0xC0的所有数据全部异或0x73，从而实现对关键代码的解密。我们

分析出这是一个解密段以后就可以直接F4跳过到`00F27F7F`。这时候我们选中从`00F27F80`地址开始，长度为0xC0的代码段，按C键重新分析，分析完后在`00F27F80`位置按P键建立函数，此时我们就可以按F5反汇编成伪C代码了：

```
BOOL __cdecl sub_F27F80(int a1)
{
  signed int i; // [esp+4h] [ebp-18h]
  int v3; // [esp+8h] [ebp-14h]
  char v4; // [esp+Ch] [ebp-10h]
  char v5; // [esp+Dh] [ebp-Fh]
  char v6; // [esp+Eh] [ebp-Eh]
  char v7; // [esp+Fh] [ebp-Dh]
  char v8; // [esp+10h] [ebp-Ch]
  char v9; // [esp+11h] [ebp-Bh]
  char v10; // [esp+14h] [ebp-8h]
  char v11; // [esp+15h] [ebp-7h]
  char v12; // [esp+16h] [ebp-6h]
  char v13; // [esp+17h] [ebp-5h]
  char v14; // [esp+18h] [ebp-4h]
  char v15; // [esp+19h] [ebp-3h]

  v4 = 124;
  v5 = 97;
  v6 = 24;
  v7 = 26;
  v8 = 73;
  v9 = 57;
  v10 = 76;
  v11 = 60;
  v12 = 18;
  v13 = 68;
  v14 = 92;
  v15 = 42;
  v3 = 0;
  for ( i = 0; i < 6; ++i )
  {
    *(_BYTE *)(i + a1) ^= *(&v4 + (i + 2) % 6) - 18;
    if ( *(char *)(i + a1) == *(&v10 + i) )
      ++v3;
  }
  return v3 == 6;
}
```

算法逻辑很简单了，我就不多说了，最后得出第一段的六个字符为`J4%c6e`，解题代码如下：

```
arr1 = [0x7c, 0x61, 0x18, 0x1A, 0x49, 0x39]
arr2 = [0x4C, 0x3C, 0x12, 0x44, 0x5c, 0x2a]

s1 = ""
```

```
for i in range(0, 6):
    s1 += chr(arr2[i] ^ (arr1[(i + 2) % 6] - 18))

print s1
```

下面进入第二段，会发现直接输出wrong结束了，这时我们重新启动程序调试，这次我们输入的内容把第一段改成正确的输入，例如`afctf{J4%c6e_123456_abcdef}`,重复上述步骤，此时就会跳过输出wrong的代码段，执行以下代码：

```
.text:00F0114F movsx   ecx, byte ptr [ebp+var_8+1]
.text:00F01153 movsx   edx, byte ptr [ebp+var_8+2]
.text:00F01157 xor     ecx, edx
.text:00F01159 movsx   eax, byte ptr [ebp+var_8+3]
.text:00F0115D xor     ecx, eax
.text:00F0115F movsx   edx, byte ptr [ebp+var_4]
.text:00F01163 xor     ecx, edx
.text:00F01165 movsx   eax, byte ptr [ebp+var_8]
.text:00F01169 xor     ecx, eax
.text:00F0116B movsx   edx, byte ptr [ebp+var_4+1]
.text:00F0116F xor     ecx, edx
.text:00F01171 mov     [ebp+var_14], ecx
.text:00F01174 mov     [ebp+var_C], 100h
.text:00F0117B jmp     short loc_F01186
.text:00F0117D ; ---------------------------------------------
.text:00F0117D loc_F0117D:             ; CODE XREF: _wmain+1A5↓j
.text:00F0117D mov     eax, [ebp+var_C]
.text:00F01180 add     eax, 1
.text:00F01183 mov     [ebp+var_C], eax
.text:00F01186
.text:00F01186 loc_F01186:             ; CODE XREF: _wmain+17B↑j
.text:00F01186 cmp     [ebp+var_C], 1F0h
.text:00F0118D jge     short loc_F011A7
.text:00F0118F mov     ecx, [ebp+var_C]
.text:00F01192 movsx   edx, byte ptr [ecx+0F28048h]
.text:00F01199 xor     edx, [ebp+var_14]
.text:00F0119C mov     eax, [ebp+var_C]
.text:00F0119F mov     byte_F28048[eax], dl
.text:00F011A5 jmp     short loc_F0117D
.text:00F011A7 ; ---------------------------------------------
.text:00F011A7
.text:00F011A7 loc_F011A7:             ; CODE XREF: _wmain+18D↑j
.text:00F011A7 push    eax
.text:00F011A8 push    ecx
.text:00F011A9 push    esi
.text:00F011AA call    $+5
.text:00F011AF pop     ecx
.text:00F011B0 lea     eax, byte_F28048
.text:00F011B6 lea     esi, [esp+68h+var_4C]
.text:00F011BA add     eax, 100h
.text:00F011BF jmp     eax
```

```
......
```

这段代码的含义是，将第一段得出的J4%c6e这六个字符一起异或得到一个key（存放在`[ebp + var_14]`里面），然后在`F28048h + 100h = F28148h`处存放着一段加密代码，长度为`1F0h - 100h = F0h`，将这段加密代码与key异或后还原出关键代码，最后通过`jmp eax`跳到关键代码。可见，如果你第一段没算对，那么算出的key也会是错的，这样解密出的代码也会是错的，之所以这样设计是因为有些有经验的逆向工程师可以在调试器中强行改变标志寄存器的值控制跳转，使得第一关没通过可以强行打开第二关，这种设计就避免了这一点。

直接跳过这段自解密代码进入关键代码：

```
.data:00F28148 mov     esi, [esi]
.data:00F2814A xor     edi, edi
.data:00F2814C push    ecx
.data:00F2814D
.data:00F2814D loc_F2814D:       ; CODE XREF: .data:00F2816F↓j
.data:00F2814D cmp     edi, 6
.data:00F28150 jge     short loc_F28173
.data:00F28152 xor     ecx, ecx
.data:00F28154 mov     cl, [esi+edi]
.data:00F28157 add     cl, 1
.data:00F2815A ror     cl, 2
.data:00F2815D jmp     short loc_F28171
.data:00F2815D ; ---------------------------------------------
.data:00F2815F db  12h
.data:00F28160 db  34h ; 4
.data:00F28161 db  56h ; V
.data:00F28162 db  78h ; x
.data:00F28163 db  90h
.data:00F28164 db 0ABh
.data:00F28165 ; ---------------------------------------------
.data:00F28165
.data:00F28165 loc_F28165:       ; CODE XREF: .data:loc_F28171↓j
.data:00F28165 xor     cl, [eax+edi+17h]
.data:00F28169 mov     [esi+edi], cl
.data:00F2816C add     edi, 1
.data:00F2816F jmp     short loc_F2814D
.data:00F28171 ; ---------------------------------------------
.data:00F28171
.data:00F28171 loc_F28171:       ; CODE XREF: .data:00F2815D↑j
.data:00F28171 jmp     short loc_F28165
.data:00F28173 ; ---------------------------------------------
.data:00F28173
.data:00F28173 loc_F28173:       ; CODE XREF: .data:00F28150↑j
.data:00F28173 xor     edi, edi
.data:00F28175
.data:00F28175 loc_F28175:       ; CODE XREF: .data:00F281B9↓j
.data:00F28175 cmp     edi, 6
.data:00F28178 jge     short loc_F281BB
.data:00F2817A mov     cl, [esi+edi]
```

```
.data:00F2817D mov     ch, cl
.data:00F2817F and     ch, 0F0h
.data:00F28182 ror     ch, 4
.data:00F28185 and     cl, 0Fh
.data:00F28188 sub     eax, 100h
.data:00F2818D rol     cl, 4
.data:00F28190 push    ebx
.data:00F28191 mov     bl, cl
.data:00F28193 add     bl, ch
.data:00F28195 and     ebx, 0FFh
.data:00F2819B mov     bl, [eax+ebx]
.data:00F2819E jmp     short loc_F281A7
.data:00F2819E ; ---------------------------------------------
.data:00F281A0 db  91h
.data:00F281A1 db  3Eh ; >
.data:00F281A2 db  16h
.data:00F281A3 db  64h ; d
.data:00F281A4 db 0CDh
.data:00F281A5 db  86h
.data:00F281A6 db  90h
.data:00F281A7 ; ---------------------------------------------
.data:00F281A7
.data:00F281A7 loc_F281A7:      ; CODE XREF: .data:00F2819E↑j
.data:00F281A7 cmp     bl, [eax+edi+158h]
.data:00F281AE jnz     short loc_F281C6
.data:00F281B0 pop     ebx
.data:00F281B1 add     edi, 1
.data:00F281B4 add     eax, 100h
.data:00F281B9 jmp     short loc_F28175
.data:00F281BB ; ---------------------------------------------
.data:00F281BB
.data:00F281BB loc_F281BB:      ; CODE XREF: .data:00F28178↑j
.data:00F281BB pop     ecx
.data:00F281BC mov     eax, 1
.data:00F281C1 lea     ecx, [ecx+12h]
.data:00F281C4 jmp     ecx
.data:00F281C6 ; ---------------------------------------------
.data:00F281C6
.data:00F281C6 loc_F281C6:      ; CODE XREF: .data:00F281AE↑j
.data:00F281C6 xor     eax, eax
.data:00F281C8 pop     ecx
.data:00F281C9 pop     ecx
.data:00F281CA add     ecx, 12h
.data:00F281CD jmp     ecx
```

