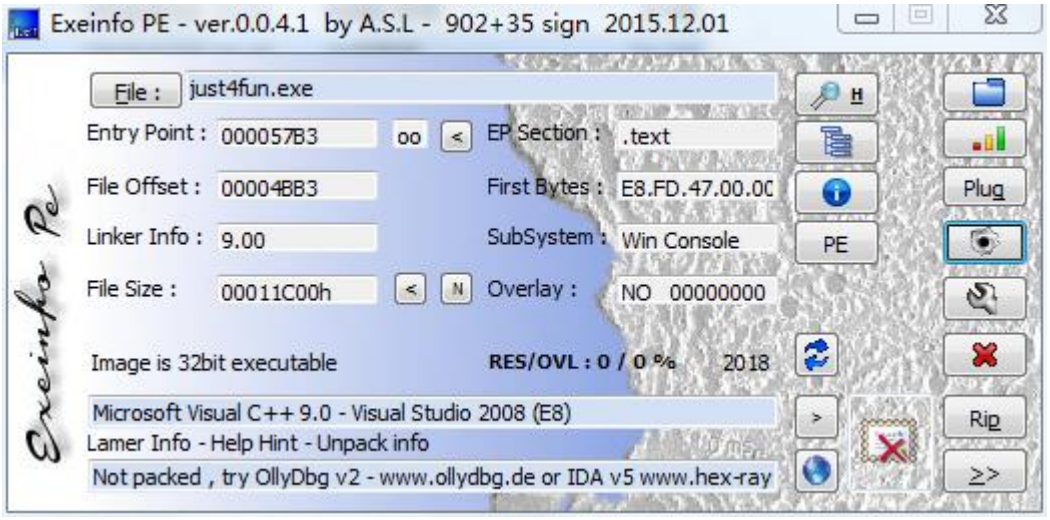
首先查壳，发现没有壳



然后拖进 IDA，找到主程序，F5

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char *v3; // ST0C_4

  v3 = (char *)malloc(0xFu);
  memset(byte_414C80, 0, 0x16u);
  sub_401090(v3);
  sub_401150(v3);
  sub_4011B0(v3, v3 + 4);
  sub_4011F0(v3, v3 + 4);
  if ( dword_413038 )
    printf(aErrorPleaseTry);
  else
    printf(aNiceYouAreRigh);
  return 0;
}
```

首先看一下第一个函数 sub_401090(v3)

```
char *__cdecl sub_401090(char *a1)
{
  char *v2; // [esp-2h] [ebp-34h]
  char v3; // [esp+2h] [ebp-30h]
  char v4; // [esp+3h] [ebp-2Fh]
  char v5; // [esp+4h] [ebp-2Eh]
  char v6; // [esp+5h] [ebp-2Dh]
  char v7; // [esp+6h] [ebp-2Ch]
  char v8; // [esp+7h] [ebp-2Bh]
  char v9; // [esp+8h] [ebp-2Ah]
  char v10; // [esp+22h] [ebp-10h]

  printf(aWelcomeToAfctf);
  v2 = &v3;
  scanf(aS, &v3);
  if ( v3 != 'A' || v4 != 'F' || v5 != 'C' || v6 != 'T' || v7 != 'F' || v8 != '{' || *(&v2 + strlen(&v3) + 3) != '}' )
    exit(0);
  *(&v2 + strlen(&v3) + 3) = 0;
  strcpy(&v10, &v9);
  strcpy(a1, &v10);
  return strcpy(byte_414C80, &v10);
}
```

获取用户输入，然后检查是否符合 AFCTF{}的格式，然后将 V9 的内容拷贝到 V10 处，双击 V9 查看堆栈情况，如下

