190505 逆向-DDCTF2019(Reverse)

原创

奈沙夜影 于 2019-05-06 00:08:37 发布 9921 收藏 1

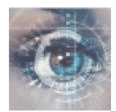
分类专栏： CrackMe CTF

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/whklhhhh/article/details/89859382
版权

CrackMe 同时被 2 个专栏收录

83 篇文章 2 订阅

订阅专栏


CTF

163 篇文章 4 订阅

订阅专栏

咕咕咕咕咕

连续若干CTF以后就是各种考试作业DDL催命_(:3」∠)_

等这两周各种考察课完了慢慢补各种活儿吧~

先交一下之前的DDWP-0-

还请各位大佬多指正~

Windows Reverse1

通过段名发现是UPX壳，upx -d脱壳后进行分析
核心函数只是通过data数组做一个转置，反求index即可
值得一说的是data的地址与实际数组有一些偏移
由于输入的可见字符最小下标就是空格的0x20，因此data这个地址实际上也是真正的表地址(0xb03018)-0x20=0xb02ff8，在实际反查的时候需要稍作处理

```
.rdata:00B02FF8                                ; char data[]
.rdata:00B02FF8 ??                              data            db ?                    ; DATA XR
.rdata:00B02FF9 ?? ?? ?? ?? ?? ??+                              align 10h
.rdata:00B02FF9 ??                              _rdata          ends
.rdata:00B02FF9
.data:00B03000                                 ; Section 3. (virtual address 00003000)
.data:00B03000                                 ; Virtual size                  : 000003E4 (    99
.data:00B03000                                 ; Section size in file          : 00000200 (    51
.data:00B03000                                 ; Offset to raw data for section: 00001600
.data:00B03000                                 ; Flags C0000040: Data Readable Writable
.data:00B03000                                 ; Alignment     : default
.data:00B03000                                 ; ================================================
.data:00B03000
.data:00B03000                                 ; Segment type: Pure data
.data:00B03000                                 ; Segment permissions: Read/Write
.data:00B03000                                 _data           segment para public 'DATA' use32
.data:00B03000                                                 assume cs:_data
.data:00B03000                                                 ;org 0B03000h
.data:00B03000 4E E6 40 BB                     ___security_cookie dd 0BB40E64Eh        ; DATA XRE
.data:00B03000                                                                         ; __securi
.data:00B03004 B1 19 BF 44                     dword_B03004    dd 44BF19B1h            ; DATA XRE
.data:00B03004                                                                         ; ___secur
.data:00B03008 FF                                              db 0FFh
.data:00B03009 FF                                              db 0FFh
.data:00B0300A FF                                              db 0FFh
.data:00B0300B FF                                              db 0FFh
.data:00B0300C FF                                              db 0FFh
.data:00B0300D FF                                              db 0FFh
.data:00B0300E FF                                              db 0FFh
.data:00B0300F FF                                              db 0FFh
.data:00B03010 FE FF FF FF                     dword_B03010    dd 0FFFFFFFEh           ; DATA XRE
.data:00B03014 01 00 00 00                     dword_B03014    dd 1                    ; DATA XRE
.data:00B03018 7E 7D 7C 7B 7A 79+              aZyxwvutsrqponm db '~}|{zyxwvutsrqponmlkjihgfedcba
```

```python
from ida_bytes import get_bytes
data = get_bytes(0xb03018,0xb03078-0xb03018)
r = "DDCTF{reverseME}"
s = ""
for i in range(len(r)):
  s += chr(data.index(r[i])+0xb03018-0xb02ff8)
print(s)
```

Windows Reverse 2

用PEID查壳发现是aspack壳，直接运行程序，然后用调试器附加上去
通过调用堆栈来追溯到输入函数
具体方法为:
当程序运行到等待输入时，在调试器中按下暂停，然后选择主线程，观察调用堆栈(Stack Trace / Call Stack)窗口

Address	Module	Function
777EA91C	ntdll.dll	ntdll_ZwReadFile+C
644E8EEC	msvcr90.dll	msvcr90__wsopen_s+222
644E9371	msvcr90.dll	msvcr90__read+BB
644AEFCD	msvcr90.dll	msvcr90__filbuf+78
644D0110	msvcr90.dll	msvcr90__vcwprintf_s+102B
644CF49C	msvcr90.dll	msvcr90__vcwprintf_s+3B7
644B6341	msvcr90.dll	msvcr90__wprintf_p+74
644B26E7	msvcr90.dll	msvcr90_scanf+13
00011000	reverse2_fi…	00011000
774C8492	kernel32.dll	kernel32_BaseThreadInitThunk+22
777E41C6	ntdll.dll	ntdll_RtlAreBitsSet+86
777E4193	ntdll.dll	ntdll_RtlAreBitsSet+53

可以看出调用堆栈中存在scanf函数，这是因为接收输入时程序阻塞，必然是在程序调用接收输入的函数过程中阻塞的。而根据栈帧的机制，函数返回值会被存放在各个栈帧中，所以scanf的上一个函数就是用户模块了。

