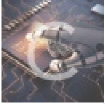
uglycode - AliCTF - 2016 - Reverse

原创

风靡义磊 于 2016-07-21 11:14:56 发布 906 收藏
分类专栏： 逆向 文章标签： alictf writeup reverse
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/w597013296/article/details/51980843
版权


逆向 专栏收录该内容
7 篇文章 0 订阅
订阅专栏
下载链接

先用IDA看看main部分

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  __int64 result; // rax@2
  char s1; // [sp+0h] [bp-110h]@1
  void *v5; // [sp+108h] [bp-8h]@3
  sub_400930(a1, a2, a3);
  puts("input you passcode:");
  gets(&s1);
  if ( !strcmp(&s1, "xx xx xx xx xx xx") )
  {
    v5 = &unk_6030C0;
    ((void (__fastcall *)(signed __int64, const char *))unk_6030C0)(2LL, "xx xx xx xx xx xx");
    result = 0LL;
  }
  else
  {
    result = 0LL;
  }
  return result;
}
```

首先程序让你输入passcode，就是"xx xx xx xx xx xx"，接下来调用unk_6030C0处的函数（注意那个参数2）。所以跳到此处看。

```
.data:00000000006030C0 unk_6030C0      db  55h ; U              ; DATA XREF: main:loc_401126o
.data:00000000006030C1                 db  48h ; H
.data:00000000006030C2                 db  89h ;
.data:00000000006030C3                 db 0E5h ;
.data:00000000006030C4                 db  48h ; H
.data:00000000006030C5                 db  81h ;
.data:00000000006030C6                 db 0ECh ;
```

出现这样的东西，说明IDA没有将其当作函数。按一下P创建函数，再按F5即可。

```
int __fastcall sub_6030C0(unsigned int a1)  //注意这里a1=2
{
```

```
  int result; // eax@12
  char v2[112]; // [sp+10h] [bp-1B0h]@1
  signed int v3; // [sp+14h] [bp-1ACh]@1
  signed int v4; // [sp+18h] [bp-1A8h]@1
  signed int v5; // [sp+1Ch] [bp-1A4h]@1
  int v6; // [sp+20h] [bp-1A0h]@1
  char v7[268]; // [sp+80h] [bp-140h]@1
  char v8[268]; // [sp+87h] [bp-139h]@12 //注意这个+87h和上面的+80h
  int v9; // [sp+18Ch] [bp-34h]@1
  char *v10; // [sp+190h] [bp-30h]@1
  int (__fastcall *v11)(char *, _QWORD); // [sp+198h] [bp-28h]@1
  void (__fastcall *v12)(char *); // [sp+1A0h] [bp-20h]@1
  void (__fastcall *v13)(char *); // [sp+1A8h] [bp-18h]@1
  void (__fastcall *v14)(char *); // [sp+1B0h] [bp-10h]@1
  int i; // [sp+1B8h] [bp-8h]@1
  char v16; // [sp+1BFh] [bp-1h]@1

  v14 = (void (__fastcall *)(char *))((((signed int)a1 + 3752700333LL) ^ 0xDEADBEEFLL) >> 2);//4005d0 -
  v13 = (void (__fastcall *)(char *))((((signed int)a1 + 3752699821LL) ^ 0xDEADBEEFLL) >> 2);//400650 -
  v12 = (void (__fastcall *)(char *))((((signed int)a1 + 3752697657LL) ^ 0xDEADBEEFLL) >> 2);//400875 -
  v11 = (int (__fastcall *)(char *, _QWORD))((((signed int)a1 + 3744295917LL) ^ 0xDEADBEEFLL) >> 2);//6
  v10 = v2;
  *(_DWORD *)v2 = 1773735261;
  v3 = 499976301;
  v4 = -111343463;
  v5 = 961141164;
  v6 = 0;
  v12(v2);
  v14(v2);
  v13(v7);    //获取用户的输入
  *(_DWORD *)v10 = 1228499401;
  *((_DWORD *)v10 + 1) = 15571229;
  v12(v2);
  v16 = 1;
  v9 = 0;
  for ( i = 0; v7[i]; ++i )
  {
    if ( i <= 6 && v7[i] != v2[i] )
      v16 = 0;
  }
  if ( v16 )
  {
    *(_DWORD *)v10 = 125;
    v12(v2);
    if ( v7[i - 1] == v2[0] )
      v7[i - 1] = 0;
    else
      v16 = 0;
  }
  if ( v16 != 1 || (result = v11(v8, a1) ^ 1, (_BYTE)result) )
  {
    *(_DWORD *)v10 = -1991698168;
    *((_DWORD *)v10 + 1) = -1443264184;
    *((_DWORD *)v10 + 2) = 0;
    v12(v2);
    result = ((int (__fastcall *)(char *))v14)(v2);
  }
  return result;
}
```

0x400875那个地方的函数其实是一种简单的解密变换，这个变换其实不复杂，可以写个函数来模拟，但目前这个变换并没有作用到我们的输入上，所以可以先不管。用动态跟踪的方式可以方便的得到变换后的结果，我们在**0x400929**（即0x400875函数返回前）设个断点就可以很清楚的看到结果了：

