2021虎符CTF逆向WP

原创

Whitebird_ 于 2021-04-06 20:28:25 发布 467 收藏 1

分类专栏： 逆向 文章标签： 安全

本文链接：https://blog.csdn.net/jxnu_666/article/details/115448655

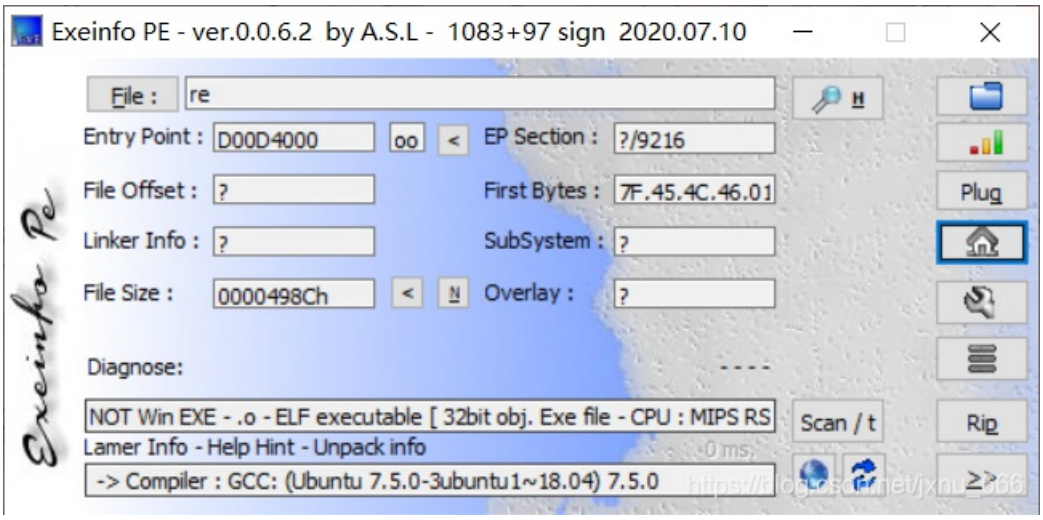
逆向 专栏收录该内容

1 篇文章 0 订阅

订阅专栏

周六打了2021虎符的比赛，由于刚入门逆向，也就写出来了一题，赛后复现了一波，可能会有写错的地方，欢迎大佬指出。

一、redemption_code



首先，拿到这道题，我们查一下壳，并没有加壳。然后拉入ida32查看一下

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v3; // $a2
  int v4; // $v0
  int v5; // $v0
  int v6; // $a2
  int v7; // $v0
  char v9[4]; // [sp+1Ch] [+1Ch] BYREF
  int v10; // [sp+20h] [+20h]
  char v11[24]; // [sp+24h] [+24h] BYREF
  char v12[24]; // [sp+3Ch] [+3Ch] BYREF
  char v13[24]; // [sp+54h] [+54h] BYREF
  char v14[24]; // [sp+6Ch] [+6Ch] BYREF