双击即可跟进去，此时看到的是一片数据

```
.text:00011000 _text segment para public 'CODE' use32
.text:00011000 assume cs:_text
.text:00011000 ;org 11000h
.text:00011000 assume es:debug013, ss:debug013, ds:_data, fs:nothing, gs:nothing
.text:00011000 dd 7968FF6Ah, 6400011Ch, 0A1h, 0EC835000h, 3000A12Ch, 0C4330001h, 28244489h
.text:00011000 dd 57565553h, 13000A1h, 50C43300h, 4024448Dh, 0A364h, 6C8B0000h, 0F98B5024h
.text:00011000 dd 5424448Bh, 20244C8Dh, 1C244489h, 204015FFh, 0F6330001h, 48247489h, 840FEE3Bh
.text:00011000 dd 129h, 9B8D06EBh, 0, 4C880F8Ah, 5C8A1434h, 4D461524h, 3FE8347h, 448A6175h
.text:00011000 dd 0D08A1424h, 2402EAC0h, 4E0C003h, 18245488h, 0E9C0CB8Ah, 88C10204h, 8A192444h
.text:00011000 dd 8A162444h, 0FE280D3h, 0C88AD202h, 0E9C0D202h, 24D10206h, 2454883Fh
.text:00011000 dd 2444881Ah, 90F6331Bh, 3454B60Fh, 20828A18h, 34000130h, 0C8B60F76h, 244C8D51h
.text:00011000 dd 3C15FF24h, 46000120h, 7C04FE83h, 85F633DFh, 858975EDh, 0A2840FF6h, 83000000h
.text:00011000 dd 1A7D03FEh, 3BAh, 52D62B00h, 1834448Dh, 73E85055h, 8A00000Bh, 8321245Ch
.text:00011000 dd 448A0CC4h, 0C88A1424h, 324D38Ah, 0C004E0C0h, 0C20204EAh, 8002E9C0h
```

这是因为代码由动态解密得到，IDA还没有对他们进行分析。

对着它们按C，即可将其作为Code识别，进行分析了

然后重新看调用堆栈窗口的具体地址即可找到目标

也可以直接在Options-general窗口中找到`Reanalyse program`按钮进行重新分析

```
11  sub_11C6A(&v4, 0, 1023);
12  v5 = 0;
13  sub_11C6A(&v6, 0, 1023);
14  ((void (__cdecl *)(const char *))msvcr90_printf)("input code:");
15  ((void (*)(const char *, ...))msvcr90_scanf)((const char *)&unk_12130, &v3);
16  if ( !(unsigned __int8)sub_111F0() )
17  {
18    ((void (__cdecl *)(const char *))msvcr90_printf)("invalid input\n");
19    ((void (__stdcall *)(_DWORD))msvcr90_exit)(0);
20  }
21  sub_11240(&v5);
22  v1 = 0;
23  sub_11C6A(&v2, 0, 1023);
24  ((void (*)(char *, const char *, ...))msvcr90_sprintf)(&v1, "DDCTF{%s}", &v5);
25  if ( !strcmp(&v1, "DDCTF{reverse+}") )
26    ((void (*)(const char *, ...))msvcr90_printf)("You've got it !!! %s\n", &v1);
27  else
28    ((void (__cdecl *)(const char *))msvcr90_printf)("Something wrong. Try again...\n");
```

关键函数是sub_111f0和sub_11240
跟踪数据变化可以直接看出两个函数分别是hexdecode和base64encode
反向计算得到flag

```
from base64 import b64decode
print(b64decode(b"reverse+").hex().upper())
```

后来发现可以用看雪的脱壳工具脱壳，可读性强很多

Confused

macOS程序
Main函数中没有逻辑，所以通过字符串查找"DDCTF"取两次交叉引用，找到`ViewController checkCode:`函数
前段主要是校验开头结尾的"DDCTF{}"标志，API的各种命名都很清晰，无需多言

关键部分在`msgSend`“onSuccess"前的最后一个判断函数sub_1000011d0中

第一个调用sub_100001f60可以看出来是构造函数，将对象的各个成员进行初始化，并把输入即a2的18个字节拷贝到全局变量中
第二个调用sub_100001f00则开始虚拟机的执行循环，初始化IP寄存器后即不断向后执行，直到碰到结尾字节0xf3为止
loop函数内部很清晰，遍历对象的所有opcode与当前IP所指的值相比较，相等时即执行对应函数，函数内部负责后移IP，使VM执行下一条指令

平常做VM可以直接上angr、pintools或者记录运行log来方便地处理，但是本题是mac平台，由于钱包原因就只能乖乖静态逆了
首先整理字节码，位于0x100001984的地方
提出部分以作示例

```
0xf0, 0x10, 0x66, 0x0, 0x0, 0x0,
0xf8,
0xf2, 0x30,
0xf6, 0xc1,
0xf0, 0x10, 0x63, 0x0, 0x0, 0x0,
0xf8,
0xf2, 0x31,
0xf6, 0xb6,
```

第一个code是`0xf0`，对应的handler为sub_100001d70

```
1 vm *__fastcall f_f0(vm *a1)
2 {
3   vm *result; // rax
4   int *v2; // [rsp+Ch] [rbp-18h]
5
6   v2 = (int *)(*(_QWORD *)&a1->ptr_op + 2LL);
7   switch ( *(unsigned __int8 *)(*(_QWORD *)&a1->ptr_op + 1LL) )
8   {
9     case 0x10u:
10      a1->a = *v2;
11      break;
12    case 0x11u:
13      a1->b = *v2;
14      break;
15    case 0x12u:
16      a1->c = *v2;
17      break;
18    case 0x13u:
```

```
19      a1->d = *v2;
20      break;
21    case 0x14u:
22      a1->a = mem[*v2];
23      break;
24    default:
25      break;
26  }
27  result = a1;
28  *(_QWORD *)&a1->ptr_op += 6LL;
```