```
-00000031 var_31          db ?
-00000030 var_30          db ?
-0000002F var_2F          db ?
-0000002E var_2E          db ?
-0000002D var_2D          db ?
-0000002C var_2C          db ?
-0000002B var_2B          db ?
-0000002A var_2A          db ?
-00000029                 db ? ; undefined
-00000028                 db ? ; undefined
-00000027                 db ? ; undefined
-00000026                 db ? ; undefined
-00000025                 db ? ; undefined
-00000024                 db ? ; undefined
-00000023                 db ? ; undefined
-00000022                 db ? ; undefined
-00000021                 db ? ; undefined
-00000020                 db ? ; undefined
-0000001F                 db ? ; undefined
-0000001E                 db ? ; undefined
-0000001D                 db ? ; undefined
-0000001C                 db ? ; undefined
-0000001B                 db ? ; undefined
-0000001A                 db ? ; undefined
-00000019                 db ? ; undefined
-00000018                 db ? ; undefined
-00000017                 db ? ; undefined
-00000016                 db ? ; undefined
-00000015                 db ? ; undefined
-00000014                 db ? ; undefined
-00000013                 db ? ; undefined
-00000012                 db ? ; undefined
-00000011                 db ? ; undefined
-00000010 var_10          db ?
-0000000F                 db ? ; undefined
-0000000E                 db ? ; undefined
-0000000D                 db ? ; undefined
-0000000C                 db ? ; undefined
-0000000B                 db ? ; undefined
-0000000A                 db ? ; undefined
-00000009                 db ? ; undefined
-00000008                 db ? ; undefined
-00000007                 db ? ; undefined
-00000006                 db ? ; undefined
-00000005                 db ? ; undefined
-00000004                 db ? ; undefined
-00000003                 db ? ; undefined
-00000002                 db ? ; undefined
-00000001                 db ? ; undefined
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
+00000008 arg_0           dd ?                    ; offset
```

V9 就是从 0x2A 处开始，而 0x30 是 V3 的地址，可以发现 0x2F 保存的应该是”A”，0x30 保存的是”F”，0x2E 保存的是”C”，0x2D 保存的是”T”，0x2C 保存的是”F”，0x2B 保存的是”{”，所以将 V9 拷贝到 V10 实际上就是将 AFCTF{}中的字符串拷贝到 V10 处。回到代码处，接下来再将这个字符串拷贝到 a1 和 0x414C80 处。

下面回到主函数，看下一个函数

```
1  char *__cdecl sub_401150(char *a1)
2  {
3    char *result; // eax
4    char *v2; // [esp+0h] [ebp-8h]
5    signed int i; // [esp+4h] [ebp-4h]
6
7    result = strlen(a1);
8    v2 = result;
9    for ( i = 0; i < v2; ++i )
10   {
11     result = &a1[i];
12     a1[i] = 4 * v2 ^ (a1[i] >> 2);
13   }
14   return result;
15 }
```

这里将 a1 的每个字符串左移 2 位与右移 2 位进行异或。

第三个函数

```
1 int __cdecl sub_4011B0(int a1, int a2)
2 {
3   int result; // eax
4
5   result = a2 + 5 * (a2 - a1);
6   if ( result == -1890567614 && a1 + 13 * (a2 - a1) == -279954878 )
7     --dword_413038;
8   return result;
9 }
```

这里是解一个二元一次方程组，如果前面处理的串满足这个方程组的话，那么 0x413038 处的值减一，点进去这个地址可以看到这个地址处的值是 2

第四个函数

```
1 int __cdecl sub_4011F0(int a1, int a2)
2 {
3   int result; // eax
4
5   result = a2 + 17 * (a2 - a1);
6   if ( result == -207009661 && a1 + 7 * (a2 - a1) == 866732163 )
7     --dword_413038;
8   return result;
9 }
```

和第三个函数几乎一样，但是是另一个方程组，若满足，则 0x413038 处的值减一

回到主函数，看到

```
if ( dword_413038 )
  printf(aErrorPleaseTry);
else
  printf(aNiceYouAreRigh);
return 0;
```

也就是说，输入的串经过变换后，若满足两个方程组的话，那么就输出 nice you are right。但是，如果这个时候尝试去解方程组的话，就会发现两个变量，四条式子，根本无解，所以不管输入什么都不正确。那么说明这个地方的算法不是正确算法，那么回到汇编处，往下翻看代码，当翻到这个位置时，看到