```
[------------------------------------code-------------------------------------]
   0x400921:    mov    eax,DWORD PTR [rbp-0x14]
   0x400924:    cmp    eax,DWORD PTR [rbp-0x18]
   0x400927:    jl     0x4008f3
=> 0x400929:    add    rsp,0x28
   0x40092d:    pop    rbx
   0x40092e:    pop    rbp
   0x40092f:    ret
   0x400930:    nop
[-----------------------------------stack-------------------------------------]
0000| 0x7fffffffdc70 --> 0x100000000
0008| 0x7fffffffdc78 --> 0x7fffffffdcc0 ("inputs you flag:")
0016| 0x7fffffffdc80 --> 0x7fffffffdcc0 ("inputs you flag:")
0024| 0x7fffffffdc88 --> 0x1000000010
0032| 0x7fffffffdc90 --> 0x7fffffffdde0 --> 0x7fffffffdf90 --> 0x0
0040| 0x7fffffffdc98 --> 0x0
0048| 0x7fffffffdca0 --> 0x7fffffffde70 --> 0x7fffffffdf90 --> 0x0
0056| 0x7fffffffdca8 --> 0x6031e6 --> 0x48fffffe50958d48
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000000000400929 in ?? ()
```

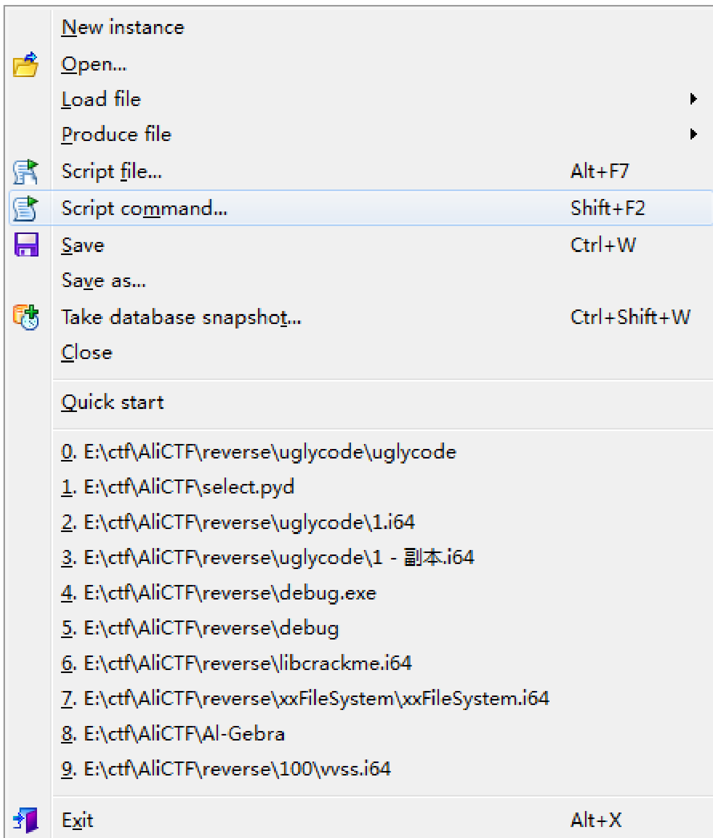
后面会对比用户输入和某个字符串是否相同，同样会调用这个函数，所以我们只要轻轻按一下c就可以了。

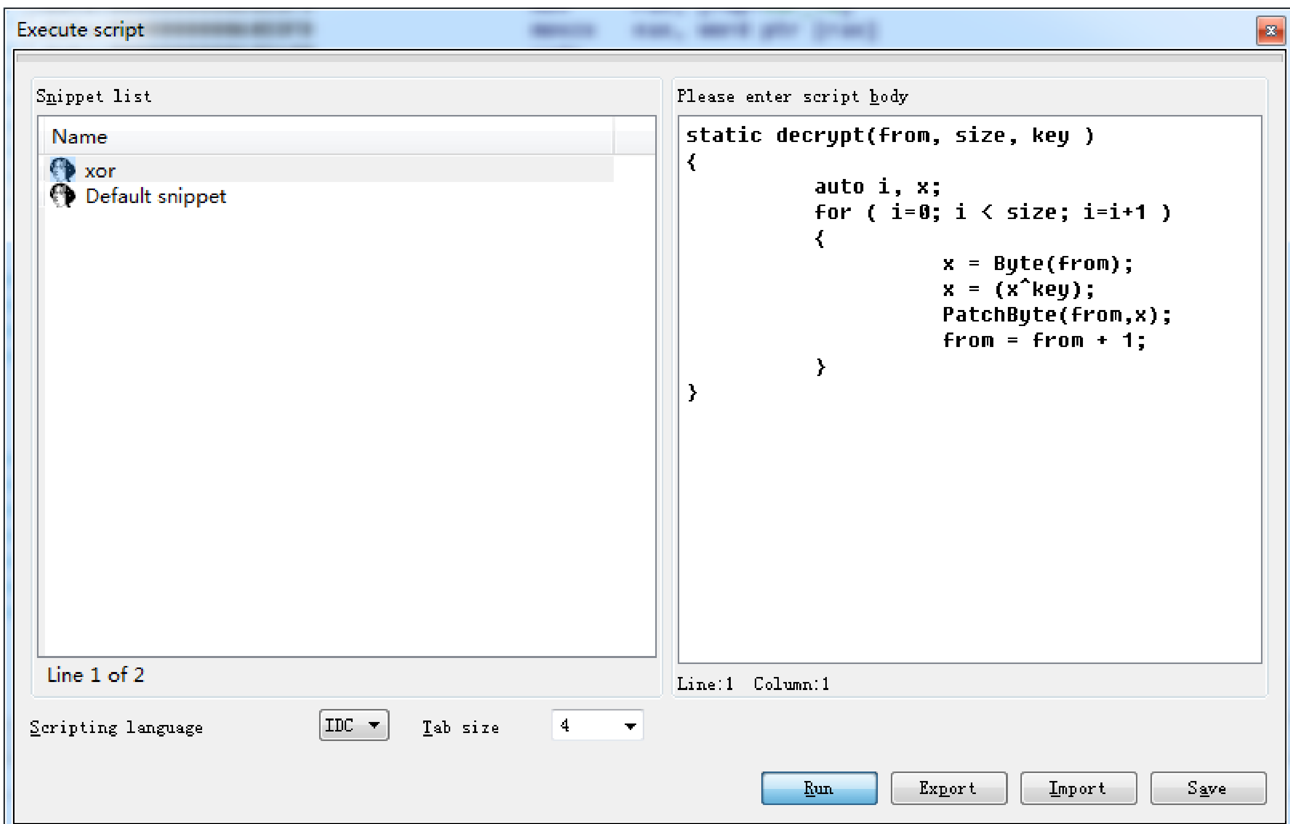
```