通过structures结构体功能可以将对象的成员重命名、整理结构成比较易读的形式
具体方法为在Structures窗口中按Insert创建结构体，按D创建成员

```
00000000 vm              struc ; (sizeof=0xB4, mappedto_59)
00000000 a               dd ?
00000004 b               dd ?
00000008 c               dd ?
0000000C d               dd ?
00000010 flags           dd ?
00000014 field_14        dd ?
00000018 ptr_op          dd ?
0000001C field_1C        dd ?
00000020 op_list         op 9 dup(?)
000000B0 f               dd ?
000000B4 vm              ends
000000B4
00000000 ; ---------------------------------------------------
00000000
00000000 op              struc ; (sizeof=0x10, mappedto_60)
00000000                                         ; XREF: vm/r
00000000 opcode          db ?
00000001                 db ? ; undefined
00000002                 db ? ; undefined
00000003                 db ? ; undefined
00000004                 db ? ; undefined
00000005                 db ? ; undefined
00000006                 db ? ; undefined
00000007                 db ? ; undefined
00000008 func            dq ?
00000010 op              ends
00000010
```

最后将a1的类型按y重定义成结构体指针 `vm*` 即可

可以看出来f0根据第二个字节来决定将后4个字节的Int存入某个寄存器中，例如0x10表示a
然后是`0xf8`，对应的handler为sub_100001C60

```
1 __int64 __fastcall f_f8(vm *a1)
2 {
3   __int64 result; // rax
4 
5   result = caser_add(a1->a, 2);
6   a1->a = (char)result;
7   ++*(_QWORD *)&a1->ptr_op;
8   return result;
9 }
```

即对a寄存器的值在字母域内加2，相当于ROT2吧

以此类推，`0xf2`是判断a寄存器内的值是否和全局变量中以后一个字节为偏移的值相等，实际上也就是刚才memcpy进来的input
`0xf6`则是根据`0xf2`判断的结果来决定是否跳转，下同循环
因此整个算法实际上就是逐字符判断定值+2是否与输入相等，只需要将code_array中的定值取出即可
例如一个正则表达式

```
code="""0xf0, 0x10, ... 略"""
import re
data = re.findall(r"\n?0xf0, \n?0x10, \n?0x(..)", code)

print(data)
for i in data:
    v = int(i,16)+2
    print(chr(v-26 if (v>ord('Z') and v<ord('a')) or v>ord('z') else v),end='')
```

obfuscating macros

本题是一个经过OLLVM混淆的较简单算法的程序

基础分析

主函数中通过cin接受输入，传入两个函数中处理后要求皆返回True
两个函数都被OLLVM了
第一个函数通过黑盒可以知道是HexDecode，输出仍保存在原来的位置里，若输入超出数字和大写ABCDEF则Decode失败返回False
第二个函数则是对输入进行了一些比较，输出比较的结果

有下列几种思路解题：

1. 动态调试
2. 单字节穷举
3. 反混淆

动态调试

对于被控制流平坦化处理过的程序，执行流完全打散，所以很难知道各个代码块之间的关系，再加上虚假执行流会污染代码块，使得同一个真实块出现多次，难以分辨真实代码
因此在不deflat的情况下最好的办法就是单步执行慢慢跟，等到执行真实代码，尤其是一些运算的时候稍作注意
本题中通过这样的办法发现了这样一处代码

```
291      }
292      v23 = (signed int)sub_4082B6(1975487151);
293      *(_QWORD *)sub_4086E2((__int64)&v53, (__int64)&v23) = &loc_405FA3;
294      if ( v50 )
295      {
296        v4 = v24++;                                    // input
297        *v26 -= *v4;
298        if ( !v12 )
299          v12 = 162LL;
300        if ( !v50 )
301        {
302          v3 = *(_QWORD *)sub_408712((__int64)&v53, (__int64)&v14);
303          goto LABEL_4;
304        }
305      }
306      v23 = 161LL;
307      *(_QWORD *)sub_4086E2((__int64)&v53, (__int64)&v23) = &loc_40607F;
308      v23 = (signed int)sub_4082B6(1975487145);
309      *(_QWORD *)sub_4086E2((__int64)&v53, (__int64)&v23) = &loc_4060C3;
310      if ( v50 )
311      {
    00005FA3 check:296 (405FA3)
```

```
x View-1
00000E04FF0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000E05000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000E05010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000E05020  00 00 00 00 00 00 00 00  31 00 00 00 00 00 00 00  ........1.......
00000E05030  79 AB AB AB AB AB AB AB  AB AB 00 42 41 42 41 42  y.........BABAB
00000E05040  41 42 41 42 00 00 00 00  00 00 00 00 00 00 00 00  ABAB............
00000E05050  00 00 00 00 00 00 00 00  21 00 00 00 00 00 00 00  ........!.......
00000E05060  10 54 E0 00 00 00 00 00  9B 00 00 00 00 00 00 00  .T..............
```

指针++后取值，很符合字符串的逐字符处理逻辑
点进去看一下可以发现正是hexdecode过后的输入，而v26与输入产生了联系，所以我们下一步要跟着v26的数据走

```
21      if ( v50 )
22      {
23        v45 = 163LL;
24        if ( *v26 )                                  // judge
25          v45 -= 2LL;
26        if ( !v12 )
27          v12 = v45;
28        if ( !v50 )
29        {
```