```
.text:00401230
.text:00401230 loc_401230:                             ; CODE XREF: .text:00404B9B↓p
.text:00401230                 push    ebp
.text:00401231                 mov     ebp, esp
.text:00401233                 push    ecx
.text:00401233 ; ---------------------------------------------------------------------------
.text:00401234                 dd 0E15h dup(0)
.text:00404A88                 dd 90000000h, 2 dup(0)
.text:00404A94 ; ---------------------------------------------------------------------------
.text:00404A94                 mov     dword ptr [ebp-4], 0
.text:00404A9B                 jmp     short loc_404AA6
.text:00404A9D ; ---------------------------------------------------------------------------
.text:00404A9D
.text:00404A9D loc_404A9D:                             ; CODE XREF: .text:00404B23↓j
.text:00404A9D                 mov     eax, [ebp-4]
.text:00404AA0                 add     eax, 1
.text:00404AA3                 mov     [ebp-4], eax
.text:00404AA6
.text:00404AA6 loc_404AA6:                             ; CODE XREF: .text:00404A9B↑j
.text:00404AA6                 mov     eax, [ebp+0Ch]
.text:00404AA9                 cdq
.text:00404AAA                 sub     eax, edx
.text:00404AAC                 sar     eax, 1
.text:00404AAE                 cmp     [ebp-4], eax
```

从这里开始，左边的.text 全部都是红色，说明 IDA 在分析这个位置的时候不能正常分析。看到代码中间存在很多 0，这里就存在问题了，因为，如果是正常的程序，.text 段是不会存在如此多的 0，所以这个位置肯定被动过手脚，那么将这些 0 NOP 掉（可以使用 OD，然后批量修改机器码，然后保存，也可以用 IDC 脚本，或者直接修改汇编指令为 jmp 跳过这大段 0）。再继续往下看

```
.text:00404B4E                 add     esp, 4
.text:00404B51                 add     eax, 1
.text:00404B54                 mov     [ebp-18h], eax
.text:00404B57                 mov     eax, [ebp-18h]
.text:00404B5A                 shl     eax, 2
.text:00404B5D                 cdq
.text:00404B5E                 mov     ecx, 3
.text:00404B63                 idiv    ecx
.text:00404B65                 add     eax, 1
.text:00404B68                 push    eax
.text:00404B69                 call    _malloc
.text:00404B6E                 add     esp, 4
.text:00404B71                 mov     [ebp-14h], eax
.text:00404B74                 mov     byte ptr [ebp-0Ch], 63h
.text:00404B78                 mov     byte ptr [ebp-0Bh], 68h
.text:00404B7C                 mov     byte ptr [ebp-0Ah], 65h
.text:00404B80                 mov     byte ptr [ebp-9], 65h
.text:00404B84                 leave
.text:00404B85                 retn
.text:00404B86 ; ---------------------------------------------------------------------------
.text:00404B86                 mov     byte ptr [ebp-8], 72h
.text:00404B8A                 mov     byte ptr [ebp-7], 73h
.text:00404B8E                 mov     byte ptr [ebp-6], 0
.text:00404B92                 mov     edx, [ebp-18h]
.text:00404B95                 push    edx
.text:00404B96                 push    offset byte_414C80
```

0x404B84 与 0x404B85 的 leave 和 retn 这里有问题，因为可以看到上下两处的代码显然是相连的，而且下面的代码也不是一个正常函数开始的情况，即 push ebp；mov ebp, esp 之类的，所以这里也将它 NOP 掉。往下翻