  std::operator<<<std::char_traits<char>>(&std::cout, "your redemption code: ", envp);
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(v11);
  std::getline<char,std::char_traits<char>,std::allocator<char>>(&std::cin, v11);// 输出flag
  pre(v11);                             // 加密函数
  std::allocator<char>::allocator(v9);
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(
    v12,
    "I Love Ninja Must Die 3. Beautiful Art And Motive Operation Is Creative.",
    v9);
  std::allocator<char>::~allocator(v9);
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(v13, v12);
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(v14, v11);
  v10 = server_check_redemption_code(v13, v14); // 校验flag
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(v14);
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(v13);
  if ( v10 == 7 )                       // 校验完返回7
  {
    v4 = std::operator<<<std::char_traits<char>>(&std::cout, "award is: flag[", v3);
    v5 = std::operator<<<char>(v4, v11);        // 输出flag
    v7 = std::operator<<<std::char_traits<char>>(v5, "]", v6);
  }
```

看了下大概的流程，写了点注释，接下来我们的重点就是pre(v11);和server_check_redemption_code 两个函数

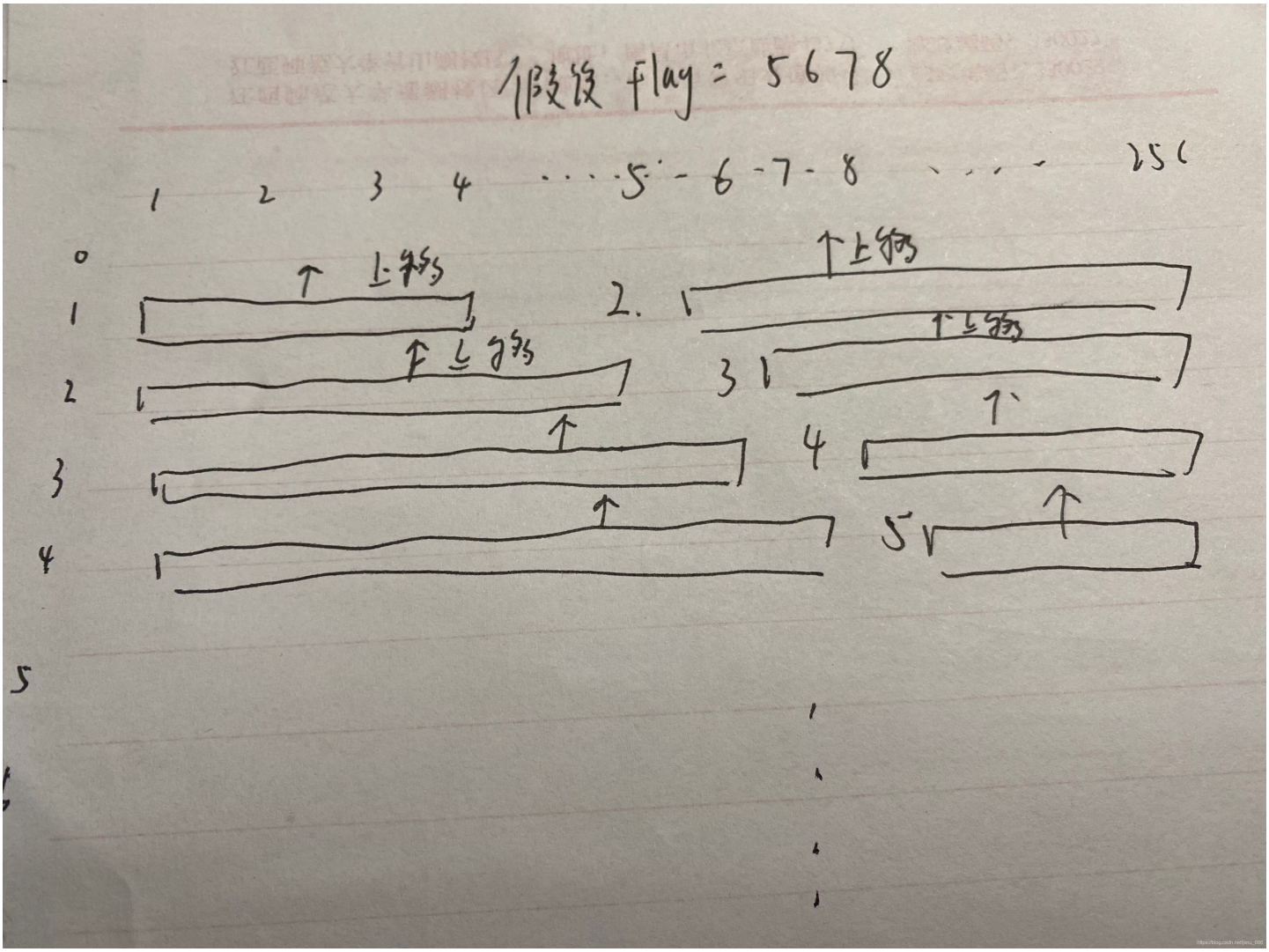
```
int __fastcall pre(int a1)
{
  int v1; // $a2
  int v2; // $v0
  _BOOL4 v3; // $s0
  int v4; // $a2
  int v5; // $v0
  char v8[4]; // [sp+20h] [+20h] BYREF
  char v9[24]; // [sp+24h] [+24h] BYREF
  char v10[24]; // [sp+3Ch] [+3Ch] BYREF