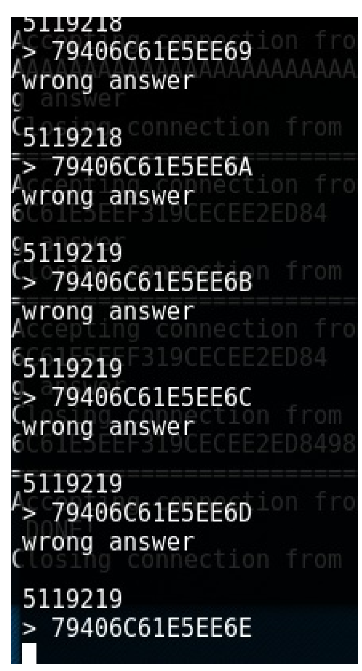
这里判断的v26的值是否为0，等价于`cmp data,input; jz xxx;`
因此可以知道要求第一个值为79，同理继续往下跟即可获得所有flag
另外快速一点的方法是使用断点脚本，在0x405fc6处下断并设置下述脚本，则会在output窗口打印出所需值

```
v = GetRegValue("ecx")
SetRegValue(v,"eax")
print("%X"%v)
```

另外算法实际上也并不复杂，简单来说是通过一个data数组和另一个数组异或产生的数据，前一个数组是逐个赋值的，所以并不好找出顺序来静态解出flag

单字节穷举

由于该程序的算法是逐字节校验，并且当某一个值错误时就会退出，因此可以应用pintools类的侧信道攻击
但并不能直接上轮子，因为在check的前面还有一个hexdecode，使得输入必须两个一组，并且由于数字和字母处理逻辑不同所以也会产生执行次数的跃变，要做特殊修正
首先字典调整成`["%02x"%i for i in range(1,0x100)]`
然后要判断key中存在的字母个数，经测试发现每个字母大概会使运行次数增加1681-1683次，将误差消除后比较即可



大概在一小时左右可以得到flag，效率虽然比较感人，但优势在于期间不用关注该题，只等躺着拿flag就行了233

在之前的轮子上做了微调的脚本如下

```
#-*- coding:utf-8 -*-
import popen2,string

INFILE = "test"
CMD = "/root/pin/pin -t /root/pin/source/tools/ManualExamples/obj-intel64/inscount1.so -- /root/Project/obfuscating_macros.out <" + INFILE
#choices = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!#$%&'()*+,-./:;<=>?@[\]^_`{|}~"#自定义爆破字典顺序，将数字和小写字母提前可以使得速度快一些~
choices = ["%02X"%i for i in range(1,0x100)]

def execlCommand(command):
    global f
    fin,fout = popen2.popen2(command)
    result1 = fin.readline()#获取程序自带打印信息，wrong或者correct
    print result1
    if(result1 != 'wrong answer\n'):#输出Correct时终止循环
        f = 0
    result2 = fin.readline()#等待子进程结束,结果输出完成
    fin.close()



def writefile(data):
    fi = open(INFILE,'w')
```

```
    fi.write(data)
    fi.close()

def pad(data, n, padding):
    return data + padding * (n - len(data))

flag = ''
f = 1
while(f):
    dic = {}
    l = 0#初始化计数器
    for i in choices:
        key = flag + i#测试字符串
        print ">",key
        writefile(pad(key, 8, '0'))
        execlCommand(CMD)
        fi = open('./inscount.out', 'r')
        # 管道写入较慢，读不到内容时继续尝试读
        while(1):
            try:
                n = int(fi.read().split(' ')[1], 10)
                break
            except IndexError:
                continue
        fi.close()
        c = 0
        for ch in key:
            if(ch in string.ascii_uppercase):
                c += 1682
        print n-c
        if(n-c-l>50 and l ):
            flag += i
            break
        else:
            l = n-c
print flag
```

反混淆

毫无疑问的是对于被OLLVM混淆过的程序，纯静态分析难度较大
而本题的混淆方案经过处理，网上现成的轮子只有TSRC于17年发布的文章，但由于块的分布不同所以还需要调整，由于我并不熟悉angr就不班门弄斧了，等一个师傅指教

黑盒破解2-时间谜题

本题目继承去年的黑盒破解，有大量的冗余无关代码，因此逆起来比较吃力，需要耐心和经常整理思路
整体而言题目很新颖，但难度有些高，需要做题人完全理解遗传算法，`大胆`、跳出常规思路去解题

个人认为题目的难点主要是正常做题是倒着来，即思考如何能达到目标–使correct标志位为1，进而追溯哪些数据和条件影响该标志位，然后逆向相关的函数并寻找它们和输入的关系，从而反推出输入。
而本题要求首先逆清整个程序的逻辑和几乎所有功能，然后再思考哪些逻辑不合常理，或者说应该被修改，这是一个很反常的思路。
换言之，这是不同于以往找出正确输入的`CrackMe`，而是一个`修复类型`的题目

main中主要有两个函数比较重要，分别是负责初始化的init和负责调用VM的check

```
printf("--------------------[Welcome To BlackBoxII world!
printf("\n\nPlease Input Your Passcode,If You See print the
if ( !(unsigned int)init() )
{
  printf("Error code:%x\n", (unsigned int)times);
  exit(0);
}
scanf("%s", input, 0x64i64);
printf("%s\n", input);
if ( *(_DWORD *)input != 'tixe' )
{
  check();
  v3 = "\nCongratulation!!! you are success!\n";
  if ( !imp_flag )
```

init里无需多言，主要是各种各样的初始化，里面还有一些混淆，例如随机数、内存交换解密等等，但不受输入影响所以不用关心

check里跟去年一样的方式比对code，然后调用对应的函数
去年的目标是在表中构造出`Binggo!\0`的字符串，然后调用输出函数print出来，最后hash校验成功使得flag出现
而今年这个结构仍然保留了，所以构造`Binggo!\0`的字符串还是能输出`Congratulations!`，而题目里却没有说明这个字符串，对做题人来说题目里出现了`如果构造一个输出满足hash则能给出正确反馈`的状况，所以这是一个很大的坑...
还好我翻出了去年的题目一看hash完全一样，打扰了 乖乖看后面那个函数吧

```
49                                                  // sub_7ff603121ff0
50                                                  // sub_7ff603122210
51                                                  // sub_7ff603121c50
52          *((_BYTE *)char_table + 50) = 0;
53        }
54        ++v6;
55        --v8;
56      }
57      while ( v8 );
58      ++v3;
59      input_len = -1i64;
60      do
61        ++input_len;
62      while ( input[input_len] );
63      v2 = whatmem;
64    }
65    while ( v0 < input_len );
66  }
67  call_func_index_list_data2_cp_ = 0x40;
68  final();
69  return 1i64;
70}
```