```
.text:00404D83                  mov     byte ptr [ebp-1Fh], 6Dh
.text:00404D87                  mov     byte ptr [ebp-1Eh], 46h
.text:00404D8B                  mov     byte ptr [ebp-1Dh], 66h
.text:00404D8F                  mov     byte ptr [ebp-1Ch], 62h
.text:00404D93                  mov     byte ptr [ebp-1Bh], 6Ch
.text:00404D97                  mov     byte ptr [ebp-1Ah], 64h
.text:00404D9B                  mov     byte ptr [ebp-19h], 77h
.text:00404D9F                  mov     byte ptr [ebp-18h], 0
.text:00404DA3                  leave
.text:00404DA4                  retn
.text:00404DA5 ; ---------------------------------------------------------
.text:00404DA5                  lea     ecx, [ebp-38h]
.text:00404DA8                  push    ecx
.text:00404DA9                  mov     edx, [ebp-14h]
.text:00404DAC                  push    edx
.text:00404DAD                  call    _strcmp
.text:00404DB2                  add     esp, 8
.text:00404DB5                  test    eax, eax
.text:00404DB7                  jnz     short loc_404DCA
```

这里同样的道理，404DA3 和 404DA4 将它 NOP 掉，然后再往后看的时候，就没有这些不正常的代码了。将修改好的文件重新拖进 IDA，可以得到两个新的函数，这里就应该是正确算法了

第一个函数

```
int __cdecl sub_401230(int a1, int a2)
{
  int result; // eax
  int i; // [esp+0h] [ebp-4h]

  for ( i = 0; ; ++i )
  {
    result = a2 / 2;
    if ( i >= a2 / 2 )
      break;
    *(i + a1) ^= *(a1 + a2 - 1 - i);
    *(a1 + a2 - 1 - i) ^= *(i + a1);
    *(i + a1) ^= *(a1 + a2 - 1 - i);
  }
  return result;
}
```

这里很简单，就是通过异或的方式，将一个字符串前后颠倒

第二个函数：

```
int sub_404B30()
{
  int result; // eax
  char v1; // [esp+4h] [ebp-50h]
  char v2; // [esp+8h] [ebp-4Ch]
  char v3; // [esp+17h] [ebp-3Dh]
  char v4; // [esp+1Ch] [ebp-38h]
  char v5; // [esp+1Dh] [ebp-37h]
  char v6; // [esp+1Eh] [ebp-36h]
  char v7; // [esp+1Fh] [ebp-35h]
  char v8; // [esp+20h] [ebp-34h]
  char v9; // [esp+21h] [ebp-33h]
  char v10; // [esp+22h] [ebp-32h]
  char v11; // [esp+23h] [ebp-31h]
  char v12; // [esp+24h] [ebp-30h]
  char v13; // [esp+25h] [ebp-2Fh]
  char v14; // [esp+26h] [ebp-2Eh]
  char v15; // [esp+27h] [ebp-2Dh]
  char v16; // [esp+28h] [ebp-2Ch]
  char v17; // [esp+29h] [ebp-2Bh]
  char v18; // [esp+2Ah] [ebp-2Ah]
  char v19; // [esp+2Bh] [ebp-29h]
  char v20; // [esp+2Ch] [ebp-28h]
  char v21; // [esp+2Dh] [ebp-27h]
  char v22; // [esp+2Eh] [ebp-26h]
  char v23; // [esp+2Fh] [ebp-25h]
  char v24; // [esp+30h] [ebp-24h]
  char v25; // [esp+31h] [ebp-23h]
  char v26; // [esp+32h] [ebp-22h]
  char v27; // [esp+33h] [ebp-21h]
  char v28; // [esp+34h] [ebp-20h]
  char v29; // [esp+35h] [ebp-1Fh]
  char v30; // [esp+36h] [ebp-1Eh]
  char v31; // [esp+37h] [ebp-1Dh]
  char v32; // [esp+38h] [ebp-1Ch]
  char v33; // [esp+39h] [ebp-1Bh]
  char v34; // [esp+3Ah] [ebp-1Ah]
  char v35; // [esp+3Bh] [ebp-19h]
  int v36; // [esp+3Ch] [ebp-18h]
  char *v37; // [esp+40h] [ebp-14h]
  int v38; // [esp+44h] [ebp-10h]
  char v39; // [esp+48h] [ebp-Ch]
  char v40; // [esp+49h] [ebp-Bh]
  char v41; // [esp+4Ah] [ebp-Ah]
  char v42; // [esp+4Bh] [ebp-9h]
  char v43; // [esp+4Ch] [ebp-8h]
  char v44; // [esp+4Dh] [ebp-7h]
  char v45; // [esp+4Eh] [ebp-6h]
  int v46; // [esp+50h] [ebp-4h]

  v46 = 0;
  v38 = 0;
  v36 = strlen(byte_414C80) + 1;
  v37 = malloc(4 * v36 / 3 + 1);
  v39 = 'c';
  v40 = 'h';
  v41 = 'e';
  v42 = 'e';
  v43 = 'r';
  v44 = 's';
  v45 = 0;
  sub_401230(byte_414C80, v36);
  while ( v38 < v36 )
  {
    v3 = byte_414C80[v38++];
    if ( v38 >= v36 )
```