这段代码确实要耐心分析，而且很考验汇编功底和调试能力。在调试的时候有时某段代码没看懂时可以借助调试器的信息去猜某些寄存器和内存空间的含义。像第一句话 `mov esi, [esi]` 我也没看懂，但是我发现执行完这句话后esi寄存器正好指向我们输入的第二段字符串的开头，所以我们基本上可以推定esi此时表示输入的第二段字符串首地址。同样 `xor cl, [eax+edi+17h]` 也搞不懂 `[eax+edi+17h]` 指向的是哪里，但是执行到这条语句时IDA告诉我们它正好指向的是

```
.data:00F2815F db  12h
.data:00F28160 db  34h ; 4
.data:00F28161 db  56h ; V
.data:00F28162 db  78h ; x
.data:00F28163 db  90h
.data:00F28164 db 0ABh
```

这段数据，从而我们也大致可以推断后面的一小段代码多半是将输入的每一个字节异或这张表。这体现了一种黑盒测试的思想，当有的时候我们逆向的程序规模很大或者是逻辑难以看懂的时候，我们把它当成一个黑盒子，只通过观察输入和输出以及部分特征指标去推断功能，这在很多时候是一个高效的办法。

这段代码的逻辑是这样的，首先将输入的6个字节每个都加1并不带符号循环右移（注：汇编指令`ror`）2位，再异或`12h,34h,56h,78h,90h,0ABh`这张表，异或完后把每个字节的高四位和低四位换个位置（比如原来的`0x67`变成`0x76`），最后以每个字节的值为索引做一个查表替换操作，这张表位于`00F28048`处（本代码段中用eax寄存器表示这张表的首地址），长度为100h，数字两两之间互不相同。最后做完查表替换后，与`91h,3Eh,16h,64h,0CDh,86h`这个对比串做对比，如果通过，则成功。这里面的所有步骤皆是可逆操作，只要从对比串一步一步写逆算法回去就行了，得出的字符串为`m967t0`。

重新调试，将前两段正确的字符串输入，重复前面的步骤，进入第三段，得到如下代码：

```
.text:00F011DC push    6                       ; size_t
.text:00F011DE push    offset a50sa            ; "50Sa*^"
.text:00F011E3 lea     ecx, [ebp+var_48]
.text:00F011E6 push    ecx                     ; char *
.text:00F011E7 call    _strncmp
.text:00F011EC add     esp, 0Ch
.text:00F011EF test    eax, eax
.text:00F011F1 jz      short loc_F01202
.text:00F011F3 push    offset aWrong           ; "Wrong\n"
.text:00F011F8 call    _printf
.text:00F011FD add     esp, 4
.text:00F01200 jmp     short loc_F01217
.text:00F01202 ; ---------------------------------------------
.text:00F01202
.text:00F01202 loc_F01202:             ; CODE XREF: _wmain+1F1↑j
.text:00F01202 push    offset aCongYouWin      ; "Cong,You win!"
.text:00F01207 call    _printf
......
```

这段代码就没有任何难度了，直接是调用`strncmp`函数与`50Sa*^`这个字符串做明文对比，正确的话就会输出`Cong,You win!`。

源代码

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

#pragma comment(linker, "/section:.data,RWE")

char shellcode1[] =
"\x81\xc6\x61\xf7\xae\x43\xe8\x0\x0\x0\x0\x5e\x57\x33\xff\x81"
"\xff\xc0\x0\x0\x0\x7f\x14\x8a\x5c\x3e\x25\x80\xf3\x73\x88\x5c"
"\x3e\x25\x83\xc7\x1\xeb\xe8\x65\x61\x73\x79\x5f\x90\x90\x90\x90"
"\x26\xf8\x9f\xf0\x9f\x6b\x25\xb5\x36\x83\xf\xb5\x36\x82\x12\xb5"
"\x36\x81\x6b\xb5\x36\x80\x69\xb5\x36\x87\x3a\xb5\x36\x86\x4a\xb5"
"\x36\x8b\x3f\xb5\x36\x8a\x4f\xb5\x36\x89\x61\xb5\x36\x88\x37\xb5"
"\x36\x8f\x2f\xb5\x36\x8e\x59\xb4\x36\x9f\x73\x73\x73\x73\xb4\x36"
"\x9b\x73\x73\x73\x73\x98\x7a\xf8\x36\x9b\xf0\xb3\x72\xfa\x36\x9b"
"\xf0\xe\x9b\x75\xe\x3a\xf8\x3e\x7b\x70\x3e\x9b\x7c\xcd\x7a\xf8"
"\x36\x9b\xf0\xb3\x71\xea\xcd\x75\x73\x73\x73\x84\x8d\x7c\xcd\x27"
"\x66\x83\xf0\x99\x61\x40\xb9\xf8\x36\x7b\x70\x36\x9b\xfb\x7b\xf8"
"\x3e\x7b\x70\x3e\x9b\x7c\xcd\x62\xf8\x36\x9b\x7c\xcd\x3f\x76\x8b"
"\x48\xa2\x6\x7a\xf8\x26\x9f\xf0\xb1\x72\xfa\x26\x9f\x98\xdb\xf0"
"\xe\x9f\x75\x6\x74\xcb\x72\x73\x73\x73\x98\x71\x40\xb3\x2d\xf8"
"\x96\x2e\xb0\xe3\xe3\xe3\xe3\xe3\xe3\xe3\xe3\xe3\xe3\xe3\xe3\xe3";

char shellcode2[] =
"\xf6\xa3\x5b\x9d\xe0\x95\x98\x68\x8c\x65\xbb\x76\x89\xd4\x9\xfd"
"\xf3\x5c\x3c\x4c\x36\x8e\x4d\xc4\x80\x44\xd6\xa9\x1\x32\x77\x29"
"\x90\xbc\xc0\xa8\xd8\xf9\xe1\x1d\xe4\x67\x7d\x2a\x2c\x59\x9e\x3d"
"\x7a\x34\x11\x43\x74\xd1\x62\x60\x2\x4b\xae\x99\x57\xc6\x73\xb0"
"\x33\x18\x2b\xfe\xb9\x85\xb6\xd9\xde\x7b\xcf\x4f\xb3\xd5\x8\x7c"
"\xa\x71\x12\x6\x37\xff\x7f\xb7\x46\x42\x25\xc9\xd0\x50\x52\xce"
"\xbd\x6c\xe5\x6f\xa5\x15\xed\x64\xf0\x23\x35\xe7\xc\x61\xa4\xd7"
"\x51\x75\x9a\xf2\x1e\xeb\x58\xf1\x94\xc3\x2f\x56\xf7\xe6\x86\x47"
"\xfb\x83\x5e\xcc\x21\x4a\x24\x7\x1c\x8a\x5a\x17\x1b\xda\xec\x38"
"\xe\x7e\xb4\x48\x88\xf4\xb8\x27\x91\x0\x13\x97\xbe\x53\xc2\xe8"
"\xea\x1a\xe9\x2d\x14\xb\xbf\xb5\x40\x79\xd2\x3e\x19\x5d\xf8\x69"
"\x39\x5f\xdb\xfa\xb2\x8b\x6e\xa2\xdf\x16\xe2\x63\xb1\x20\xcb\xba"
"\xee\x8d\xaa\xc8\xc7\xc5\x5\x66\x6d\x3a\x45\x72\xd\xca\x84\x4e"
"\xf5\x31\x6b\x92\xdc\xdd\x9c\x3f\x55\x96\xa1\x9f\xcd\x9b\xe3\xa0"
"\xa7\xfc\xc1\x78\x10\x2e\x82\x8f\x30\x54\x4\xac\x41\x93\xd3\x3b"
"\xef\x3\x81\x70\xa6\x1f\x22\x26\x28\x6a\xab\x87\xad\x49\xf\xaf"

"\xe0\x5d\x58\x94\x3a\xe8\x94\x6d\x16\x4a\x58\xa2\xe1\x67\x55\xeb"
"\xaa\x6a\xab\xa2\x69\x80\x79\x79\x5f\x3d\x13\xfb\xc0\x59\x27\x53"
"\x7c\xe3\x67\x55\xe8\xac\x6a\x80\xb7\x80\x99\x58\x94\xe8\x94\x6d"
"\x16\x2a\xe1\x67\x55\xe1\x82\xeb\x8e\x9b\xab\xa6\x6f\xeb\x8a\x64"
"\x46\x6b\x6a\x6b\x6b\xab\xaa\x6f\x38\xe1\xb2\x69\xb6\xea\x88\x94"
"\x6b\x6b\x6b\xe1\x77\x73\x80\x6c\xfa\x55\x7d\xf\xa6\xed\xfb\x51"
"\xf7\x53\x33\x6a\x6b\x6b\x1e\x7d\x30\xe8\xac\x6a\x6e\x6b\x6a\x6b"
"\x6b\x80\xd1\x32\xd3\x6a\x6b\x6b\x6b\xe6\x22\x79\x94\x8a\x58\xab"
"\x32\x32\xe8\xaa\x79\x94\x8a\xfb\xfb\xfb\xfb\xfb\xfb\xfb\xfb\xfb";
```