  if ( std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::length(a1) != 14 )// flag长度14
  {
    v2 = std::operator<<<std::char_traits<char>>(&std::cout, "redemption code format error", v1);
    std::ostream::operator<<(v2, &std::endl<char,std::char_traits<char>>);
    exit(-1);
  }
  std::allocator<char>::allocator(v8);
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(
    v9,
    "Ninja Must Die 3 Is A Cruel Game, So Hard For Me",
    v8);
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(v10, a1);
  v3 = server_check_redemption_code(v9, v10) == -1;// v9是上面的字符串，v10是flag
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(v10);
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(v9);
  std::allocator<char>::~allocator(v8);
  if ( v3 )
  {
    v5 = std::operator<<<std::char_traits<char>>(&std::cout, "game error", v4);
    std::ostream::operator<<(v5, &std::endl<char,std::char_traits<char>>);
    exit(-2);
  }
```

对pre函数写了点注释，我们看到pre的函数和main函数都有server_check_redemption_code，我们的分析重点就是后面一个函数

```
if ( std::operator==<char>(a2, &unk_401D94) )
  return 0;
v9 = std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::length(a1);// 字符串的长度
v3 = std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::length(a2);// flag的长度
v10 = v3;
if ( v3 >= 0x200000 )
  _cxa_throw_bad_array_new_length();
s = (void *)operator new[](v3 << 10);          // a2就是flag，下面我们用flag表示更直观
memset(s, 0, v10 << 10);
*((_DWORD *)s
+ *(char *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](a2, 0)) = 1;// s[flag[0]]=1
v4 = 0;