是否输出Congratulations的校验标志有两处交叉引用，一处在VM的handler中，另一处则在这个final进去以后的某部分中

事实上这里才是今年题目的开始
关于VM相关的内容可以参考去年的WP中re2黑盒破解的部分，不过去年的时候还不怎么会用结构体功能来恢复对象2333
另外handler有一些升级，需要注意到，主要在于当下标寄存器大于char_table的范围即256时，去年是直接return，防止越界读写
而今年刻意地提供了这个功能

```
1   def f0:
2       if(a>=b):
3           next = t[a]
4           if(a>=b):
5               next = next-127
6               g = t[next]
7           else:
8               m2[a] = input[a-32+t[a]] - 33
9       else:
10          buf = t[a]
11
12  def f1:
13      if(a<b):
14          if(buf):
15              c[a] = buf
16          return
17      if(next-33<=0x27):
18          next = next-33
19          g[a] = next
20          return
21      if(next-84<-0x2a):
22          next = next+117
23          g[a] = next+117
24          return
25      g[a] = buf
```

随便逆了两个func发现这个VM具备任意写的能力，那么这样可操作的范围就大了，甚至大到让人无所适从的地步
毕竟理论上来说任意写甚至可以直接按照pwn的思路来get shell
找了一圈leak，发现堆地址在init函数中提供了

```
             * ((unsigned __int64)((unsigned __int64)v9 * (unsigned __int
  *(_QWORD *)((char *)char_table + 255) = (char *)m3 + (signed int)random_2;
  m3_plus_random = (struc_3 *)((char *)m3 + (signed int)random_2);
```

继续往下看，在final()函数中做了很多事情
首先是随机生成1000个字符串，每个字符串15字节，构成一个对象数组，字符串的字符种类有3种，分别是大写字母、小写字母和数字

```
39        {
40          v9 = rand() % v1->kinds;
41          if ( !v9 )
42            goto LABEL_8;
43          kind = v9 - 1;
```

```
44          if ( kind )
45            break;
46          v13 = rand();                                 // 数字
47          *ch_value = v13
48                    - 10
49                    * (((((unsigned __int64)(0x66666667i64 * v13) >> 32) & 0x80000000) != 0i64
50                     + ((signed int)((unsigned __int64)(0x66666667i64 * v13) >> 32) >> 2))
51                    + v1->basic_num;
52 LABEL_10:
53          ++ch_value;
54          if ( ++v7 >= v1->individual_size )
55            goto LABEL_11;
56        }
57        if ( kind == 1 )
58        {
59          rand_v = rand() % 26;
60          basic_v = v1->basic_uppercase;
61        }
62        else
63        {
64 LABEL_8:
65          rand_v = rand() % 26;
66          basic_v = v1->basic_lowercase;
67        }
68        *ch_value = rand_v + basic_v;
69        goto LABEL_10;
70      }
```

然后循环100次

```
generate_population((signed int *)((char *)m3 + (signed int)random_2));// 初始化种群
v0 = 1;
LOBYTE(v1) = puts("\n");
v2 = random_2;
v3 = 0;
v4 = (char *)m3;
for ( i = (signed int)random_2; v0 < *(_DWORD *)((char *)m3 + (signed int)random_2 + 24); i = (signed int)random_
{
  v1 = -1i64;
  do
    ++v1;
  while ( qword_7FF6B39E7C10[v1] );
  if ( v1 < 0x36 )
    break;
  calc_values_and_sort(&v4[i]);                 // 计算相似度并排序
  m3_random_ptr = (struc_3 *)((char *)m3 + (signed int)random_2);
  v7 = m3_random_ptr->population_size;
  if ( v7 > 0 )
  {
    v8 = 0i64;
    v9 = data3_sum;
    do
    {
      v9 += data3[v8].value;
      ++v8;
      --v7;
    }
    while ( v7 );
    data3_sum = v9;
  }
  rand();
```

```
53    v10 = 0;
54    v11 = m3_random_ptr->population_size;
55    div = (double)(data3_sum / m3_random_ptr->population_size) * *(double *)&m3_random_ptr->weight_cho;// 0.0018
56    data3_sum = (signed int)div;
57    if ( v11 > 0 )
```

```
58    {
59      v13 = 0i64;
60      do
61      {
62        ++v13;
63        ++v10;
64        *((_BYTE *)&data3[v13] - 16) = *((_DWORD *)&data3[v13] - 5) < (signed int)div;// 标志位
65      }
66      while ( v10 < m3_random_ptr->population_size );
67    }                                             // value[i] = str_i - str_input
68                                                  // X=avg(value)*0.0018
69    substr_change((char *)m3 + (signed int)random_2);// 随机选取1000对字符串，如果都小于X，则随机交换他们的1-7位，若value变小则保留，否则不变
70    one_byte_change((char *)m3 + (signed int)random_2);// 对于小于X的每个字符串随机选取一个值，+1写入，取value较小的值写入(先取+1，如果比原来的大则取-1)
71    LOBYTE(v1) = check_finish(v14, (struc_3 *)((char *)m3 + (signed int)random_2));// 检查最优个体是否满足条件
72    if ( (_BYTE)v1 )
73    {
74      v2 = random_2;
75      v4 = (char *)m3;
76      break;
77    }
78    LODWORD(v1) = (unsigned int)((unsigned __int64)(0x66666667i64 * ++v0) >> 32) >> 31;
79    if ( v0 == 10 * (v0 / 10) )
80    {
81      Sleep(0x1Eu);
82      LOBYTE(v1) = printf(".");
83    }
84    v2 = random_2;
```