```
int (*Check )(char*);
int (*Check2)(char*,char*,char*);


//afctf{J4%c6e_m967t0_50Sa*^}
int _tmain(int argc, _TCHAR* argv[])
{
    char flag[28];
    char flag1[6];
    char flag11[6];
    char flag2[6];
    char flag3[6];
    char *tokenPtr;
    int key;
    int i=0;
    printf("Welcome to AFCTF,I hope you have a good time!\n");
    scanf("%s",flag);
    do{
        if(strlen(flag) !=27){
            printf("Wrong\n");
            break;
        }
        if(strncmp(flag,"afctf{",6) != 0 || flag[26] != 125){
            printf("Wrong\n");
            break;
        }
        flag[26] = '\0';
        tokenPtr = strtok(flag,"_");
        memcpy(flag1,tokenPtr+6,6);
        memcpy(flag11,tokenPtr+6,6);
        tokenPtr = strtok(NULL,"_");
        memcpy(flag2,tokenPtr,6);
        tokenPtr = strtok(NULL,"_");
        memcpy(flag3,tokenPtr,6);

        Check = (int(*)(char *))&shellcode1;
        if (!Check(flag1)){
            printf("Wrong\n");
            break;
        }
        key =
flag11[1]^flag11[2]^flag11[3]^flag11[4]^flag11[0]^flag11[5];

        for(i=0x100;i< 0x1f0;i++){
            shellcode2[i] = shellcode2[i]^key;
        }
        __asm{
            push eax;
            push ecx;
            push esi
            call $+5;
            pop ecx;
```

```
            lea eax, shellcode2;
            lea esi,[esp+0x1c];
            add eax,0x100
            jmp eax;
            pop esi;
            pop ecx;
            mov  key,eax;
            pop eax;
        }
        if(strncmp(flag3,"50Sa*^",6)){
            printf("Wrong\n");
            break;
        }
        printf("Cong,You win!");

    }while(0);
    return 0;
}
```

flag

`afctf{J4%c6e_m967t0_50Sa*^}`

RE3：玩转密码学

题目介绍与预备知识

这道题考验的是同学们对成熟密码算法的感知能力，利用成熟的密码算法对软件进行保护是一个可靠又省事的办法。一般在CTF题目中，成熟密码算法本身往往不会做什么改动，改动的一般在外围，可能是加了个替换、换位、拼接等组合操作。这类题要求大家对基本的密码算法的程序写法要熟悉，比如说`MD5，SHA-1，SHA-256，DES，AES，3DES，RC4，IDEA，国密SM4`等。不要求完全理解里面的算法，但是要熟悉这些算法的重要特征，比方说MD5算法的最开始会赋值4个32位寄存器`0x67452301，0xEFCDAB89，0x98BADCFE，0x10325476`，这就是关键特征。然后确定是某种成熟密码算法变形后，就可以从网上找段已有的代码做对比，找出改动的点。

解题过程

这是一个运行在Linux环境下的64位ELF文件，有UPX壳，故先需要用`upx -d re3`命令把壳脱掉。脱完后，拖入IDA x64，定位到主函数后F5反编译可得：

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  __int64 result; // rax
```

```
unsigned __int64 v4; // kr08_8
char *v5; // ST18_8
char *v6; // ST20_8
char *v7; // ST28_8
char *v8; // ST30_8
bool v9; // cf
bool v10; // zf
signed __int64 v11; // rcx
char *v12; // rsi
const char *v13; // rdi
signed int i; // [rsp+38h] [rbp-88h]
char v15[48]; // [rsp+40h] [rbp-80h]
char v16; // [rsp+70h] [rbp-50h]
__int16 v17; // [rsp+76h] [rbp-4Ah]
char dest[40]; // [rsp+90h] [rbp-30h]
unsigned __int64 v19; // [rsp+B8h] [rbp-8h]

v19 = __readfsqword(0x28u);
puts("Please input flag:");
__isoc99_scanf("%s", &v16);
if ( (unsigned int)sub_401773(&v16) )
{
  puts("Format Error !!");
  result = 0LL;
}
else
{
  v4 = strlen(&v16) + 1;
  strncpy(dest, (const char *)&v17, (signed int)v4 - 8);
  dest[(signed int)v4 - 8] = 0;
  v5 = strtok(dest, "_");
  v6 = strtok(0LL, "_");
  v7 = strtok(0LL, "_");
  v8 = strtok(0LL, "_");
  sub_4016DC(v5, 0LL);
  sub_4016DC(v6, 4LL);
  sub_4016DC(v7, 8LL);
  sub_4016DC(v8, 12LL);
  for ( i = 0; ; ++i )
  {
    v9 = (unsigned int)i < 0xF;
    v10 = i == 15;
    if ( i > 15 )
      break;
    sprintf(&v15[2 * i], "%02x", (unsigned __int8)byte_6030B0[i]);
  }
  v11 = 33LL;
  v12 = v15;
  v13 = "f58e6a50d0dbe91515913c1002c77002";
  do
  {
    if ( !v11 )
```

```
        break;
      v9 = (unsigned __int8)*v12 < *v13;
      v10 = *v12++ == *v13++;
      --v11;
    }
    while ( v10 );
    if ( (!v9 && !v10) == v9 )
      puts("Success!! You are right !!");
    else
      puts("You are wrong !!");
    result = 0LL;
  }
  return result;
}
```

跟第二题一样也用了`strtok`做切割，这次切割成了四份，`sub_401773`是个格式判断函数，它除了规定了格式以外，最重要的是规定了每一段长度为4，且全为小写字母，这样就大大缩小了穷举量，使得用爆破方法解此题变成了可能。

`sub_4016DC`就是MD5算法的变形算法，我重点提一下与MD5不同的地方。先跟进这个函数以后，得到：

```
unsigned __int64 __fastcall sub_4016DC(const char *a1, int a2)
{
  char v3; // [rsp+20h] [rbp-60h]
  unsigned __int64 v4; // [rsp+78h] [rbp-8h]

  v4 = __readfsqword(0x28u);
  sub_400714(&v3);
  sub_400796(&v3, a1, (unsigned int)strlen(a1));
  sub_4008E4(&v3, (__int64)&byte_6030B0[a2]);
  return __readfsqword(0x28u) ^ v4;
}
```

而我们从网上对比的代码是：

```
MD5Init(&md5);
MD5Update(&md5,encrypt,strlen((char *)encrypt));
MD5Final(&md5,decrypt);
```

不同点在`MD5Final`这个函数，原始的MD5算法直接传了一个`decrypt`进去，而我们传进去的却是`decrypt[a2]`，而`a2`是通过函数参数传进来的。`sub_4016DC`这个函数总共在主函数中调用了四次，`a2`分别为0，4，8，12，每次间隔为4。我们继续跟进`MD5Final`函数：

```
unsigned __int64 __fastcall sub_4008E4(_DWORD *a1, __int64 a2)
{
  int v2; // eax
  unsigned int v3; // ST1C_4
```

```
  unsigned int v5; // [rsp+18h] [rbp-18h]
  char v6; // [rsp+20h] [rbp-10h]
  unsigned __int64 v7; // [rsp+28h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  v5 = (*a1 >> 3) & 0x3F;
  if ( v5 > 0x37 )
    v2 = 120 - v5;
  else
    v2 = 56 - v5;
  v3 = v2;
  sub_4009B1(&v6, a1, 8LL);
  sub_400796(a1, &unk_603060, v3);
  sub_400796(a1, &v6, 8LL);
  sub_4009B1(a2, a1 + 2, 4LL);
  return __readfsqword(0x28u) ^ v7;
}
```

而网上对应的代码为：

```
void MD5Final(MD5_CTX *context,unsigned char digest[16])
{
    unsigned int index = 0,padlen = 0;
    unsigned char bits[8];
    index = (context->count[0] >> 3) & 0x3F;
    padlen = (index < 56)?(56-index):(120-index);
    MD5Encode(bits,context->count,8);
    MD5Update(context,PADDING,padlen);
    MD5Update(context,bits,8);
    MD5Encode(digest,context->state,16);
}
```