```
      v2 = 0;
    else
      v2 = byte_414C80[v38++];
    if ( v38 >= v36 )
      v1 = 0;
    else
      v1 = byte_414C80[v38++];
    v37[v46++] = off_413034[(v3 >> 2) & 0x3F];
    v37[v46++] = off_413034[(((v2 & 0xFF) >> 4) | 16 * v3) & 0x3F];
    v37[v46++] = off_413034[(((v1 & 0xFF) >> 6) | 4 * v2) & 0x3F];
    v37[v46++] = off_413034[v1 & 0x3F];
  }
  if ( v36 % 3 == 1 )
  {
    v37[--v46] = 61;
    goto LABEL_14;
  }
  if ( v36 % 3 == 2 )
LABEL_14:
    v37[--v46] = 61;
  v37[v46] = 0;
  v4 = 'A';
  v5 = 'E';
  v6 = 'B';
  v7 = 'L';
  v8 = 'M';
  v9 = 'C';
  v10 = 'F';
  v11 = '5';
  v12 = 'b';
  v13 = 'm';
  v14 = '5';
  v15 = '1';

  v16 = 'Z';
  v17 = 'l';
  v18 = '9';
  v19 = 'z';
  v20 = 'M';
  v21 = 'V';
  v22 = '9';
  v23 = 'l';
  v24 = 'U';
  v25 = 'l';
  v26 = '9';
  v27 = 'k';
  v28 = 'b';
  v29 = 'm';
  v30 = 'F';
  v31 = 'f';
  v32 = 'b';
  v33 = 'l';
  v34 = 'd';
  v35 = 'w';
  LOBYTE(v36) = 0;
  result = strcmp(v37, &v4);
  if ( !result )
    result = printf(aS_0, &v39);
  return result;
}
```

可以看到这里出现了“cheers”，那么可以确定它就是正确算法了。这里看到它首先调用了一个函数，参数是 0x414C80 处的值，点进去看到发现是未初始化的变量

```
data:00414C80 byte_414C80     db 380h dup(?)          ; DATA XREF: _main+15↑o
data:00414C80                                         ; sub_401090+AC↑o ...
data:00414C80 _data           ends
```

但是在主函数调用的一个函数中，它在最后返回的时候将用户输入的字符串拷贝到这里了

```
  strcpy(&v10, &v9);
  strcpy(a1, &v10);
  return strcpy(byte_414C80, &v10);
}
```

继续分析，看到这一大串操作

```
  while ( v38 < v36 )
  {
    v3 = byte_414C80[v38++];
    if ( v38 >= v36 )
      v2 = 0;
    else
      v2 = byte_414C80[v38++];
    if ( v38 >= v36 )
      v1 = 0;
    else
      v1 = byte_414C80[v38++];
    v37[v46++] = off_413034[(v3 >> 2) & 0x3F];
    v37[v46++] = off_413034[(((v2 & 0xFF) >> 4) | 16 * v3) & 0x3F];
    v37[v46++] = off_413034[(((v1 & 0xFF) >> 6) | 4 * v2) & 0x3F];
    v37[v46++] = off_413034[v1 & 0x3F];
  }
  if ( v36 % 3 == 1 )
  {
    v37[--v46] = 61;
    goto LABEL_14;
  }
  if ( v36 % 3 == 2 )
LABEL_14:
    v37[--v46] = '=';
```

如果直接看的话，可能很难分析出它的逆算法，但是如果同学们的基础够扎实的话，会发现，这个实际上就是 base64 的加密算法，而后面也可以看到比对字符串