for ( i = 1; i < v10; ++i )                    // v10=flag的长度14
{
  for ( j = 0; j < 256; ++j )                  // 一个行flag长度，列为256的二维数组
  {
    if ( j != *(char *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](a2, i) )// 查找flag[i]
      *((_DWORD *)s + 256 * i + j) = *((_DWORD *)s + 256 * v4 + j);// s[i][j]=s[v4][j] v4=0
    else
      *((_DWORD *)s + 256 * i + j) = i + 1;    // 找到了就 s[i][j]=i+1
  }
  v4 = *((_DWORD *)s
       + 256 * v4                              // v4=s[v4][flag[i]] 相当于转到下一行去了
       + *(char *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](a2, i));
}
v7 = 0;
for ( k = 0; k < v9; ++k )                     // v9=字符串长度
{
  v7 = *((_DWORD *)s                           // v7=s[v7][字符串[k]] v7=0
       + 256 * v7
       + *(char *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](a1, k));
  if ( v7 == v10 )                             // 找到这个字符串和flag第一个头相同的地方
    return k - v10 + 1;                        // 返回偏移值
}
return -1;
```


中间的算法，我拿纸画了下



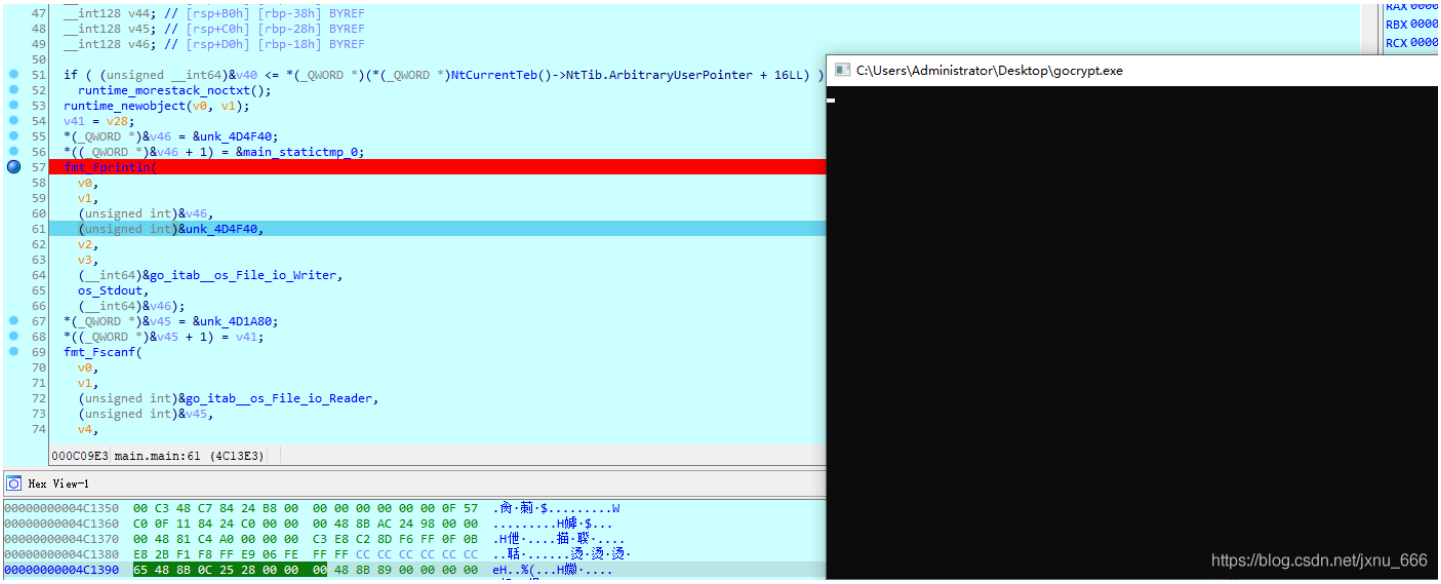
实际上就是在第i行(从0开始)第x填上一个i+1，x是输入的ascii码
然后后面的比较就是必须一整串对比下来，其实就是个字符串对比的过程，最后返回的是偏移

第一个pre函数中的server_check_redemption_code返回了偏移地址-1，不符合。第二个server_check_redemption_code返回偏移地址7，我们从第7个字符开始数14个就是flag

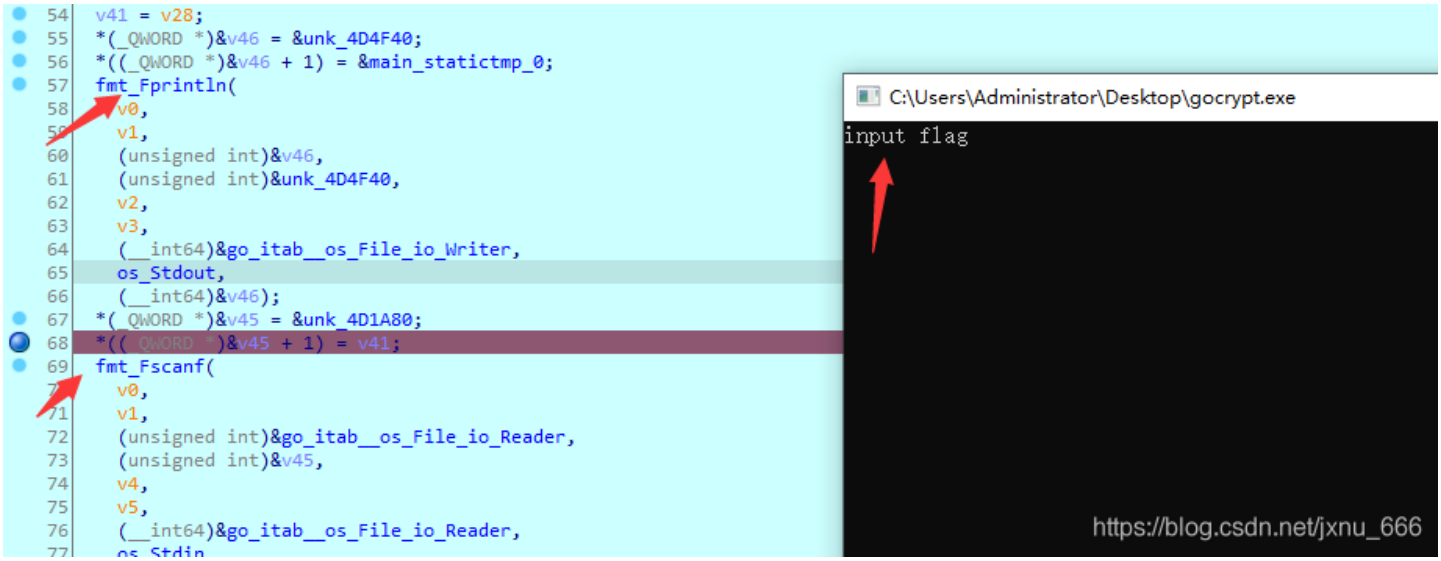
```
'I Love Ninja Must Die 3. Beautiful Art And Motive Operation Is Creative.",
```

二、GoEncrypt

首先，搜索函数main,找到main_main开始分析



在这下个断点，然后调试下，发现fmt_Fprintln是输出“input flag”




```
  2LL,
  (__int64)&v45,
  1LL);
*(_QWORD *)&v29 = v41[1];
main_check(v0, v1, v6, *v41, v7, v8, *v41, v29);// 是一个check flag
if ( v32 )
{
```