1. 计算每个字符串和标准字符串的相似度，具体计算方法为逐字符做差取绝对值累加，将相似度保存在对象中，然后对整个数组按照相似度排序
2. 求出相似度的平均值，乘以一个比例0.0018（double类型，根据IEEE标准解析），得到值X
3. 遍历所有字符串，相似度小于X的打上标记
4. 随机选取1000对字符串，如果都具备标记则随机交换后半部分，交换完后计算相似度，如果降低则保留，否则还原
5. 遍历带有标记的字符串，随机选取一个值做+1/-1浮动，选择相似度较小的保留
6. 判断相似度最小的字符串的相似度是否小于某个值0x147，若小于则取相似度排名200-300的带有标记的字符串进行两两异或，从尾部开始逐字符计算，若相等则中断并判断下标是否小于某个值Len=0x4E57795F，若小于则判断Len+4是否大于等于30，是则退出，否则从Len开始重复该串的0-Len位，并在最后附加fill_value="5mE9"。这一段逻辑比较绕，建议自己调试两遍看看memmove的作用

```
121  v14 = v2->final_len;
122  if ( v14 + 4 >= 30 )
123    return 0;
124  memmove(&Dst + v14, &key, v14);
125  len_key = -1i64;
126  v16 = -1i64;
127  do
128    v12 = *((_BYTE *)&key + v16++ + 1) == 0;
129  while ( !v12 );
130  *(_DWORD *)((char *)&key + v16) = v2->fill_value;
```

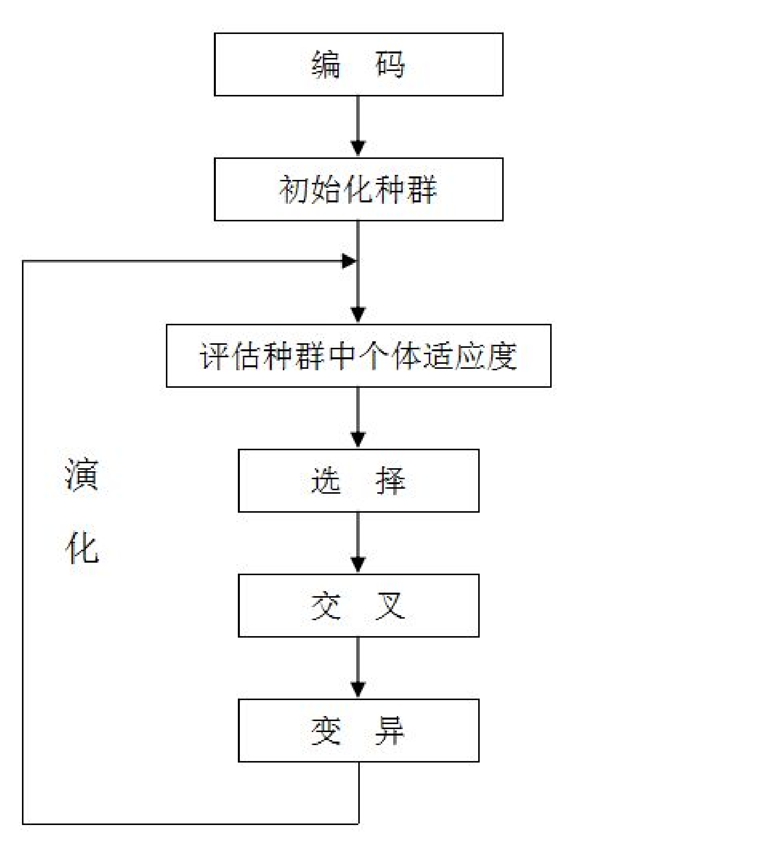
根据逻辑想到遗传算法

> 遗传算法（Genetic Algorithm, GA）是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。

其主要特点是直接对结构对象进行操作，不存在求导和函数连续性的限定；具有内在的隐并行性和更好的全局寻优能力；采用概率化的寻优方法，不需要确定的规则就能自动获取和指导优化的搜索空间，自适应地调整搜索方向。

> 遗传算法以一种群体中的所有个体为对象，并利用随机化技术指导对一个被编码的参数空间进行高效搜索。其中，选择、交叉和变异构成了遗传算法的遗传操作；参数编码、初始群体的设定、适应度函数的设计、遗传操作设计、控制参数设定五个要素组成了遗传算法的核心内容

> 摘自简书超详细的遗传算法(Genetic Algorithm)解析



上述所有参数都是对象的成员

```
00000000 struc_3 struc ; (sizeof=0x68, mappedto_56)
00000000 population_size dd ?
00000004 individual_size dd ?
00000008 kinds dd ?
0000000C basic_lowercase db ?
0000000D basic_num db ?
0000000E basic_uppercase db ?
0000000F basic_4 db ?
00000010 swap_range dd ?
00000014 condition_finish dd ?
00000018 max_loop dd ?
0000001C field_1C dd ?
00000020 weight_cho1 dq ?                    ; 0.003
00000028 weight_cho2 dq ?                    ; 0.007
00000030 weight_cho dq ?                     ; 0.0018
00000038 weight_cho3 dq ?                    ; 0.00215
00000040 final_len dd ?
00000044 fill_value dd ?
00000048 max_len dd ?
0000004C what dd ?
00000050 field_48 dq ?
00000058 ptr_population dq ?
00000060 ptr_table dq ?
00000068 struc_3 ends
00000068
```

