PlaidCTF 2022 coregasm writeup

原创

1mmorta1 于 2022-04-11 15:25:40 发布

分类专栏： reverse 文章标签： linux 安全

本文链接：https://blog.csdn.net/qq_41866334/article/details/124100130

题目给了elf文件coregasm 和它crash以后的coredump文件，我们需要利用coredump文件里的信息来逆向elf文件。这一题分成四个部分：

```
puts("Would you like to see a magic trick?");
puts("Printing all the flags...");
fflush(0LL);
v3 = open("/dev/urandom", 0);
if ( read(v3, globalbuf, 0x40uLL) != 64 )
{
  v4 = 0xBB;
  v5 = "x == 64";
  goto LABEL_5;
}
close(v3);
flag4(globalbuf);
flag3(globalbuf);
flag2(globalbuf);
flag1(globalbuf);
puts("///time for core///");
fflush(0LL);
if ( strcmp("///time for core///", *argv) )
{
  v4 = '\xC5';
  v5 = "strcmp(\"///time for core///\", argv[0]) == 0";
LABEL_5:
  __assert_fail(v5, "./main.c", v4, "main");
}
```


flag1

这是最简单的，因为crash后的coredump文件里一定是有异或了0xa5的flag1. 所以我们直接对于整个coredump文件异或0xa5，就找到了flag1 。

flag2

```
  v1 = fopen("./otp", "r");
  if ( fread(v4, 0x80uLL, 1uLL, v1) != 1 )
    __assert_fail("items == 1", "./main.c", 0x2Au, "flag2");
  for ( i = 0LL; i != 64; ++i )
    s[i] ^= v4[i];
  *(_QWORD *)s ^= 0x6301641F2866C34BuLL;
  *((_QWORD *)s + 1) ^= 0x1EB4DEF5AC740DCFuLL;
  *((_QWORD *)s + 2) ^= 0x4F490B1C93DF4671uLL;
  *((_QWORD *)s + 3) ^= 0x9F82C6EC691CA0B0LL;
  *((_QWORD *)s + 4) ^= 0xC2D142FCAF5DCA6BLL;
  *((_QWORD *)s + 5) ^= 0xFA68305EB42FCB00LL;
  *((_QWORD *)s + 6) ^= 0x62212646A9E04B61uLL;
  *((_QWORD *)s + 7) ^= 0xBB73AD9A9992C6BuLL;
  puts("Flag 2:");
  puts(s);
  fflush(0LL);
  for ( result = 0LL; result != 64; ++result )
    s[result] ^= v4[result + 64];
  return result;
}
```


找到我们发现这部分的加密用到了从./otp里读取的数据，这部分数据也应该残留在了coredump文件里。 用010Editer打开人工翻翻找到了0x80长度的数据如下：

```
h: 1B 80 32 DA 78 8C 0D F2 65 C6 A0 32 97 BF DA 7F  .€2ÚxŒ.òeÆ 2—¿Ú.
h: 1F 27 FB F1 7D 65 28 DE D1 81 7E 08 82 A7 EC 01  .'ûñ}e(ÞÑ.~.,§ì.
h: 0A 40 10 F5 38 17 63 67 EA 4E BA 20 7F 10 48 DA  .@.õ8.cgêNº  ..HÚ
h: 40 6B C0 89 6E 86 24 72 4B 0C B9 89 81 4C A3 39  @kÀ‰n†$rK.¹‰.L£9
h: 3B 31 94 E7 D7 3E 08 80 4F 73 60 CA BB 6E 13 75  ;1"ç×>.€Os`Ê»n.u
h: 80 83 14 CD 45 E9 07 22 FE 4A 25 80 F6 5B D7 80  €ƒ.ÍEé."þJ%€ö[×€
h: 58 31 40 32 91 81 0B DC C0 4D 7D 5D 46 64 44 AF  X1@2'..ÜÀM}]FdD¯
h: 32 66 95 51 BD 54 AD 9E 74 F9 12 13 98 2C 4D 0C  2f•Q½T-žtù..˜,M.
```

```
import base64
import string
flag1 = b'PCTF{banana_banana}\x00                                                '

# flag1 = [int.from_bytes(flag1[i:i+8],'little') for i in range(0,64,8)]

xrn = [0x80083ED7E794313B,0x75136EBBBF60734F,0x6C46A704AF4D8380,0xC1991AB8C1674BBF,0xDC0B819132401105,0xAF446446
5D7D4DC0,0x9EAD54BD51956632,0xC4D2C981312F974]
xrn = b''.join([i.to_bytes(8,'little') for i in xrn])
def xor(a,b):
    assert(len(a)==len(b))
    return [a[i]^b[i] for i in range(len(b))]
tmp = xor(xrn,flag1)
# print(tmp)
with open("core","rb") as f:
    con = f.read()


num = 'G4Ay2niMDfJlxqAyl7/afx8n+/F9ZSje0YF+CIKn7AEKQBD1OBdjZ+pOuiB/EEjaQGvAiW6GJHJLDLmJgUyjOTsxlOfXPgiAT3NgyrtuE
3WAgxTNRekHIv5KJYD2W9eAWDFAMpGBC9zATX1dRmRErzJmlVG9VK2edPkSE5gsTQw='
num = base64.b64decode(num)

print(bytes(xor(tmp,num[64:128]))) # b'PCTF{banana*banana$banana!banana}\x00                                '
```