[------------------------------------code-------------------------------------]
   0x400921:    mov    eax,DWORD PTR [rbp-0x14]
   0x400924:    cmp    eax,DWORD PTR [rbp-0x18]
   0x400927:    jl     0x4008f3
=> 0x400929:    add    rsp,0x28
   0x40092d:    pop    rbx
   0x40092e:    pop    rbp
   0x40092f:    ret
   0x400930:    nop
[-----------------------------------stack-------------------------------------]
0000| 0x7fffffffdc70 --> 0x7fffffffe070 --> 0x1
0008| 0x7fffffffdc78 --> 0x7fffffffdcc0 --> 0x7b667463696c61 ('alictf{')
0016| 0x7fffffffdc80 --> 0x7fffffffdcc0 --> 0x7b667463696c61 ('alictf{')
0024| 0x7fffffffdc88 --> 0x700000007
0032| 0x7fffffffdc90 --> 0x0
0040| 0x7fffffffdc98 --> 0x0
0048| 0x7fffffffdca0 --> 0x7fffffffde70 --> 0x7fffffffdf90 --> 0x0
0056| 0x7fffffffdca8 --> 0x60322e --> 0xcc45c701ff45c6
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000000000400929 in ?? ()
```

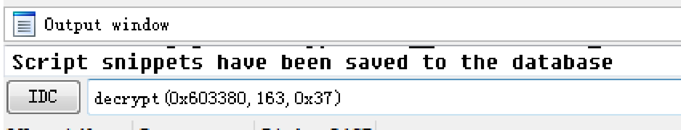
原来是这意思，后面还有判断字符串结尾是'}'（也不难猜到）。最后调用v11，也就是**0x603540**，注意参数，第一个是v8，对比v7可以看出v8=v7+7，相当于去掉前面的"alictf"（也可以动态跟踪来确认），第二个参数还是a1=2。所以点进去看这个函数。