```
v8 = *(_QWORD *)NtCurrentTeb()->NtTib.ArbitraryUserPointer;
if ( (unsigned __int64)&v33 <= *(_QWORD *)(v8 + 16) )
  runtime_morestack_noctxt();
result = (_QWORD *)a8;
if ( (_QWORD)a8 == 42LL )
{
  regexp_MustCompile(a1, a2, a3, v8, a5, a6, (__int64)&unk_5041E5, 78LL);
  regexp___Regexp__FindStringSubmatch(a1, a2, v10, v11, v12, v13, v21, a7, 42LL);
  result = v22;
```

这个比较可疑，点进去看看

```
aFlag09aF809aF4 db '^flag{([0-9a-f]{8})-([0-9a-f]{4})-([0-9a-f]{4})-([0-9a-f]{4})-([0'
                                        ; DATA XREF: main_check+48↑o
                db '-9a-f]{12})}$QueryPerformanceFrequency syscall returned zero, run'
                db 'ning on unsupported hardwarereflect.Value.Interface: cannot retur'
                db 'n value obtained from unexported field or methodQueryPerformanceF'
                db 'requency overflow 32 bit divider, check nosplit discussion to pro'
                db 'ceedNoMatchEmptyMatchLiteralCharClassAnyCharNotNLAnyCharBeginLine'
                db 'EndLineBeginTextEndTextWordBoundaryNoWordBoundaryCaptureStarPlusQ'
```

```
db 'uestRepeatConcatAlternate000102030405060708091011121314151617181 9'
db '2021222324252627282930313233343536373839404142434445464748495 0515'
db '2535455565758596061626364656667686970717273747576777879808182838 4'
db '8586878889909192939495969798999',0
db    0
db    0
```

这个是给了flag的格式

flag{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}

每一位的范围0-9和a-f

```
*(_QWORD *)&v29 = v41[1];
main_check(v0, v1, v6, *v41, v7, v8, *v41, v29);// 是一个check flag
if ( v32 )
{
  v36 = v34;
  v35 = v33;
  v38 = v32;
  runtime_newobject(v0, v1);
  *v30 = main_statictmp_4;
  v37 = main_statictmp_5;
  main_NewCipher(v0, v1, v11, v12, v13, v14, (__int64)v30, 16LL);
  v39 = v33;
  runtime_newobject(v0, v1);
  v40 = v31;
  v15 = v33;
  main___myCipher__Encrypt(v0, v34, v38, v39, v16, v17, v39, v31, 16LL, 16LL, v38, v33);
  if ( v15 < 8 )
    runtime_panicslice();
  v20 = v38;
  main___myCipher__Encrypt(
    v0,
    v38,
    v38 + (((8 - v36) >> 63) & 8),
    v40,
    v18,
    v19,
    v39,
    v40 + 8,
    8LL,
    8LL,
    v38 + (((8 - v36) >> 63) & 8),
    v35 - 8);
  internal_bytealg_Equal(v0, v38, v21, v22, v23, v24, v40, 16, 16LL, (__int64)&v37, 16, 16);
  if ( (_BYTE)v36 == 8 )
  {
    *(_QWORD *)&v43 = &unk_4D4F40;
    *((_QWORD *)&v43 + 1) = &main_statictmp_2;
```

这个是对flag加密

是一个xtea加密，可以参考tea加密

```
/*
xtea加密函数
num_rounds      加密轮数
uint32_t* origin   为要加密的数据是两个32位无符号整数
uint32_t* k        为加密解密密钥，为4个32位无符号整数，即密钥长度为128位
*/
void xtea_encode(unsigned int num_rounds, uint32_t origin[2], uint32_t const key[4]) {
    unsigned int i;
    uint32_t v0 = origin[0], v1 = origin[1], sum = 0, delta = 0x9E3779B9;
    for (i = 0; i < num_rounds; i++) {
        v0 += (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
        sum += delta;
        v1 += (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum >> 11) & 3]);
    }
    origin[0] = v0; origin[1] = v1;
}
/*
xtea解密函数
num_rounds      加密轮数
```