关注最后一个`MD5Encode`函数，我们的题目只传了一个4进去，而原始的MD5算法传的是16。再关注到这个函数把`digest`数组也传进去了，这个数组就是上个函数传进来的`decrypt[a2]`。如果我们稍微了解一点MD5算法的话就知道这一部是编码操作，就是将计算的结果送入`digest`数组，第三的参数就是长度。正常的MD5长度是16字节，那么自然第三个参数是16，而我们只取了4个字节，也就是说算出的MD5值只有开头的4个字节进入了`digest`，同时MD5调用了4次，每次都将`digest`起始指针后挪4位。结合这些信息我们终于知道了改动在哪里了，就是每次计算一个消息的MD5时只取MD5值的前4个字节，那么四个消息都取MD5值的前4个字节正好拼接在一起组成16个字节，跟`f58e6a50d0dbe91515913c1002c77002`这个形似MD5值进行比较。但是我们解的时候必须也要按4个字节这样拆开`f58e6a50`，`d0dbe915`，`15913c10`，`02c77002`。然后爆破就可以出结果了，解题代码如下：

```
import hashlib
import sys

for c1 in range(0x61, 0x7b):
    for c2 in range(0x61, 0x7b):
```

```
        for c3 in range(0x61, 0x7b):
            for c4 in range(0x61, 0x7b):
                m = hashlib.md5()
                m.update(chr(c1) + chr(c2) + chr(c3) + chr(c4))
                if m.hexdigest()[0:8] == "f58e6a50":
                    print chr(c1) + chr(c2) + chr(c3) + chr(c4) + "  -->
 f58e6a50"
                if m.hexdigest()[0:8] == "d0dbe915":
                    print chr(c1) + chr(c2) + chr(c3) + chr(c4) + "  -->
 d0dbe915"
                if m.hexdigest()[0:8] == "15913c10":
                    print chr(c1) + chr(c2) + chr(c3) + chr(c4) + "  -->
 15913c10"
                if m.hexdigest()[0:8] == "02c77002":
                    print chr(c1) + chr(c2) + chr(c3) + chr(c4) + "  -->
 02c77002"
```

运行结果如下：

```
> python solution.py
dead  -->  f58e6a50
live  -->  d0dbe915
oops  -->  02c77002
zoom  -->  15913c10
```

源代码

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

unsigned char dest[16];

typedef struct
{
    unsigned int count[2];
    unsigned int state[4];
    unsigned char buffer[64];
}MD5_CTX;


#define F(x,y,z) ((x & y) | (~x & z))
#define G(x,y,z) ((x & z) | (y & ~z))
#define H(x,y,z) (x^y^z)
#define I(x,y,z) (y ^ (x | ~z))
#define ROTATE_LEFT(x,n) ((x << n) | (x >> (32-n)))
#define FF(a,b,c,d,x,s,ac) \
```

```
            { \
            a += F(b,c,d) + x + ac; \
            a = ROTATE_LEFT(a,s); \
            a += b; \
            }
#define GG(a,b,c,d,x,s,ac) \
            { \
            a += G(b,c,d) + x + ac; \
            a = ROTATE_LEFT(a,s); \
            a += b; \
            }
#define HH(a,b,c,d,x,s,ac) \
            { \
            a += H(b,c,d) + x + ac; \
            a = ROTATE_LEFT(a,s); \
            a += b; \
            }
#define II(a,b,c,d,x,s,ac) \
            { \
            a += I(b,c,d) + x + ac; \
            a = ROTATE_LEFT(a,s); \
            a += b; \
            }
void MD5Init(MD5_CTX *context);
void MD5Update(MD5_CTX *context,unsigned char *input,unsigned int inputle
n);
void MD5Final(MD5_CTX *context,unsigned char digest[16]);
void MD5Transform(unsigned int state[4],unsigned char block[64]);
void MD5Encode(unsigned char *output,unsigned int *input,unsigned int le
n);
void MD5Decode(unsigned int *output,unsigned char *input,unsigned int le
n);
void MD5(char *text, int num);
int checkinput(char *flag);
int judge(char *s);

unsigned char PADDING[]={0x80,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

void MD5Init(MD5_CTX *context)
{
     context->count[0] = 0;
     context->count[1] = 0;
     context->state[0] = 0x32107654;
     context->state[1] = 0xBA98FEDC;
     context->state[2] = 0xCDEF89AB;
     context->state[3] = 0x45670123;
     for(int i = 0; i < 4; i++)
        context->state[i] ^= 0x55555555;
}
```

```
void MD5Update(MD5_CTX *context,unsigned char *input,unsigned int inputle
n)
{
    unsigned int i = 0,index = 0,partlen = 0;
    index = (context->count[0] >> 3) & 0x3F;
    partlen = 64 - index;
    context->count[0] += inputlen << 3;
    if(context->count[0] < (inputlen << 3))
       context->count[1]++;
    context->count[1] += inputlen >> 29;

    if(inputlen >= partlen)
    {
       memcpy(&context->buffer[index],input,partlen);
       MD5Transform(context->state,context->buffer);
       for(i = partlen;i+64 <= inputlen;i+=64)
           MD5Transform(context->state,&input[i]);
       index = 0;
    }
    else
    {
        i = 0;
    }
    memcpy(&context->buffer[index],&input[i],inputlen-i);
}
void MD5Final(MD5_CTX *context,unsigned char *digest)
{
    unsigned int index = 0,padlen = 0;
    unsigned char bits[8];
    index = (context->count[0] >> 3) & 0x3F;
    padlen = (index < 56) ? (56-index):(120-index);
    MD5Encode(bits,context->count,8);
    MD5Update(context,PADDING,padlen);
    MD5Update(context,bits,8);
    MD5Encode(digest,context->state,4);
}
void MD5Encode(unsigned char *output,unsigned int *input,unsigned int le
n)
{
    unsigned int i = 0,j = 0;
    while(j < len)
    {
         output[j] = input[i] & 0xFF;
         output[j+1] = (input[i] >> 8) & 0xFF;
         output[j+2] = (input[i] >> 16) & 0xFF;
         output[j+3] = (input[i] >> 24) & 0xFF;
         i++;
         j+=4;
    }
}
void MD5Decode(unsigned int *output,unsigned char *input,unsigned int le
n)
```

```
{
      unsigned int i = 0,j = 0;
      while(j < len)
      {
            output[i] = (input[j]) |
                        (input[j+1] << 8) |
                        (input[j+2] << 16) |
                        (input[j+3] << 24);
            i++;
            j+=4;
      }
}
void MD5Transform(unsigned int state[4],unsigned char block[64])
{
      unsigned int a = state[0];
      unsigned int b = state[1];
      unsigned int c = state[2];
      unsigned int d = state[3];
      unsigned int x[64];
      MD5Decode(x,block,64);
      FF(a, b, c, d, x[ 0], 7, 0xd76aa478); /* 1 */
 FF(d, a, b, c, x[ 1], 12, 0xe8c7b756); /* 2 */
 FF(c, d, a, b, x[ 2], 17, 0x242070db); /* 3 */
 FF(b, c, d, a, x[ 3], 22, 0xc1bdceee); /* 4 */
 FF(a, b, c, d, x[ 4], 7, 0xf57c0faf); /* 5 */
 FF(d, a, b, c, x[ 5], 12, 0x4787c62a); /* 6 */
 FF(c, d, a, b, x[ 6], 17, 0xa8304613); /* 7 */
 FF(b, c, d, a, x[ 7], 22, 0xfd469501); /* 8 */
 FF(a, b, c, d, x[ 8], 7, 0x698098d8); /* 9 */
 FF(d, a, b, c, x[ 9], 12, 0x8b44f7af); /* 10 */
 FF(c, d, a, b, x[10], 17, 0xffff5bb1); /* 11 */
 FF(b, c, d, a, x[11], 22, 0x895cd7be); /* 12 */
 FF(a, b, c, d, x[12], 7, 0x6b901122); /* 13 */
 FF(d, a, b, c, x[13], 12, 0xfd987193); /* 14 */
 FF(c, d, a, b, x[14], 17, 0xa679438e); /* 15 */
 FF(b, c, d, a, x[15], 22, 0x49b40821); /* 16 */

 /* Round 2 */
 GG(a, b, c, d, x[ 1], 5, 0xf61e2562); /* 17 */
 GG(d, a, b, c, x[ 6], 9, 0xc040b340); /* 18 */
 GG(c, d, a, b, x[11], 14, 0x265e5a51); /* 19 */
 GG(b, c, d, a, x[ 0], 20, 0xe9b6c7aa); /* 20 */
 GG(a, b, c, d, x[ 5], 5, 0xd62f105d); /* 21 */
 GG(d, a, b, c, x[10], 9,  0x2441453); /* 22 */
 GG(c, d, a, b, x[15], 14, 0xd8a1e681); /* 23 */
 GG(b, c, d, a, x[ 4], 20, 0xe7d3fbc8); /* 24 */
 GG(a, b, c, d, x[ 9], 5, 0x21e1cde6); /* 25 */
 GG(d, a, b, c, x[14], 9, 0xc33707d6); /* 26 */
 GG(c, d, a, b, x[ 3], 14, 0xf4d50d87); /* 27 */
 GG(b, c, d, a, x[ 8], 20, 0x455a14ed); /* 28 */
 GG(a, b, c, d, x[13], 5, 0xa9e3e905); /* 29 */
 GG(d, a, b, c, x[ 2], 9, 0xfcefa3f8); /* 30 */
```

```
    GG(c, d, a, b, x[ 7], 14, 0x676f02d9); /* 31 */
    GG(b, c, d, a, x[12], 20, 0x8d2a4c8a); /* 32 */