由于我们之前对虚拟机的逆向已经知道有任意写的能力，于是这里的所有参数我们都是可以修改的

由于我们之前对虚拟机的逆向已经知道有任意写的能力，于是这里的所有参数我们都是可以修改的
那么问题就在于修改哪些了
默认情况下怎么跑都会由于X=平均值*选优比例0.0018=0而没有字符串被标记，于是后面几个函数就都没有用了
所以首先要修改的就是比例0.0018，虽然对象中有几个备选比例成员，测试发现都还是不行，于是大胆给他上了200（……
这个200也是需要用double类型存储的，用matlab转换

> octave:3> num2hex(200)
> ans = 4069000000000000
> 发现经过100轮遗传以后产生了很多字符串，但相似度仍然很低，于是再把max_loop遗传代数调大到1000，发现出现了很多跟标准字符串相似的串
>
> ```
> [1]:   0    rXWZ2OGQwo8Em9r
> [2]:   1    rXWZ2OGQwo8Em9q
> [3]:   1    rXWZ2OGQwo8Em9q
> [4]:   1    rXWZ2OGQwo8Em9q
> [5]:   1    rXWZ2OGQwo8Em9q
> [6]:   1    rXWZ2OGQwo8Em9q
> [7]:   1    rXWZ2OGQwo8Em9q
> [8]:   1    rXWZ2OGQwo8Em9q
> [9]:   2    rXWZ2OGQwo8Em9p
> [10]:  2    rXWZ2OGQwo8Em7r
> [11]:  3    rXWZ2OGQwo8Em9u
> [12]:  3    rXWZ2OGQwo8Em9u
> [13]:  4    rXWZ2OGQwn8Em9n
> [14]:  4    rXWZ2OGQwo8Em8u
> [15]:  4    rXWZ2OGQwo8Em7p
> [16]:  4    rXWZ2OGQwo8Em9v
> [17]:  4    rXWZ2OGQwo8Em9v
> [18]:  4    rXWZ2OGQwo8Em5r
> ```

注意特征是不同的字符全部位于尾端，猜测是因为交换的算法是两串交换尾端部分
再观察排名200-300的串

```
[199]:  28   rXWZ2OGQwo8EmDa
[200]:  28   rXWZ2OGQwo8Em9V
[201]:  28   rXWZ2OGQwo8EmTq
[202]:  28   rXWZ2OGQwo8EmJg
[203]:  28   rXWZ2OGQwo8EmDa
[204]:  28   rXWZ2OGQwo8EmQn
[205]:  28   rXWZ2OGQwo8EmRu
[206]:  29   rXWZ2OGQwo8EmPl
[207]:  29   rXWZ2OGQwo8Em9U
[208]:  29   rXWZ2OGQwo8Em5Y
[209]:  29   rXWZ2OGQwo8EmUq
[210]:  29   rXWZ2OGQwo8EmGc
[211]:  29   rXWZ2OGQwo8Em5Y
[212]:  30   rXWZ2OGQwo8EmNi
[213]:  30   rXWZ2OGQwo8EmRw
[214]:  30   rXWZ2OGQwo8EmPk
[215]:  30   rXWZ2OGQwo8EmGb
[216]:  30   rXWZ2OGQwo8EmPk
[217]:  31   rXWZ2OGQwn8EmVs
```