flag3

```
  *(_QWORD *)s ^= 0x2F01D6F7C8701DA9uLL;
  *((_QWORD *)s + 1) ^= 0x230ED5E2EC453098uLL;
  *((_QWORD *)s + 2) ^= 0x2F01DAE2EF4A3F97uLL;
  *((_QWORD *)s + 3) ^= 0x2301DAE2EC45309BuLL;
  *((_QWORD *)s + 4) ^= 0x230ED5E2EC4A3F97uLL;
  *((_QWORD *)s + 5) ^= 0x2002D5E2EC4A3F97uLL;
  *((_QWORD *)s + 6) ^= 0x200ED5E2EF4A3F97uLL;
  *((_QWORD *)s + 7) ^= 0x6140948CF3453C97uLL;
  puts("Flag 3:");
  puts((const char *)s);
  fflush(0LL);
  v2 = *s;
  s[15] = 0x12345678;
  v3 = s[1] + v2;
  v4 = s[4] - s[3];
  v5 = s[7];
  v6 = s[10] / s[9];
  v7 = s[13];
  s[2] = v3;
  v8 = s[6] * v5;
  v9 = s[12] ^ v7;
  s[5] = v4;
  s[8] = v8;
  s[14] = v9;
  v10 = v6;
  s[11] = v6;
  *s = v4 & v3;
  s[1] = v8 | v4;
  s[4] = v9 * v6;
  s[3] = v8 % v6;
  s[6] = v9 / v3;
  s[7] = v4 + v9;
  s[9] = v9 - v8;
  s[10] = v6 ^ v3;
  result = v6 / v4;
  s[13] = v8 & v3;
  s[12] = v10 % v4;
  return result;
}
```


flag3 一开始一直卡住了，因为发现后面这些赋值的信息是不够的， 它只能推出：s[1]+s[0], s[6]*s[7] 和s[12]==s[13] 这些信息，而且coredump文件里也没有额外的信息，这明显是推不出flag的。因此怀疑前面还有信息。 去找了找flag4的加密函数（截图在下文中），发现flag4最后赋值时每个u_int64都是一样的。

因此我们可以通过s[1]+s[0]的值以及，s[0]='PCTF' 推出这个一样的值，最后正着推出整个flag3.