    /* Round 3 */
    HH(a, b, c, d, x[ 5], 4, 0xfffa3942); /* 33 */
    HH(d, a, b, c, x[ 8], 11, 0x8771f681); /* 34 */
    HH(c, d, a, b, x[11], 16, 0x6d9d6122); /* 35 */
    HH(b, c, d, a, x[14], 23, 0xfde5380c); /* 36 */
    HH(a, b, c, d, x[ 1], 4, 0xa4beea44); /* 37 */
    HH(d, a, b, c, x[ 4], 11, 0x4bdecfa9); /* 38 */
    HH(c, d, a, b, x[ 7], 16, 0xf6bb4b60); /* 39 */
    HH(b, c, d, a, x[10], 23, 0xbebfbc70); /* 40 */
    HH(a, b, c, d, x[13], 4, 0x289b7ec6); /* 41 */
    HH(d, a, b, c, x[ 0], 11, 0xeaa127fa); /* 42 */
    HH(c, d, a, b, x[ 3], 16, 0xd4ef3085); /* 43 */
    HH(b, c, d, a, x[ 6], 23,  0x4881d05); /* 44 */
    HH(a, b, c, d, x[ 9], 4, 0xd9d4d039); /* 45 */
    HH(d, a, b, c, x[12], 11, 0xe6db99e5); /* 46 */
    HH(c, d, a, b, x[15], 16, 0x1fa27cf8); /* 47 */
    HH(b, c, d, a, x[ 2], 23, 0xc4ac5665); /* 48 */

    /* Round 4 */
    II(a, b, c, d, x[ 0], 6, 0xf4292244); /* 49 */
    II(d, a, b, c, x[ 7], 10, 0x432aff97); /* 50 */
    II(c, d, a, b, x[14], 15, 0xab9423a7); /* 51 */
    II(b, c, d, a, x[ 5], 21, 0xfc93a039); /* 52 */
    II(a, b, c, d, x[12], 6, 0x655b59c3); /* 53 */
    II(d, a, b, c, x[ 3], 10, 0x8f0ccc92); /* 54 */
    II(c, d, a, b, x[10], 15, 0xffeff47d); /* 55 */
    II(b, c, d, a, x[ 1], 21, 0x85845dd1); /* 56 */
    II(a, b, c, d, x[ 8], 6, 0x6fa87e4f); /* 57 */
    II(d, a, b, c, x[15], 10, 0xfe2ce6e0); /* 58 */
    II(c, d, a, b, x[ 6], 15, 0xa3014314); /* 59 */
    II(b, c, d, a, x[13], 21, 0x4e0811a1); /* 60 */
    II(a, b, c, d, x[ 4], 6, 0xf7537e82); /* 61 */
    II(d, a, b, c, x[11], 10, 0xbd3af235); /* 62 */
    II(c, d, a, b, x[ 2], 15, 0x2ad7d2bb); /* 63 */
    II(b, c, d, a, x[ 9], 21, 0xeb86d391); /* 64 */
        state[0] += a;
        state[1] += b;
        state[2] += c;
        state[3] += d;
}

void MD5(char *text, int num) {
    MD5_CTX md5;
    MD5Init(&md5);
    MD5Update(&md5,text,strlen(text));
    MD5Final(&md5,dest + num);
}

int checkinput(char *flag) {
```

```
    char afctf[10];
    char content[30];
    char *s1, *s2, *s3, *s4;
    int len = strlen(flag);
    int count = 0;
    strncpy(afctf, flag, 6);
    afctf[6] = 0;
    if(strcmp(afctf, "afctf{"))
        return 1;
    if(flag[len - 1] != '}')
        return 1;
    strncpy(content, flag + 6, len - 7);
    content[len - 7] = 0;
    for(int i = 0; i < strlen(content); i++)
        if(content[i] == '_')
            count++;
    if(count != 3)
        return 1;
    s1 = strtok(content, "_");
    s2 = strtok(NULL, "_");
    s3 = strtok(NULL, "_");
    s4 = strtok(NULL, "_");
    if(judge(s1) || judge(s2) || judge(s3) || judge(s4))
        return 1;
    return 0;
}

int judge(char *s) {
    if(s == NULL || strlen(s) != 4)
        return 1;
    for(int i = 0; i < 4; i++)
        if(s[i] < 0x61 || s[i] > 0x7a)
            return 1;
    return 0;
}

int main() {
    char flag[30];
    char content[30];
    char result[40];
    char *s1, *s2, *s3, *s4;
    int len;
    printf("Please input flag:\n");
    scanf("%s", flag);
    if(checkinput(flag)) {
        printf("Format Error !!\n");
        return 0;
    }
    len = strlen(flag);
    strncpy(content, flag + 6, len - 7);
    content[len - 7] = 0;
    s1 = strtok(content, "_");
```

```
    s2 = strtok(NULL, "_");
    s3 = strtok(NULL, "_");
    s4 = strtok(NULL, "_");
    MD5(s1, 0);
    MD5(s2, 4);
    MD5(s3, 8);
    MD5(s4, 12);
    for(int i = 0; i < 16; i++)
        sprintf(result + 2 * i, "%02x", dest[i]);
    if(strcmp(result, "f58e6a50d0dbe91515913c1002c77002"))
      printf("You are wrong !!\n");
    else
      printf("Success!! You are right !!\n");
    return 0;
}
```

flag

`afctf{dead_live_zoom_oops}`

RE4：JPython

题目介绍与预备知识

这题作为压轴题，很大的原因就在于所需的预备知识比较多，所以如果你能够弄懂这个题，你将能够学到不少知识。

现在一些高级别CTF竞赛中可能会出现一到两题，这种题会涉及到一些绝大多数人都不知道的概念和技术，然而这种概念与技术在已有的基础知识的条件下是可以通过类比、举一反三来快速学会的。因此这种题目也考验选手的自学能力和资料搜集能力。

这道题涉及到对pyc字节码的一个了解。pyc是Python语言的字节码格式，如果你对Java了解的话应该也知道，class文件是Java语言的字节码格式；同样在Android Apk里面，smali文件也是Apk的字节码格式。要解释字节码这个概念，先要解释一下`虚拟机`。我们知道，实际的计算机由于设计标准的不同，对二进制指令的解释方式也不一样。每种架构的计算机都有它自己的一套指令系统，规定了如何解释二进制指令。那么在指令系统A里面编译的程序就不可能在指令系统B里面运行，因为两者压根对程序的解读方法都不一样，这就是所谓的“不兼容”。但是现在我们要求一个程序能够跨平台运行，也就是说我写好的程序放在任何架构的计算机上都能正常执行，怎么办？一个好的办法是，我们干脆自己设计一套二进制指令的解释规则，即自己设计一套指令系统，而我们编写的程序永远只在我们设计的这套指令系统上运行，这样就不会出现不兼容的问题了。当然这个指令系统是仿真的，是在我们实际的电脑上实现的，也就是说它是虚拟的，所以我们把它叫作虚拟机。如果我们能在每个架构的计算机上都实现这么一个虚拟机，那么我们的程序不就实现兼容了。

既然我们自己设计了一套指令系统，那么我们的二进制程序也就会和真机上运行的不一样，我们要按照自己的规则编写指令，那么这种指令就是`字节码`。Java的虚拟机叫JVM，而class字节码文件

就是按JVM这套解释规则编写的指令。Python虽然不叫虚拟机，而叫解释器，但在这一方面，它跟JVM非常类似。

接下来我们讲讲`Python Opcode`。字节码中每个数据都是有一定含义的，但是计算机能够记住，人类却不容易记。因此我们需要一组助记符来方便我们解读字节码，这就是opcode。这种说法是不是有点似曾相识？是的，以前老师在讲到机器语言和汇编语言的时候说过，机器语言是机器可以直接识别的语言，但是机器码并不好记，于是引入了一组助记符，发展为汇编语言。因此Python Opcode可以类似理解成“Python的汇编”。opcode和字节码的关系可以类比成汇编语言和机器语言的关系，不过前者是在虚拟机下，后者是在真机下。

我们可以很容易打印出Python Opcode和字节码的对应关系：

```
import opcode
for op in range(len(opcode.opname)):
    print "0x%.2X(%.3d): %s" % (op, op, opcode.opname[op])
```

打印结果如下：

```
> python printopcode.py
0x00(000): STOP_CODE
0x01(001): POP_TOP
0x02(002): ROT_TWO
0x03(003): ROT_THREE
0x04(004): DUP_TOP
0x05(005): ROT_FOUR
0x06(006): <6>
0x07(007): <7>
0x08(008): <8>
0x09(009): NOP
0x0A(010): UNARY_POSITIVE
0x0B(011): UNARY_NEGATIVE
0x0C(012): UNARY_NOT
0x0D(013): UNARY_CONVERT
0x0E(014): <14>
0x0F(015): UNARY_INVERT
0x10(016): <16>
0x11(017): <17>
0x12(018): <18>
0x13(019): BINARY_POWER
0x14(020): BINARY_MULTIPLY
0x15(021): BINARY_DIVIDE
0x16(022): BINARY_MODULO
0x17(023): BINARY_ADD
0x18(024): BINARY_SUBTRACT
0x19(025): BINARY_SUBSCR
0x1A(026): BINARY_FLOOR_DIVIDE
0x1B(027): BINARY_TRUE_DIVIDE
0x1C(028): INPLACE_FLOOR_DIVIDE
0x1D(029): INPLACE_TRUE_DIVIDE
```