确认猜想，就是尾部若干字符不同
那么flag为啥还没出现呢？
因为数据成员中还有一个Len，默认值高达0x4E57795F，会导致任何串都无法进入
而怎么调整也并没有任何线索，它从0-23全部可取，每一个值都会导致解锁密钥不同，且进入解密时会破坏原密文
暴力试了所有值都无果，最终想到了对标准串暴力枚举所有Len和相同的值下标
结果正确的标准是对数据RC4解密后hash的值为36468080

```
class RC4:
```

```
    def __init__(self, public_key=None):
        if not public_key:
            public_key = 'none_public_key'
        self.public_key = public_key
        self.index_i = 0;
        self.index_j = 0;
        self._init_box()

    def _init_box(self):
        """
        初始化 置换盒
        """

        self.Box = [i for i in range(256)]
        key_length = len(self.public_key)
        j = 0
        for i in range(256):
            index = ord(self.public_key[(i % key_length)])
            j = (j + self.Box[i] + index) % 256
            self.Box[i], self.Box[j] = self.Box[j], self.Box[i]

    def do_crypt(self, string):
        """
        加密/解密
        string : 待加/解密的字符串
        """

        out = []
        for s in string:
            self.index_i = (self.index_i + 1) % 256
            self.index_j = (self.index_j + self.Box[self.index_i]) % 256
            self.Box[self.index_i], self.Box[self.index_j] = self.Box[self.index_j], self.Box[self.index_i]
            r = (self.Box[self.index_i] + self.Box[self.index_j]*2 + self.index_i + self.index_j) % 256

            # r = (self.Box[self.index_i] + self.Box[self.index_j]) % 256
            R = self.Box[r]  # 生成伪随机数
            out.append(((s) ^ R))

        return (out)
# c = [0x7f, 0x2e, 0x79, 0x56, 0x6, 0xc2, 0xb8, 0x47, 0x52, 0xe1, 0xb9, 0x7f, 0x38, 0x1b, 0xa, 0xcc, 0x18,
0x7a, 0xec, 0xf8, 0xa2, 0x89, 0x91, 0x78, 0xa6, 0x4b, 0x1b, 0x85, 0x93, 0x9a, 0x4c, 0x59, 0x6e, 0xf5, 0xf4,
 0x7c, 0xd2, 0xf4, 0x2, 0x6, 0xe4, 0xfb, 0xcb, 0xd7, 0x7c, 0xa9, 0x85, 0xe5, 0x0, 0x15, 0x90, 0x6, 0x4f, 0x
1f, 0x52, 0x54, 0xf, 0x5a, 0x3d, 0x87, 0x32, 0x5b, 0xd6, 0xb2]
c = [0xbe, 0x70, 0x48, 0xc6, 0xa1, 0x60, 0x68, 0xcf, 0xd2, 0x6e, 0x9e, 0x60, 0x81, 0x1a, 0x5d, 0x4a, 0x71,
0x9b, 0xea, 0x51, 0x2c, 0xb7, 0x46, 0xa1, 0x15, 0xb9, 0xee, 0xc6, 0xc8, 0x0, 0x8f, 0x10, 0xd, 0xc0, 0xe2, 0
x79, 0xa5, 0x88, 0xcc, 0x6f, 0xc2, 0x1d, 0xd7, 0x8e, 0x2, 0xf6, 0x1b, 0x7a, 0x5f, 0x5b, 0x6f, 0xe3, 0x59, 0
xe, 0x3f, 0x91, 0x16, 0x3c, 0x32, 0x95, 0x29, 0xec, 0xcb, 0xec]

def hash1(x):
    r = 0
    for i in x:
        if(i>0x7f):
            i = i+0xffffff00
        r = r*16 + i
        v46 = r&0xf0000000
        if(v46!=0):
            r = r^(v46>>24)
        r = (~v46) & r & 0xffffffff
    return r
```

```
true_h1 = 36468080

ori = "rXWZ2OGQwo8Em9y"

for final_len in range(24):
    for i in range(len(ori)):
        if(i>final_len):
            continue
        ss = ["" for k in range(48)]
        tmp = ["" for k in range(24)]
        s = ""
        for k in range(len(ori)):
            ss[k] = ori[k]
        ss[i]=""
        for k in range(final_len):
            tmp[k] = ss[k]
        for k in range(final_len):
            ss[final_len+k-1] = tmp[k]
        for k in range(len(ss)):
            if(ss[k]==""):
                ss[k] = "8"
                ss[k+1] = "E"
                ss[k+2] = "m"
                ss[k+3] = "9"
                break
        print(ss)
        for k in ss:
            if(k==""):
                break
            s += k
        print(final_len, i, s)


        rc4 = RC4(s)
        p = rc4.do_crypt(c)
        print("".join(map(chr,p)))
        h1 = hash1(p)
        print((i,s,h1))
        if(h1==36468080):
            print("!"*150)
```

打印出的值检索"!!!"可以发现flag

```
Flag:DDCTF{019_4x5a9FJEIk}
(12, 'rXWZ20GQwo8ErXWZ20GQwo8E8Em9', 36468080)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
['r', 'X', 'W', 'Z', '2', '0', 'G', 'Q', 'w', 'o', '8', 'E', 'r', 'X', 'W', 'Z'
13 13 rXWZ20GQwo8ErXWZ20GQwo8Em8Em9
»ß      ?}EÛ® Aä
(13, 'rXWZ20GQwo8ErXWZ20GQwo8Em8Em9', 52821380)
```

事后跟出题老师交流了一下发现问题出在min_value上，期望解法应该是将其改为1，这样当最优解即跟标准串完全一致的串出现时，排名200-300的串才具备N个字符不同的特征

默认的0x147出现时，排名200-300的串可能有N+1，N+2个字符不同，而RC4解密是一次性的，如果第一个进行解密的串不符标准就会破坏密文导致后续正确key即使出现也无法解密出flag

赛时写了一个方便Patch的IDA脚本，在generation之前执行即可Patch各个参数

```
from ida_bytes import patch_bytes
```

```
import struct
base = idaapi.get_imagebase()
GA = Qword(base+0x286a8)+Qword(base+0x286b0)
print(hex(GA))
min_value = GA+0x14
loop_size = GA+0x18
weight = GA+0x30
ptr_str = GA+0x60
final_len = GA+0x40

w = "3ff0000000000000"
#w = "3FF007D4E4D205F3"

w = w.decode("hex")[::-1]
patch_bytes(weight, w)
patch_bytes(final_len, struct.pack("<L",0xd))
patch_bytes(loop_size,struct.pack("<L",20000))
patch_bytes(min_value,struct.pack("<L",1))
#patch_bytes(ptr_str,struct.pack("<Q",guess_str))
print("Patch finished")
```

查看当前种群

```
base = idaapi.get_imagebase()
data3 = Qword(base+0x29d90)
GA = Qword(base+0x286a8)+Qword(base+0x286b0)
#min_value = Dword(GA+0x14)
min_value = Dword(base+0x29d80)
for i in range(1000):
  v = Dword(data3+4+24*i)
  if(v>=min_value):
    break
  if(i<300):
    s = ""
    ptr_s = Qword((data3+16+24*i))
    for j in range(16):
      s += chr(Byte(ptr_s+j))
    print("[%d]:\t%d\t%s"%(i+1,v,s))
print("alive:%d, v=%d"%(i+1,v))
```