```
int __fastcall sub_603540(__int64 a1, unsigned int a2)
{
  int result; // eax@5
  char *v3; // [sp+20h] [bp-20h]@1
  char *v4; // [sp+28h] [bp-18h]@1
  _BYTE *i; // [sp+38h] [bp-8h]@1
  char *j; // [sp+38h] [bp-8h]@6
  char *k; // [sp+38h] [bp-8h]@11

  v4 = (char *)((((signed int)a2 + 3744296941LL) ^ 0xDEADBEEFLL) >> 2);// 603440
  v3 = (char *)((((signed int)a2 + 3744293869LL) ^ 0xDEADBEEFLL) >> 2);// 603740
  for ( i = (_BYTE *)((((signed int)a2 + 3744297197LL) ^ 0xDEADBEEFLL) >> 2);// 603380
        i != (_BYTE *)(((((signed int)a2 + 3744297197LL) ^ 0xDEADBEEFLL) >> 2) + 163);
        ++i )
  {
    *i ^= 0x37u;
  }
  if ( (unsigned __int8)((int (__fastcall *)(__int64, _QWORD))((((signed int)a2 + 3744297197LL) ^ 0xDEA
                          a1,
                          a2) ^ 1 )
  {
    result = 0;
  }
  else
  {
    for ( j = (char *)((((signed int)a2 + 3744296941LL) ^ 0xDEADBEEFLL) >> 2); j != v4 + 250; ++j )
      *j ^= 0x49u;
    if ( (unsigned __int8)((int (__fastcall *)(__int64, _QWORD))v4)(a1 + 2, a2) ^ 1 )
    {
      result = 0;
    }
    else
    {
      for ( k = (char *)((((signed int)a2 + 3744293869LL) ^ 0xDEADBEEFLL) >> 2); k != v3 + 672; ++k )
        *k ^= *(_BYTE *)(a1 + 5) ^ *(_BYTE *)(a1 + 6);
      result = ((int (__fastcall *)(__int64, _QWORD))v3)(a1 + 6, a2);
    }
  }
  return result;
}
```

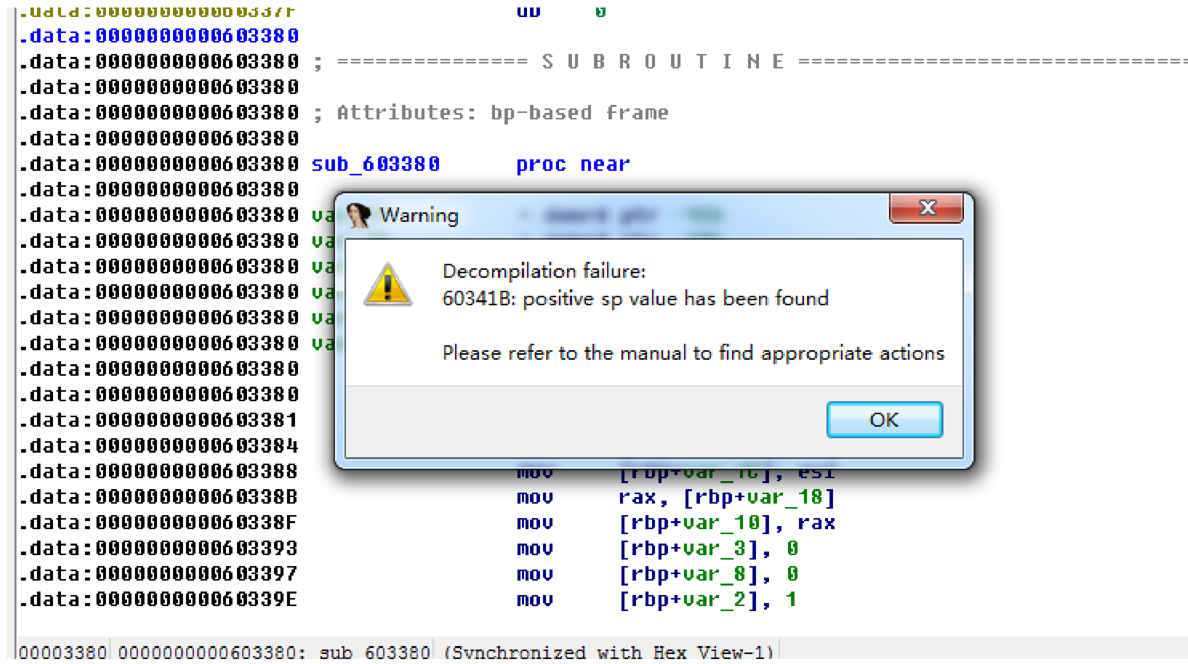
程序对**0x603380**处163字节进行了异或操作，再调用之。说明这个函数曾经进行了加密操作，可以写个脚本去修改文件，不过在写Writeup的时候发现了更好的方法。


New instance
Open...
Load file
Produce file
Script file... Alt+F7
Script command... Shift+F2
Save Ctrl+W
Save as...
Take database snapshot... Ctrl+Shift+W
Close
Quick start
0. E:\ctf\AliCTF\reverse\uglycode\uglycode
1. E:\ctf\AliCTF\select.pyd
2. E:\ctf\AliCTF\reverse\uglycode\1.i64
3. E:\ctf\AliCTF\reverse\uglycode\1 - 副本.i64
4. E:\ctf\AliCTF\reverse\debug.exe
5. E:\ctf\AliCTF\reverse\debug
6. E:\ctf\AliCTF\reverse\libcrackme.i64
7. E:\ctf\AliCTF\reverse\xxFileSystem\xxFileSystem.i64
8. E:\ctf\AliCTF\Al-Gebra
9. E:\ctf\AliCTF\reverse\100\vvss.i64
Exit Alt+X

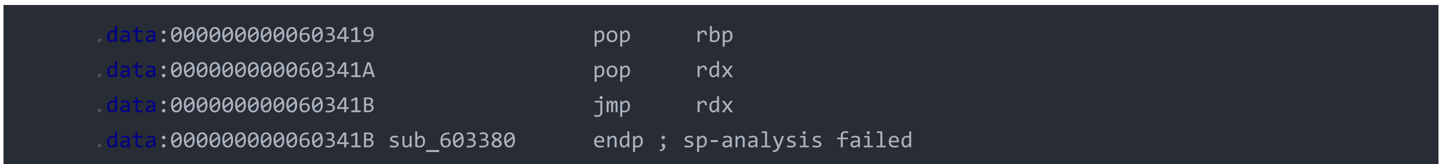

Execute script
Snippet list
Name
xor
Default snippet
Line 1 of 2
Please enter script body
static decrypt(from, size, key)
{
auto i, x;
for (i=0; i < size; i=i+1)
{
x = Byte(from);
x = (x^key);
PatchByte(from,x);
from = from + 1;
}
}
Line:1 Column:1
Scripting language
IDC
Tab size
4
Run
Export
Import
Save



哗的一下就解密了，真牛X，比手动改文件不知道高到哪里去了。然后去这个解密后的函数那里看看。



分析失败，堆栈不平衡。到函数底部发现：