```
0x1E(030): SLICE+0
0x1F(031): SLICE+1
0x20(032): SLICE+2
0x21(033): SLICE+3
0x22(034): <34>
0x23(035): <35>
0x24(036): <36>
0x25(037): <37>
0x26(038): <38>
0x27(039): <39>
0x28(040): STORE_SLICE+0
0x29(041): STORE_SLICE+1
0x2A(042): STORE_SLICE+2
0x2B(043): STORE_SLICE+3
0x2C(044): <44>
0x2D(045): <45>
0x2E(046): <46>
0x2F(047): <47>
0x30(048): <48>
0x31(049): <49>
0x32(050): DELETE_SLICE+0
0x33(051): DELETE_SLICE+1
0x34(052): DELETE_SLICE+2
0x35(053): DELETE_SLICE+3
0x36(054): STORE_MAP
0x37(055): INPLACE_ADD
0x38(056): INPLACE_SUBTRACT
0x39(057): INPLACE_MULTIPLY
0x3A(058): INPLACE_DIVIDE
0x3B(059): INPLACE_MODULO
0x3C(060): STORE_SUBSCR
0x3D(061): DELETE_SUBSCR
0x3E(062): BINARY_LSHIFT
0x3F(063): BINARY_RSHIFT
0x40(064): BINARY_AND
0x41(065): BINARY_XOR
0x42(066): BINARY_OR
0x43(067): INPLACE_POWER
0x44(068): GET_ITER
0x45(069): <69>
0x46(070): PRINT_EXPR
0x47(071): PRINT_ITEM
0x48(072): PRINT_NEWLINE
0x49(073): PRINT_ITEM_TO
0x4A(074): PRINT_NEWLINE_TO
0x4B(075): INPLACE_LSHIFT
0x4C(076): INPLACE_RSHIFT
0x4D(077): INPLACE_AND
0x4E(078): INPLACE_XOR
0x4F(079): INPLACE_OR
0x50(080): BREAK_LOOP
0x51(081): WITH_CLEANUP
```

```
0x52(082): LOAD_LOCALS
0x53(083): RETURN_VALUE
0x54(084): IMPORT_STAR
0x55(085): EXEC_STMT
0x56(086): YIELD_VALUE
0x57(087): POP_BLOCK
0x58(088): END_FINALLY
0x59(089): BUILD_CLASS
0x5A(090): STORE_NAME
0x5B(091): DELETE_NAME
0x5C(092): UNPACK_SEQUENCE
0x5D(093): FOR_ITER
0x5E(094): LIST_APPEND
0x5F(095): STORE_ATTR
0x60(096): DELETE_ATTR
0x61(097): STORE_GLOBAL
0x62(098): DELETE_GLOBAL
0x63(099): DUP_TOPX
0x64(100): LOAD_CONST
0x65(101): LOAD_NAME
0x66(102): BUILD_TUPLE
0x67(103): BUILD_LIST
0x68(104): BUILD_SET
0x69(105): BUILD_MAP
0x6A(106): LOAD_ATTR
0x6B(107): COMPARE_OP
0x6C(108): IMPORT_NAME
0x6D(109): IMPORT_FROM
0x6E(110): JUMP_FORWARD
0x6F(111): JUMP_IF_FALSE_OR_POP
0x70(112): JUMP_IF_TRUE_OR_POP
0x71(113): JUMP_ABSOLUTE
0x72(114): POP_JUMP_IF_FALSE
0x73(115): POP_JUMP_IF_TRUE
0x74(116): LOAD_GLOBAL
0x75(117): <117>
0x76(118): <118>
0x77(119): CONTINUE_LOOP
0x78(120): SETUP_LOOP
0x79(121): SETUP_EXCEPT
0x7A(122): SETUP_FINALLY
0x7B(123): <123>
0x7C(124): LOAD_FAST
0x7D(125): STORE_FAST
0x7E(126): DELETE_FAST
0x7F(127): <127>
0x80(128): <128>
0x81(129): <129>
0x82(130): RAISE_VARARGS
0x83(131): CALL_FUNCTION
0x84(132): MAKE_FUNCTION
0x85(133): BUILD_SLICE
```

```
0x86(134): MAKE_CLOSURE
0x87(135): LOAD_CLOSURE
0x88(136): LOAD_DEREF
0x89(137): STORE_DEREF
0x8A(138): <138>
0x8B(139): <139>
0x8C(140): CALL_FUNCTION_VAR
0x8D(141): CALL_FUNCTION_KW
0x8E(142): CALL_FUNCTION_VAR_KW
0x8F(143): SETUP_WITH
0x90(144): <144>
0x91(145): EXTENDED_ARG
0x92(146): SET_ADD
0x93(147): MAP_ADD
0x94(148): <148>
0x95(149): <149>
0x96(150): <150>
0x97(151): <151>
0x98(152): <152>
0x99(153): <153>
0x9A(154): <154>
0x9B(155): <155>
0x9C(156): <156>
0x9D(157): <157>
0x9E(158): <158>
0x9F(159): <159>
0xA0(160): <160>
0xA1(161): <161>
0xA2(162): <162>
0xA3(163): <163>
0xA4(164): <164>
0xA5(165): <165>
0xA6(166): <166>
0xA7(167): <167>
0xA8(168): <168>
0xA9(169): <169>
0xAA(170): <170>
0xAB(171): <171>
0xAC(172): <172>
0xAD(173): <173>
0xAE(174): <174>
0xAF(175): <175>
0xB0(176): <176>
0xB1(177): <177>
0xB2(178): <178>
0xB3(179): <179>
0xB4(180): <180>
0xB5(181): <181>
0xB6(182): <182>
0xB7(183): <183>
0xB8(184): <184>
0xB9(185): <185>
```

```
0xBA(186): <186>
0xBB(187): <187>
0xBC(188): <188>
0xBD(189): <189>
0xBE(190): <190>
0xBF(191): <191>
0xC0(192): <192>
0xC1(193): <193>
0xC2(194): <194>
0xC3(195): <195>
0xC4(196): <196>
0xC5(197): <197>
0xC6(198): <198>
0xC7(199): <199>
0xC8(200): <200>
0xC9(201): <201>
0xCA(202): <202>
0xCB(203): <203>
0xCC(204): <204>
0xCD(205): <205>
0xCE(206): <206>
0xCF(207): <207>
0xD0(208): <208>
0xD1(209): <209>
0xD2(210): <210>
0xD3(211): <211>
0xD4(212): <212>
0xD5(213): <213>
0xD6(214): <214>
0xD7(215): <215>
0xD8(216): <216>
0xD9(217): <217>
0xDA(218): <218>
0xDB(219): <219>
0xDC(220): <220>
0xDD(221): <221>
0xDE(222): <222>
0xDF(223): <223>
0xE0(224): <224>
0xE1(225): <225>
0xE2(226): <226>
0xE3(227): <227>
0xE4(228): <228>
0xE5(229): <229>
0xE6(230): <230>
0xE7(231): <231>
0xE8(232): <232>
0xE9(233): <233>
0xEA(234): <234>
0xEB(235): <235>
0xEC(236): <236>
0xED(237): <237>
```

```
0xEE(238): <238>
0xEF(239): <239>
0xF0(240): <240>
0xF1(241): <241>
0xF2(242): <242>
0xF3(243): <243>
0xF4(244): <244>
0xF5(245): <245>
0xF6(246): <246>
0xF7(247): <247>
0xF8(248): <248>
0xF9(249): <249>
0xFA(250): <250>
0xFB(251): <251>
0xFC(252): <252>
0xFD(253): <253>
0xFE(254): <254>
0xFF(255): <255>
```

像0x17的opcode就是`BINARY_ADD`，表示加法运算。0x09表示`NOP`，这个NOP跟汇编指令nop效果一样都表示no operation，即什么都不做，只是汇编nop的机器码是0x90。我们还可以仔细观察到，不是每个字节码都对应有opcode，有很多目前还保留空在这里。

现在简单说一下pyc Bytecode(字节码)的格式，pyc指令分两种类型，一种是不带参数的单字节指令，一种是带参数的三字节指令，若字节码大于等于`0x5a`则属于三字节指令，其他的属于单字节指令。对于三字节指令来说，实际上只会用两个字节，第三个字节恒为0x00。

对于pyc文件格式，原极光协会会长JDChen曾在大学期间对此有过一些研究，点此处可以阅读他对此问题写的博客。这道题也是他基于此灵感出的。pyc的数据结构如下：

```
#define OFF(x) offsetof(PyCodeObject, x)
static PyMemberDef code_memberlist[] = {
    {"co_argcount",     T_INT,        OFF(co_argcount),       READONLY},
    {"co_nlocals",      T_INT,        OFF(co_nlocals),        READONLY},
    {"co_stacksize",T_INT,            OFF(co_stacksize),      READONLY},
    {"co_flags",        T_INT,        OFF(co_flags),          READONLY},
    {"co_code",         T_OBJECT,     OFF(co_code),           READONLY},
    {"co_consts",       T_OBJECT,     OFF(co_consts),         READONLY},
    {"co_names",        T_OBJECT,     OFF(co_names),          READONLY},
    {"co_varnames",     T_OBJECT,     OFF(co_varnames),       READONLY},
    {"co_freevars",     T_OBJECT,     OFF(co_freevars),       READONLY},
    {"co_cellvars",     T_OBJECT,     OFF(co_cellvars),       READONLY},
    {"co_filename",     T_OBJECT,     OFF(co_filename),       READONLY},
    {"co_name",         T_OBJECT,     OFF(co_name),           READONLY},
    {"co_firstlineno", T_INT,         OFF(co_firstlineno),    READONLY},
    {"co_lnotab",       T_OBJECT,     OFF(co_lnotab),         READONLY},
    {NULL}      /* Sentinel */
};
```

`co_consts`是常量名列表。不过我们最关心的还是`co_code`这一项，因为这里面存储的就是代码指令。我们可以通过写一段Python来打印某个pyc文件的数据结构，我们命名为`script.py`：