```
  v4 = 'A';
  v5 = 'E';
  v6 = 'B';
  v7 = 'L';
  v8 = 'M';
  v9 = 'C';
  v10 = 'F';
  v11 = '5';
```

```
v12 = 'b';
v13 = 'm';
v14 = '5';
v15 = '1';
v16 = 'Z';
v17 = 'l';
v18 = '9';
v19 = 'z';
v20 = 'M';
v21 = 'V';
v22 = '9';
v23 = 'l';
v24 = 'U';
v25 = 'l';
v26 = '9';
v27 = 'k';
v28 = 'b';
v29 = 'm';
v30 = 'F';
v31 = 'f';
v32 = 'b';
v33 = 'l';
v34 = 'd';
v35 = 'w';
```

也就是说将这个字符串进行 base64 解码，然后再颠倒字符串就能得到 flag

Flag：AFCTF{pWn_and_Re_1s_funny!0K@}

这里讲一下为何输入这个串会从主程序跳到正确算法，这里的关键点就是主函数第一个调用的函数中，这里的拷贝函数没有做长度检查

```
  strcpy(&v10, &v9);
  strcpy(a1, &v10);
  return strcpy(byte_414C80, &v10);
}
```

所以 v9 拷贝到 v10 的时候发生栈溢出，溢出到了返回地址，此时的堆栈情况是这样的

```
    | pWn_and_Re_1s_fu | nny!|0K@|
    |<------------------------>|<--->|<--->|
          V10                   EBP    return address
```

根据小端序的原则，0（数字）K@所表示的地址是 404B30，而正确算法的地址正好就是 0x404B30

```
.text:00404B30 var_10          = dword ptr -10h
.text:00404B30 var_C           = byte ptr -0Ch
.text:00404B30 var_B           = byte ptr -0Bh
.text:00404B30 var_A           = byte ptr -0Ah
.text:00404B30 var_9           = byte ptr -9
.text:00404B30 var_8           = byte ptr -8
.text:00404B30 var_7           = byte ptr -7
.text:00404B30 var_6           = byte ptr -6
.text:00404B30 var_4           = dword ptr -4
.text:00404B30
.text:00404B30                 push    ebp
.text:00404B31                 mov     ebp, esp
.text:00404B33                 sub     esp, 54h
.text:00404B36 ; 50:   v46 = 0;
.text:00404B36                 mov     [ebp+var_4], 0
.text:00404B3D ; 51:   v38 = 0;
.text:00404B3D                 mov     [ebp+var_10], 0
.text:00404B44 ; 52:   v36 = strlen(byte_414C80) + 1;
.text:00404B44                 push    offset byte_414C80 ; char *
.text:00404B49                 call    _strlen
.text:00404B4E                 add     esp, 4
.text:00404B51                 add     eax, 1
.text:00404B54                 mov     [ebp+var_18], eax
.text:00404B57 ; 53:   v37 = malloc(4 * v36 / 3 + 1);
.text:00404B57                 mov     eax, [ebp+var_18]
.text:00404B5A                 shl     eax, 2
.text:00404B5D                 cdq
.text:00404B5E                 mov     ecx, 3
.text:00404B63                 idiv    ecx
.text:00404B65                 add     eax, 1
.text:00404B68                 push    eax             ; size_t
.text:00404B69                 call    _malloc
```

所以当函数返回的时候，EIP 被劫持到 0x404B30 处，运行这里的代码，从而运行正确的算法。

PS.在使用 c/c++编程的时候，若没有对输入的串进行长度检查，很容易就造成堆栈溢出，堆栈溢出带来的危害是巨大的，这道题目的正确算法是一个普通的 base64 算法，但要是一个病毒的代码呢，或者是一段拿到系统 shell 的代码呢，这样你的这个程序就会被黑客们当做跳板，从而对你们的计算机进行操控（嗯，rm -f* 滑稽脸）。而找到程序漏洞并利用，这就是 pwn 的精髓，而这道题只是简单的结合了 pwn 和 reverse 的一道题目，也主要是给大家简单了解一下 pwn 是什么样的一个东西，希望大家在以后的二进制学习中能有所收获，加油！