```
.data:0000000000603419                  pop     rbp
.data:000000000060341A                  pop     rdx
.data:000000000060341B                  jmp     rdx
.data:000000000060341B sub_603380       endp ; sp-analysis failed
```

其实，`pop rdx` + `jmp rdx` 不就是 `ret` 么？（我不相信rdx的值会被用到）于是修改一下，让IDA不会分析错误：

```
IDC>PatchByte(0x60341A,0xC3) //0xC3是ret
IDC>PatchByte(0x60341B,0x90) //0x90是nop
```

接着按F5就可以分析了。

```
__int64 __fastcall sub_603380(_WORD *a1)
{
  int v2; // [sp+14h] [bp-8h]@1
  char v3; // [sp+19h] [bp-3h]@1
  signed __int16 i; // [sp+1Ah] [bp-2h]@1

  v3 = 0;
  v2 = 0;
  for ( i = 1; i < *a1 + 1; ++i )
  {
    if ( i & 1 )
    {
      if ( i == 3 * (((unsigned int)(21846 * i) >> 16) - (i >> 15)) )
        ++v2;
    }
    else
    {
      ++v2;
    }
  }
  if ( v2 == 19509 )
    v3 = 1;
  return (unsigned __int8)v3;
}
```

注意一下，`a1`是我们输入字符串除掉”alictf{“的首地址，这里强制为`WORD`类型，就相当于拿到前两个字节。上面这段代码还是挺好理解的，可以写个脚本拿到这两个字节的内容了：

```
>>> ans=0
>>> for i in range(1,0xffff):
    if i%2==1 and i==3*(((21846*i)>>16)-(i>>15)):
        ans+=1
    elif i%2==0:
        ans+=1
    if ans==19509:
        print(hex(i))
        break

0x7250
>>> chr(0x50)
'P'
>>> chr(0x72)
'r'
```

回到原来的函数，继续解密：

```
IDC>decrypt(0x603440,250,0x49)
```

来到**0x603440**（也需要修正返回处代码）：

```
__int64 __fastcall sub_603440(_DWORD *a1)
{
  __int64 v2; // [sp+1Ch] [bp-10h]@1
  char v3; // [sp+27h] [bp-5h]@1
  signed int i; // [sp+28h] [bp-4h]@1

  v3 = 0;
  v2 = 0LL;
  for ( i = 1; *a1 + 1 > i; ++i )
  {
    if ( i & 1 )
    {
      if ( i % 3 )
      {
        if ( i % 5 )
        {
          if ( i == 7 * (i / 7) )
            ++v2;
        }
        else
        {
          ++v2;
        }
      }
      else
      {
        ++v2;
      }
    }
    else
    {
      ++v2;
    }
  }
  if ( v2 == 665543088 )
    v3 = 1;
  return (unsigned __int8)v3;
}
```

注意这里传参的时候是前面的 `a1` +2，就是接下来的字符，因为是 `DWORD` 所以这次我们可以拿到4个字符。这段代码的意思是，[1,n]的整数中能被2,3,5,7中至少一个整除的共有665543088个，求n。这次因为DWORD的范围比较大，直接遍历恐怕会比较耗时，所以用容斥原理结合二分法来做。

```
>>> def f(n):#小于等于n的被2,3,5或7整除的整数个数
    s=0
    s+=n//2
    s+=n//3
    s+=n//5
    s+=n//7
    s-=n//6
    s-=n//10
    s-=n//14
    s-=n//15
    s-=n//21
    s-=n//35
    s+=n//30
    s+=n//42
    s+=n//70
    s+=n//105
    s-=n//210
    return s