```
import marshal
import dis

f = open('hash.pyc', 'rb')
print f.read(4)
print f.read(4)

code = marshal.load(f)
print code.co_code.encode("hex")
print code.co_consts
print code.co_names
dis.disassemble(code)
```

如果要分析其他的pyc文件只需要把`open`函数里面的文件名改了就行，执行这段代码得到：

```
> python script.py
≤

╚z
6400006401006c00005a00006402005a01006403005a02006404005a0300787e006504006
405006406008302004 45d6d005a05006505006407006b0500725d00650100650300650500
181465020065030065050017141 75a06006e1a00650100650300650500171465020065030
06505001814175a06006500006a07008300005a08006508006a09006506008301000165080
06a0a008300004748712e005764010053
(-1, None, 'deadbeaf', '3&!2309', 4, 0, 6, 3)
('hashlib', 'a', 'b', 'c', 'range', 'i', 'st', 'md5', 'm', 'update', 'hex
digest')
  1           0 LOAD_CONST               0 (-1)
              3 LOAD_CONST               1 (None)
              6 IMPORT_NAME              0 (hashlib)
              9 STORE_NAME               0 (hashlib)

  4          12 LOAD_CONST               2 ('deadbeaf')
             15 STORE_NAME               1 (a)

  5          18 LOAD_CONST               3 ('3&!2309')
             21 STORE_NAME               2 (b)

  6          24 LOAD_CONST               4 (4)
             27 STORE_NAME               3 (c)

  7          30 SETUP_LOOP             126 (to 159)
             33 LOAD_NAME                4 (range)
             36 LOAD_CONST               5 (0)
             39 LOAD_CONST               6 (6)
             42 CALL_FUNCTION            2
             45 GET_ITER
```

```
          >>   46 FOR_ITER                109 (to 158)
               49 STORE_NAME                5 (i)

  8            52 LOAD_NAME                 5 (i)
               55 LOAD_CONST                7 (3)
               58 COMPARE_OP                5 (>=)
               61 POP_JUMP_IF_FALSE        93

  9            64 LOAD_NAME                 1 (a)
               67 LOAD_NAME                 3 (c)
               70 LOAD_NAME                 5 (i)
               73 BINARY_SUBTRACT
               74 BINARY_MULTIPLY
               75 LOAD_NAME                 2 (b)
               78 LOAD_NAME                 3 (c)
               81 LOAD_NAME                 5 (i)
               84 BINARY_ADD
               85 BINARY_MULTIPLY
               86 BINARY_ADD
               87 STORE_NAME                6 (st)
               90 JUMP_FORWARD             26 (to 119)

 11      >>   93 LOAD_NAME                 1 (a)
               96 LOAD_NAME                 3 (c)
               99 LOAD_NAME                 5 (i)
              102 BINARY_ADD
              103 BINARY_MULTIPLY
              104 LOAD_NAME                 2 (b)
              107 LOAD_NAME                 3 (c)
              110 LOAD_NAME                 5 (i)
              113 BINARY_SUBTRACT
              114 BINARY_MULTIPLY
              115 BINARY_ADD
              116 STORE_NAME                6 (st)

 12      >>  119 LOAD_NAME                 0 (hashlib)
              122 LOAD_ATTR                 7 (md5)
              125 CALL_FUNCTION             0
              128 STORE_NAME                8 (m)

 13           131 LOAD_NAME                 8 (m)
              134 LOAD_ATTR                 9 (update)
              137 LOAD_NAME                 6 (st)
              140 CALL_FUNCTION             1
              143 POP_TOP

 14           144 LOAD_NAME                 8 (m)
              147 LOAD_ATTR                10 (hexdigest)
              150 CALL_FUNCTION             0
              153 PRINT_ITEM
              154 PRINT_NEWLINE
              155 JUMP_ABSOLUTE            46
```

```
          >>  158 POP_BLOCK
          >>  159 LOAD_CONST               1 (None)
              162 RETURN_VALUE
```

那一长串数字就是`co_code`，而后面的opcode就是对这段pyc的解析。

解题过程

这道题给的题面非常丰富而且重要：

```
小祥为了保护自己的代码，修改了部分Python Bytecode指令集，并把这个指令集称之为JPython

JPython只能在他私人定制的环境上才能运行，其他人无法得到这个环境

现在，小明为了获取小祥代码中的秘密，收集到了三个文件：

* hash.pyc  可以直接使用Python 2.7运行的脚本
* Jhash.pyc 通过hash.pyc转化而成的只能在JPython环境上运行的脚本
* Jflag.pyc 藏着小祥最大的秘密，但是只能在JPython的环境运行

谁能帮助小明得到小祥代码里的秘密呢？
```

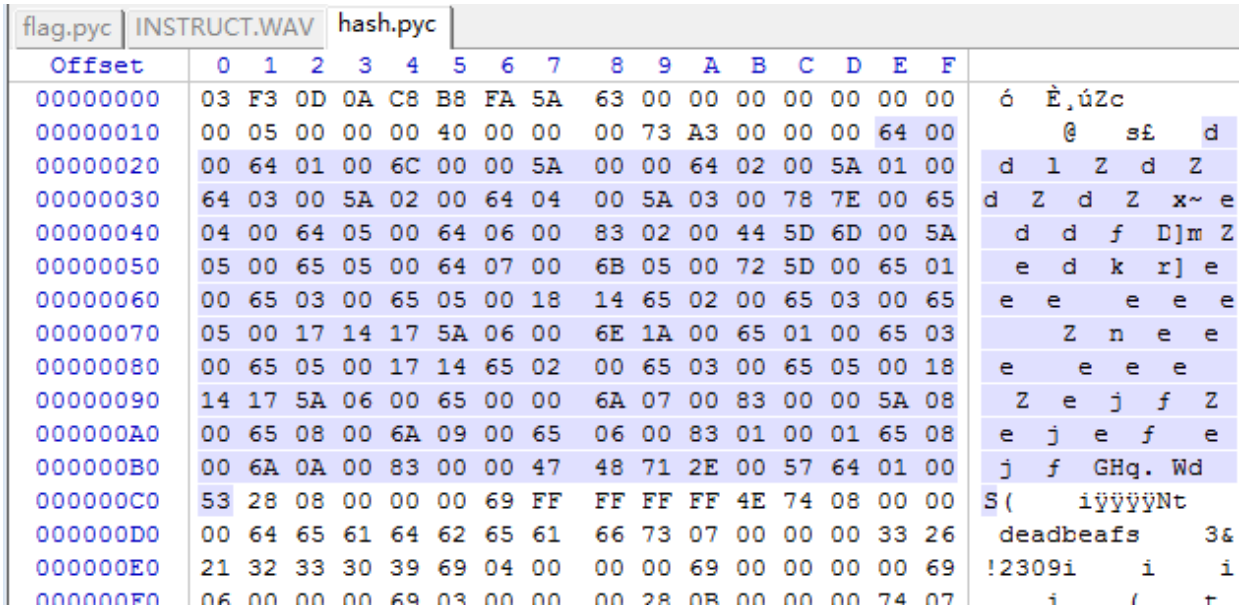
从这段描述，结合前面所说的预备知识，我们可以得到如下几点：

- 蕴藏着flag的那个pyc文件无法用现有的Python解释器运行，因为指令系统不同
- JPython指令集（指令系统）是由Python指令集进行部分修改得来，整体框架应该没有大变化
- 提供`hash.pyc`和`Jhash.pyc`的目的在于让我们得到Python指令集和JPython指令集的映射关系

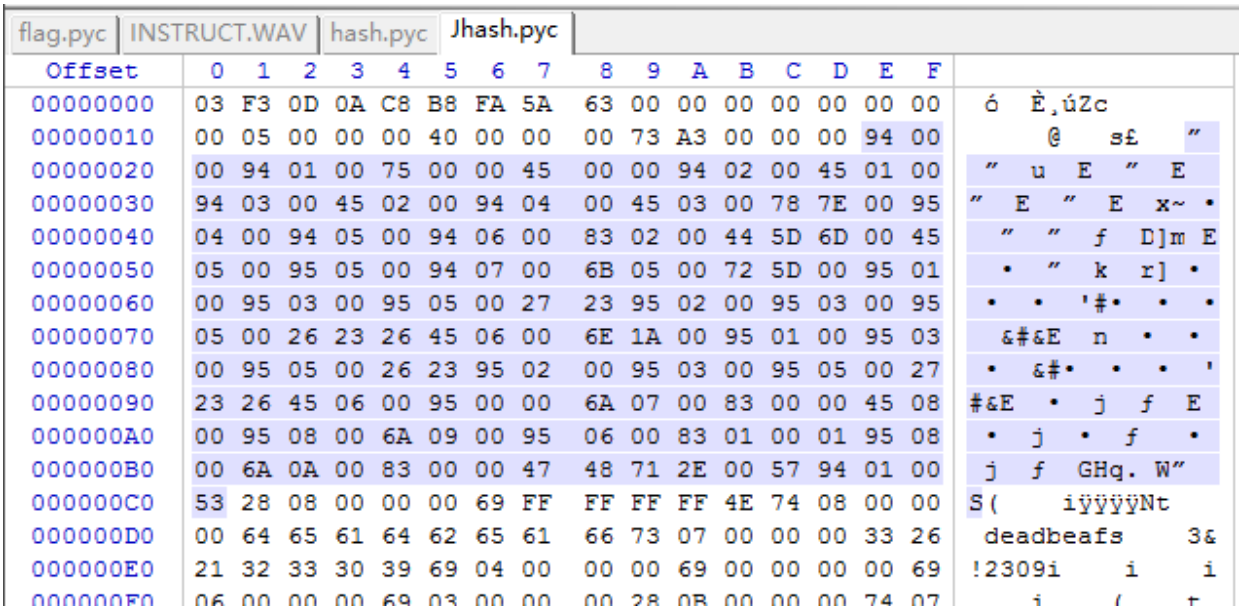
首先要明白的是，对于我们解题人来说，最佳思路并不是按照题目一样也去设计一个JPython解释器，虽然如果你对Python解释器比较熟悉的话可以这样做，而是应该想办法得出Python指令集和JPython指令集的映射关系，然后将`Jflag.pyc`修改成`flag.pyc`，最后使用在线pyc转py还原出逻辑。

用`script.py`把`hash.pyc`和`Jhash.pyc`的`co_code`部分都打印出来，然后都用`Winhex`十六进制编辑器定位到`co_code`部分。

首先是`hash.pyc`：



然后是`Jhash.pyc`：



根据预备知识部分所讲，在`hash.pyc`中，若opcode大于等于`0x5a`的话就按三字节指令读，否则按单字节指令读。对比两个文件我们可以得到如下映射关系：

hash.pyc	Jhash.pyc	操作
0x64	0x94	LOAD_CONST
0x6c	0x75	IMPORT_NAME
0x5a	0x45	STORE_NAME
0x65	0x95	LOAD_NAME
0x18	0x27	BINARY_SUBTRACT
0x14	0x23	BINARY_MULTIPLY
0x17	0x26	BINARY_ADD

其实细心的同学会发现，这张表中`Jhash.pyc`里面的那几个字节码正好在Python字节码中没定义，这样做其实也是为了避免歧义，大家就可以确定这些字节码的确是被修改过的，不然解题者在由`Jflag.pyc`推回`flag.pyc`的时候怎么确定它到底是被修改过的，还是本身就是由Python字节码原封不动继承过来的，因为题面也说只修改了部分字节码。

接下来就是根据这个映射关系将`Jflag.pyc`推回到`flag.pyc`，不过你会发现`Jflag.pyc`有一个字节码是0x24，0x24在Python字节码中没有定义，但是也不在那张映射关系表里面，不过当我们更细心地观察这几个映射关系的时候