PCTF{bananabnanbnanannanbnabnnabnanbnanbnanbnabanananananba}

flag4

```
  *(_QWORD *)s ^= 0xBC019EE23A6BF6BFLL;
  *((_QWORD *)s + 1) ^= 0xE9483020414B589CLL;
  *((_QWORD *)s + 2) ^= 0x217B7D11E6C9A8A3uLL;
  *((_QWORD *)s + 3) ^= 0x3B3924CE775A8541uLL;
  *((_QWORD *)s + 4) ^= 0x6BBDB2171BAD0EC8uLL;
  *((_QWORD *)s + 5) ^= 0xB0B0429F1F0242E9LL;
  *((_QWORD *)s + 6) ^= 0x5DE514AB5ABE8132uLL;
  *((_QWORD *)s + 7) ^= 0x50789E90A63C152EuLL;
  puts("Flag 4:");
  puts(s);
  fflush(0LL);
  v2 = COERCE_DOUBLE(((unsigned __int64)*((unsigned __int16 *)s + 2) << 3
  v3 = COERCE_DOUBLE(((unsigned __int64)*((unsigned __int16 *)s + 5) << 3
     + v2;
  v4 = COERCE_DOUBLE(((unsigned __int64)*((unsigned __int16 *)s + 8) << 3
     + v3;
  v5 = COERCE_DOUBLE(((unsigned __int64)*((unsigned __int16 *)s + 11) <<
     + v4;
  v6 = COERCE_DOUBLE(((unsigned __int64)*((unsigned __int16 *)s + 14) <<
     + v5;
  v7 = COERCE_DOUBLE(((unsigned __int64)*((unsigned __int16 *)s + 17) <<
     + v6;
  *(_QWORD *)&v10 = ((unsigned __int64)*((unsigned __int16 *)s + 20) << 3
  result = 0LL;
  v9 = v2
     * (v3
      * (v4
       * (v5
        * (v6
         * (v7
          * ((v10 + v7)
           * (COERCE_DOUBLE(((unsigned __int64)*((unsigned __int16 *)s +
            + v10
            + v7)))))));
  do
    *(double *)&s[8 * result++] = v9;
  while ( result != 8 );
  return result;
}
```


这个逆向是关于intel fst浮点数计算。 这个伪代码翻译的挺差的，其实看汇编指令会更加清楚一点。 事实上程序把flag每六个字节取出来放在double的后六个字节，前面补上0x3fff，然后将double转成long double
形式做连加和连乘。 由于整个程序在crash之前只有这里用到了st寄存器，因此我们可以从coredump里获得8个st寄存器的值。 具体做法是命令行运行`gdb coregasm coredump` 然后`info all-register`

```
st0            15.6579354707501636756 (raw 0x4002fa86e7581c014d00)
st1            214.831820219884093451 (raw 0x4006d6d4f22b808dbff5)
st2            2526.22752352315034186 (raw 0x400a9de3a3efb4b005ce)
st3            24751.5189151917131252 (raw 0x400dc15f09af40839c63)
st4            193984.0536771345168 46 (raw 0x4010bd70036f723852be)
st5            1139716.96293101242827 (raw 0x40138b2027b4152cb578)
st6            4457828.61824004405162 (raw 0x4015880ac93c89f5848f)
st7            8758372.44193981720582 (raw 0x401685a4647122f7c5b3)
fctrl          0x37f               0x37f
```

接下来我们调试一下这个elf文件coregasm，发现`fmulp st(1), st`指令中的pop 并不是直接把st(0) 删除，而是所有其他寄存器都往上pop一步，同时原来的st(0)赋值给了st(7) 。 因此留下来的8个寄存器的值分别代表什么就很清楚了. st(0)是八个数字的和，而st(1)-st(7) 分别是乘积。

因此写出solve代码，利用union可以对一块内存做出两种解释。

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

union LDC
{
   long double f;
   unsigned char s[10];
};

union DC
{
   double f;
   unsigned char s[8];
};

int main(){
    union LDC st[8];
    int i, j;
    union DC tmp;

    memcpy(st[0].s,"\x00M\x01\x1cX\xe7\x86\xfa\x02@",10);
    memcpy(st[1].s,"\xf5\xbf\x8d\x80+\xf2\xd4\xd6\x06@",10);
    memcpy(st[2].s,"\xce\x05\xb0\xb4\xef\xa3\xe3\x9d\n@",10);
    memcpy(st[3].s,"c\x9c\x83@\xaf\t_\xc1\r@",10);
    memcpy(st[4].s,"\xbeR8ro\x03p\xbd\x10@",10);
    memcpy(st[5].s,"x\xb5,\x15\xb4' \x8b\x13@",10);
    memcpy(st[6].s,"\x8f\x84\xf5\x89<\xc9\n\x88\x15@",10);
    memcpy(st[7].s,"\xb3\xc5\xf7\"qd\xa4\x85\x16@",10);

    for(i = 7; i >= 1; i--)
    {
        st[i].f /= st[i-1].f;
    }

    for(i = 0; i < 7; i++)
    {
        st[i].f -= st[i+1].f;
    }

    for(i = 7; i >= 0; i--)
    {
        tmp.f = st[i].f;
        for (j = 0; j <= 5; j++)
            printf("%c", tmp.s[j]);
    }
 printf("\n"); //PCTF{orange%zou&gl`d$i!dldn't^pay#bapana*aeain}
    return 0;
}
```