>>> def find(a,b):#寻找a,b之间满足题意的数，因为f是不减函数所以可以这样
    if a+1>=b:
        return a
    ans=665543088
    c=(a+b)//2
    n=f(c)
    if n==ans:
        return c
    if n>ans:
        return find(a,c)
    return find(c,b)

>>> hex(find(1,0xffffffff))
'0x336c6230'
>>> 0x336c6230.to_bytes(4,'little').decode()
'0bl3'
```

这样连起来就是”Pr0bl3”，下一个不出意外的话是”m”咯？

回到上一层函数，下面又要对**0x603740**异或解密了，为了方便我这里再引用一下代码：

```
for ( k = (char *)((((signed int)a2 + 3744293869LL) ^ 0xDEADBEEFLL) >> 2); k != v3 + 672; ++k )
    *k ^= *(_BYTE *)(a1 + 5) ^ *(_BYTE *)(a1 + 6);
```

a1 + 5 处就是’3’，a1 + 6 我不知道啊？这次他怎么不按基本法来？我当时猜了是”m”结果不对……然后可以猜0x603740处的值异或之后为0x55，因为这是 `push ebp`，函数体开头基本都有这句话；此外前面两个函数异或之后都是0x55开头的。

```
IDC>0x2b^0x55
        126.       7Eh          176o
IDC>decrypt(0x603740,672,126)
```

果然没错，顺便可以算一下第7个字符，原来是大写的’M’……现在可以进入下一层函数了。注意调用下一层函数的时候，参数 `a1` 又被加了6，所以是从M开始的字串。

```c
int __fastcall sub_603740(__int64 a1, unsigned int a2)
{
  int result; // eax@2
  char v3; // [sp+10h] [bp-120h]@1
  signed int v4; // [sp+14h] [bp-11Ch]@1
  signed int v5; // [sp+18h] [bp-118h]@1
  signed int v6; // [sp+1Ch] [bp-114h]@1
  int v7; // [sp+20h] [bp-110h]@1
  char v8; // [sp+80h] [bp-B0h]@1
  int (__fastcall *v9)(_QWORD, _QWORD); // [sp+E8h] [bp-48h]@6
  _BYTE *v10; // [sp+F0h] [bp-40h]@3
  char *v11; // [sp+F8h] [bp-38h]@1
  int (__fastcall *v12)(_QWORD, _QWORD); // [sp+100h] [bp-30h]@1
  int (__fastcall *memcmpf)(char *, char *, signed __int64); // [sp+108h] [bp-28h]@1
  void (__fastcall *v14)(__int64, char *); // [sp+110h] [bp-20h]@1
  void (__fastcall *v15)(char *); // [sp+118h] [bp-18h]@1
  void (__fastcall *putsf)(char *); // [sp+120h] [bp-10h]@1
  _BYTE *v17; // [sp+128h] [bp-8h]@3

  putsf = (void (__fastcall *)(char *))((((signed int)a2 + 3752700333LL) ^ 0xDEADBEEFLL) >> 2);// 4005d
  v15 = (void (__fastcall *)(char *))((((signed int)a2 + 3752697657LL) ^ 0xDEADBEEFLL) >> 2);// 400875
  v14 = (void (__fastcall *)(__int64, char *))((((signed int)a2 + 3752698741LL) ^ 0xDEADBEEFLL) >> 2);/
  memcmpf = (int (__fastcall *)(char *, char *, signed __int64))((((signed int)a2 + 3752699565LL) ^ 0xD
  v12 = (int (__fastcall *)(_QWORD, _QWORD))((((signed int)a2 + 3744290541LL) ^ 0xDEADBEEFLL) >> 2);//
  v11 = &v3;
  *(_DWORD *)&v3 = 1555013445;
  v4 = 1322911880;
  v5 = 729268039;
  v6 = -1545661451;
  v7 = 0;
  v15(&v3);
  v14(a1, &v8);
  if ( memcmpf(&v3, &v8, 16LL) )
  {
    result = 0;
  }
  else
  {
    *(_DWORD *)v11 = 1845058824;
    *((_DWORD *)v11 + 1) = -917915384;
    *((_DWORD *)v11 + 2) = -648471288;
    *((_DWORD *)v11 + 3) = 489249032;
    *((_DWORD *)v11 + 4) = -1724298791;
    *((_DWORD *)v11 + 5) = 141138184;
    *((_DWORD *)v11 + 6) = 1845058824;
    *((_DWORD *)v11 + 7) = -637990663;
    *((_DWORD *)v11 + 8) = 185;
    v15(&v3);
    putsf(&v3);
    v17 = v12;
    v10 = (char *)v12 + 349;
    while ( v17 != v10 )
      *v17++ ^= 0xEFu;
    v9 = v12;
    result = v12(a1 + 5, a2);
  }
  return result;
}
```

其中**0x400766**是陌生的函数，点击去经过几层会来到**0x4014FE**，这个函数一看就和`md5`有关（如果看不出，可以去网上搜一下用到的常数，比如第一个*680876936*）。最后通过大胆猜想和动态跟踪验证，可以确定0x400766整个函数就是用来对一个字符串进行md5变换。

跟着原来在0x400875里面设过的断点继续（注意不要设断点在被解密函数内部，原理清楚吧），先拿到md5变换后正确的值，为`35faf651b1a72022e8ddfed1caf7c45f`，放cmd5上果然解不开……