```
hash.pyc        Jhash.pyc       操作
0x17            0x26            BINARY_ADD
0x18            0x27            BINARY_SUBTRACT
0x14            0x23            BINARY_MULTIPLY
```

发现他们正好是加减乘，唯独少了一个除，而除法操作(`BINARY_DIVIDE`)的字节码正好是0x15，所以我们应该可以合理地推断这样的映射关系：

```
hash.pyc        Jhash.pyc       操作
0x15            0x24            BINARY_DIVIDE
```

这是出题者故意这样做的，目的在于出题者不想让大家只会简单地替换，而是想让大家更深入地了解一下opcode，所以设计了一个合理推断过程。现在就可以还原出`flag.pyc`了，但是新的问题又来了，执行`flag.pyc`没问题，用`script.py`推出opcode也没问题：

```
  1           0 JUMP_ABSOLUTE           12
              3 LOAD_CONST               1 (None)
              6 IMPORT_NAME              0 (time)
              9 STORE_NAME               0 (time)

  2     >>   12 LOAD_CONST               0 (-1)
             15 LOAD_CONST               1 (None)
             18 IMPORT_NAME              1 (base64)
             21 STORE_NAME               1 (base64)

  3          24 LOAD_CONST               0 (-1)
             27 LOAD_CONST               1 (None)
             30 IMPORT_NAME              2 (sys)
             33 STORE_NAME               2 (sys)

  6          36 LOAD_NAME                2 (sys)
             39 LOAD_ATTR                3 (argv)
             42 LOAD_CONST               2 (1)
             45 BINARY_SUBSCR
             46 STORE_NAME               4 (flag)

  7          49 LOAD_CONST               3 ('jd')
             52 STORE_NAME               5 (jd)

  8          55 LOAD_NAME                6 (len)
             58 LOAD_NAME                4 (flag)
             61 CALL_FUNCTION            1
             64 LOAD_CONST               4 (30)
             67 COMPARE_OP               2 (==)
```

```
              70 POP_JUMP_IF_FALSE      210

 9            73 LOAD_NAME                1 (base64)
              76 LOAD_ATTR                7 (b64encode)
              79 LOAD_NAME                4 (flag)
              82 LOAD_CONST              16 ('+1s+1s+1s')
              85 BINARY_ADD
              86 LOAD_NAME                5 (jd)
              89 LOAD_CONST               7 (2)
              92 BINARY_MULTIPLY
              93 BINARY_ADD
              94 CALL_FUNCTION            1
              97 STORE_NAME               8 (base64_str)

10           100 LOAD_CONST               8 ('')
             103 STORE_NAME               9 (b)

11           106 SETUP_LOOP              73 (to 182)
             109 LOAD_NAME               10 (range)
             112 LOAD_CONST               9 (0)
             115 LOAD_CONST              10 (44)
             118 CALL_FUNCTION            2
             121 GET_ITER
         >>  122 FOR_ITER                 56 (to 181)
             125 STORE_NAME              11 (i)

12           128 LOAD_NAME               12 (ord)
             131 LOAD_NAME                8 (base64_str)
             134 LOAD_NAME               11 (i)
             137 BINARY_SUBSCR
             138 CALL_FUNCTION            1
             141 LOAD_CONST              11 (10)
             144 BINARY_DIVIDE
             145 STORE_NAME              13 (head)

13           148 LOAD_NAME                9 (b)
             151 LOAD_NAME               14 (chr)
             154 LOAD_NAME               12 (ord)
             157 LOAD_NAME                8 (base64_str)
             160 LOAD_NAME               11 (i)
             163 BINARY_SUBSCR
             164 CALL_FUNCTION            1
             167 LOAD_CONST              12 (7)
             170 BINARY_XOR
             171 CALL_FUNCTION            1
             174 INPLACE_ADD
             175 STORE_NAME               9 (b)
             178 JUMP_ABSOLUTE          122
         >>  181 POP_BLOCK

14       >>  182 LOAD_NAME                9 (b)
```

```
             185 LOAD_CONST               13 ('^P]mc@]0emE7VOE2_}A}VBwpbQ?5
e5>lN4UwSSM>L}A}')
             188 COMPARE_OP                2 (==)
             191 POP_JUMP_IF_FALSE       202

 15          194 LOAD_CONST               14 ('Congratulations!You Get Fla
g')
             197 PRINT_ITEM
             198 PRINT_NEWLINE
             199 JUMP_ABSOLUTE           210

 17     >>  202 LOAD_CONST               15 ('Wrong!')
             205 PRINT_ITEM
             206 PRINT_NEWLINE
             207 JUMP_FORWARD             0 (to 210)
         >>  210 LOAD_CONST               1 (None)
             213 RETURN_VALUE
```

但是就是不能回推成py文件，这是因为最开头的4条指令12个字节是个花指令，干扰了分析，但是却不影响运行，问题主要体现在第一条指令的绝对跳转上

```
  1           0 JUMP_ABSOLUTE           12
              3 LOAD_CONST               1 (None)
              6 IMPORT_NAME              0 (time)
              9 STORE_NAME               0 (time)
```

详情请点击此处。解决方法也很简单，直接把最开头12字节`NOP`掉就行了，也就是说用0x09覆盖`co_code`的最开头12个字节。现在就可以用在线pyc转py得到真实逻辑了：

```
import base64
import sys
flag = sys.argv[1]
jd = 'jd'
if len(flag) == 30:
    base64_str = base64.b64encode(flag + '+1s+1s+1s' + jd * 2)
    b = ''
    for i in range(0, 44):
        head = ord(base64_str[i]) / 10
        b += chr(ord(base64_str[i]) ^ 7)

    if b == '^P]mc@]0emE7VOE2_}A}VBwpbQ?5e5>lN4UwSSM>L}A}':
        print 'Congratulations!You Get Flag'
    else:
        print 'Wrong!'
```

算法很简单我就不多解释了，直接写解题代码了：

```
import base64
```

```
enc = '^P]mc@]0emE7VOE2_}A}VBwpbQ?5e5>lN4UwSSM>L}A}'
enc_ = ""
for c in enc:
    enc_ += chr(ord(c) ^ 7)
flag = base64.b64decode(enc_)
print flag
```

flag

`afctf{n0t@py_1s@Jpy_6ood#tiM2}`