```
[-------------------------------------code-------------------------------------]
   0x400921:    mov    eax,DWORD PTR [rbp-0x14]
   0x400924:    cmp    eax,DWORD PTR [rbp-0x18]
   0x400927:    jl     0x4008f3
=> 0x400929:    add    rsp,0x28
   0x40092d:    pop    rbx
   0x40092e:    pop    rbp
   0x40092f:    ret
   0x400930:    nop
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdae0 --> 0x554e47 ('GNU')
0008| 0x7fffffffdae8 --> 0x7fffffffdb30 --> 0x2220a7b151f6fa35
0016| 0x7fffffffdaf0 --> 0x7fffffffdb30 --> 0x2220a7b151f6fa35
0024| 0x7fffffffdaf8 --> 0x1000000010
0032| 0x7fffffffdb00 --> 0x0
0040| 0x7fffffffdb08 --> 0x0
0048| 0x7fffffffdb10 --> 0x7fffffffdc50 --> 0x7fffffffdca0 --> 0x7fffffffde70 --> 0x7fffffffdf90 --> 0x0
0056| 0x7fffffffdb18 --> 0x60389b --> 0x48fffff508d8d48
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000000000400929 in ?? ()
gdb-peda$ x/16b 0x7fffffffdb30
0x7fffffffdb30: 0x35    0xfa    0xf6    0x51    0xb1    0xa7    0x20    0x22
0x7fffffffdb38: 0xe8    0xdd    0xfe    0xd1    0xca    0xf7    0xc4    0x5f
```

现在来验证一下0x400766是不是md5函数：

```
RAX: 0x400766 (push   rbp)
RBX: 0x0
RCX: 0x7fffffffdba0 --> 0x100000000
RDX: 0x7fffffffdd3d --> 0x5f616861685f4d ('M_haha_')
RSI: 0x7fffffffdba0 --> 0x100000000
RDI: 0x7fffffffdd3d --> 0x5f616861685f4d ('M_haha_')
RBP: 0x7fffffffdc50 --> 0x7fffffffdca0 --> 0x7fffffffde70 --> 0x7fffffffdf90 --> 0x0
RSP: 0x7fffffffdb20 --> 0x21c93bb9d
RIP: 0x6038b3 --> 0xffff50b58d48d0ff
R8 : 0xf0c0
R9 : 0x0
R10: 0x15
R11: 0x246
R12: 0x400670 (xor    ebp,ebp)
R13: 0x7fffffffe070 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x6038a9:    mov    rax,QWORD PTR [rbp-0x20]
   0x6038ad:    mov    rsi,rcx
   0x6038b0:    mov    rdi,rdx
=> 0x6038b3:    call   rax
   0x6038b5:    lea    rsi,[rbp-0xb0]
   0x6038bc:    lea    rcx,[rbp-0x120]
```

我输入的flag是"alictf{Pr0bl3M_haha_}"，这里最后的'}'没有存在，应该是前面哪个地方把它调成0了我没有留意。对比下确实是md5：

```
   0x6038b5:    lea    rsi,[rbp-0xb0]
   0x6038bc:    lea    rcx,[rbp-0x120]
=> 0x6038c3:    mov    rax,QWORD PTR [rbp-0x28]
   0x6038c7:    mov    edx,0x10
   0x6038cc:    mov    rdi,rcx
   0x6038cf:    call   rax
   0x6038d1:    test   eax,eax
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffdb20 --> 0x21c93bb9d
0008| 0x7fffffffdb28 --> 0x7fffffffdd3d --> 0x5f616861685f4d ('M_haha_')
0016| 0x7fffffffdb30 --> 0x2220a7b151f6fa35
0024| 0x7fffffffdb38 --> 0x5fc4f7cad1fedde8
0032| 0x7fffffffdb40 --> 0x7fff00000000
0040| 0x7fffffffdb48 --> 0x7ffff7ff74c0 --> 0x7ffff7a15000 --> 0x10102464c457f
0048| 0x7fffffffdb50 --> 0x7ffff7a25c58 --> 0xc00120000155e
0056| 0x7fffffffdb58 --> 0x1f25bc2
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x00000000006038c3 in ?? ()
gdb-peda$ x/16b $rsi
0x7fffffffdba0: 0x87    0x0b    0x9c    0xe6    0xff    0x7b    0xdc    0x11
0x7fffffffdba8: 0xf9    0xd8    0x62    0xbc    0xdb    0x2e    0x94    0xb6
```

```
        s+=n//105
        s-=n//210
        return s

>>> find(1,0xffffffff)
862741040
>>> hex(find(1,0xffffffff))
'0x336c6230'
>>> 0x336c6230.to_bytes(4,'little').decode()
'0bl3'
>>>
=============== RESTART: E:\ctf\AliCTF\reverse\uglycod
0x4005d0
>>>
=============== RESTART: E:\ctf\AliCTF\reverse\uglycod
0x400875
>>>
=============== RESTART: E:\ctf\AliCTF\reverse\uglycod
0x400766
>>>
=============== RESTART: E:\ctf\AliCTF\reverse\uglycod
0x400610
>>>
=============== RESTART: E:\ctf\AliCTF\reverse\uglycod
0x603a00
>>> import hashlib
>>> m=hashlib.md5()
>>> m.update('M_haha_'.encode())
>>> m.hexdigest()
'870b9ce6ff7bdc11f9d862bcdb2e94b6'
>>>
```

由于我们无法解出，所以先跳过这里看下面。这个函数也要解密，解密后：

```
signed __int64 __fastcall sub_603A00(__int64 a1, int a2)
{
  signed __int64 result; // rax@2
  char v3; // [sp+10h] [bp-90h]@1
  signed int v4; // [sp+14h] [bp-8Ch]@1
  signed int v5; // [sp+18h] [bp-88h]@1
  char *v6; // [sp+80h] [bp-20h]@1
  int (__fastcall *strcmpf)(char *, __int64); // [sp+88h] [bp-18h]@1
  void (__fastcall *v8)(char *); // [sp+90h] [bp-10h]@1
  void (__fastcall *putsf)(char *); // [sp+98h] [bp-8h]@1

  putsf = (void (__fastcall *)(char *))(((a2 + 3752700333LL) ^ 0xDEADBEEFLL) >> 2);// 4005d0
  v8 = (void (__fastcall *)(char *))(((a2 + 3752697657LL) ^ 0xDEADBEEFLL) >> 2);// 400875
  strcmpf = (int (__fastcall *)(char *, __int64))(((a2 + 3752699501LL) ^ 0xDEADBEEFLL) >> 2);// 400620
  v6 = &v3;
  *(_DWORD *)&v3 = 1095556380;
  v4 = 1842214177;
  v5 = 5869004;
  v8(&v3);
  if ( strcmpf(&v3, a1) )
  {
    result = 0LL;
  }
  else
  {
    *(_DWORD *)v6 = -1184249511;
    *((_DWORD *)v6 + 1) = 145357193;
    *((_DWORD *)v6 + 2) = 72;
    v8(&v3);
    putsf(&v3);
    result = 1LL;
  }
  return result;
}
```

注意这里参数 `a1` 传入的时候加上了5。直接判断字符串相等，v3的值可以动态调试得到。

```
   0x6038cc:    mov    rdi,rcx
   0x6038cf:    call   rax
=> 0x6038d1:    test   eax,eax
   0x6038d3:    je     0x6038df
   0x6038d5:    mov    eax,0x0
   0x6038da:    jmp    0x6039d6
   0x6038df:    mov    rax,QWORD PTR [rbp-0x38]
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdb20 --> 0x21c93bb9d
0008| 0x7fffffffdb28 --> 0x7fffffffdd3d --> 0x5f616861685f4d ('M_haha_')
0016| 0x7fffffffdb30 --> 0x2220a7b151f6fa35
0024| 0x7fffffffdb38 --> 0x5fc4f7cad1fedde8
0032| 0x7fffffffdb40 --> 0x7fff00000000
0040| 0x7fffffffdb48 --> 0x7ffff7ff74c0 --> 0x7ffff7a15000 --> 0x10102464c457f
0048| 0x7fffffffdb50 --> 0x7ffff7a25c58 --> 0xc00120000155e
0056| 0x7fffffffdb58 --> 0x1f25bc2
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x00000000006038d1 in ?? ()
gdb-peda$ set $rax=0
```

上图中判断相等的地方，把`rax`改成0可以跳过检测，接着再按一下c得到`you can get the flag if you go on`，这里不是，再按c，拿到字符串`A1w4ys_H3re`。

所以flag的形式为`alictf{Pr0bl3M????A1w4ys_H3re}`，且`M????A1w4ys_H3re`的md5值为`35faf651b1a72022e8ddfed1caf7c45f`，写个脚本搞一搞。

```
def hack():
    s=[chr(i) for i in range(0x20,0x7f)]
    s.insert(0,'_') #因为猜测第一个未知字符是'_'，所以这么写加快速度
    for i in s:
        for j in s:
            for k in s:
                for l in s:
                    out='M'+i+j+k+l+'A1w4ys_H3re'
                    m=hashlib.md5()
                    m.update(out.encode())
                    if m.hexdigest()=='35faf651b1a72022e8ddfed1caf7c45f':
                        print(out)

hack()
M_1s_A1w4ys_H3re
```

答案是`alictf{Pr0bl3M_1s_A1w4ys_H3re}`。