```
regexp__inputBytes_hasPrefix
regexp__inputBytes_index
regexp__inputBytes_context
regexp__inputReader_step
regexp__inputReader_canCheckPrefix
regexp__inputReader_hasPrefix
regexp__inputReader_index
regexp__inputReader_context
regexp_init_0
regexp__Regexp_FindStringSubmatch
regexp_mergeRuneSets_func1
regexp_makeOnePass_func1
regexp_init
regexp__onePassInst_String
type_hash_regexp_entry
type_eq_regexp_entry
type_hash_regexp_inputReader
```

```
v16 = 0LL;
LODWORD(v17) = 0;
while ( v16 < 32 )
{
  v13 = v14;
  v18 = v14 + ((v14 >> 5) ^ (16 * v14));
  v19 = *(_QWORD *)(a7 + 32);
  v20 = *(_QWORD *)(a7 + 24);
  v21 = v17;
  v22 = v17 & 3;
  if ( v22 >= v19 )
    runtime_panicindex(v19, v22, v15, a7);
  v23 = v15 + (v18 ^ (v21 + *(_DWORD *)(v20 + 4 * v22)));
  v17 = (unsigned int)(v21 + 0x12345678);
  v24 = ((unsigned int)v17 >> 11) & 3;
  if ( v24 >= v19 )
    runtime_panicindex(v19, v17, v24, a7);
  ++v16;
  a1 = (v23 + ((v23 >> 5) ^ (16 * v23))) ^ (v21 + *(_DWORD *)(v20 + 4 * v
  v14 = v13 + a1;
  v15 = v23;
}
((void (__fastcall *)(__int64, __int64, __int64, __int64, __int64))main_e
if ( (unsigned __int64)a9 < 4 )
```

相当于<<4

还是有很多相似的地方，但是delta被改了

然后现在我们还需要key和加密后的数据才能解密，分析了加密过程我们知道这个key就是v21，我们随便输入flag{12345678-1234-1234-1234-12345678abcd}满足前面条件的假flag，进行动调，找key和加密后的结果

```
45  while ( v17 < 32 )
46  {
47    v14 = v15;
48    v19 = v15 + ((v15 >> 5) ^ (16 * v15));
49    v20 = *(_QWORD *)(a7 + 32);
50    v21 = *(_QWORD *)(a7 + 24);
51    v22 = v18;
52    v23 = v18 & 3;
53    if ( v23 >= v20 )
54      runtime_panicindex(v20, v23, v16, a7);
55    v24 = v16 + (v19 ^ (v22 + *(_DWORD *)(v21 + 4 * v23)));
56    v18 = (unsigned int)(v22 + 0x12345678);
57    v25 = ((unsigned int)v18 >> 11) & 3;
58    if ( v25 >= v20 )
59      runtime_panicindex(v20, v18, v25, a7);
60    ++v17;
```


v21在50行进行了赋值，所以我们把断点下在51行

在Debugger->Debugger windows->Locals 找到v21

右键jump就能找到key了

```
debug033:000000C00006278F db  0Fh
debug033:000000C000062790 dd 10203h
debug033:000000C000062794 dd 4050607h
debug033:000000C000062798 dd 8090A0Bh
debug033:000000C00006279C dd 0C0D0E0Fh
debug033:000000C0000627A0 db    0
```

我们找到密钥后开始找加密后的flag

```
114      v36 - 8,
115      v37 - 8);
116    internal_bytealg_Equal(v0, v39, v21, v22, v23, v24, v41, 16LL, 16LL, (__int64)&v38, 16LL);
117    if ( v35 )
```

这就是个比较flag的函数，我们在里面下断点

```
1 void __fastcall internal_bytealg_Equal(int a1, int
2 {
3   if ( a8 == a11 && a7 != a10 )
4     JUMPOUT(0x402580LL);
5 }
```

```
.text:00000000004024EF mov     rsi, [rsp+arg_0]
.text:00000000004024F4 mov     rdi, [rsp+arg_18]
RIP .text:00000000004024F9 cmp     rsi, rdi
.text:00000000004024FC jz      short loc_40250E
.text:00000000004024FE lea     rax, [rsp+arg_30]
```

加密后的flag在这个里面

得到加密后的值：0EC311F045C79AF3EDF5D910542702CB

然后就是脚本了

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <math.h>
#define tea_DELTA 0x12345678


/*
xtea加密函数
num_rounds          加密轮数
uint32_t* origin    为要加密的数据是两个32位无符号整数
uint32_t* k         为加密解密密钥，为4个32位无符号整数，即密钥长度为128位
*/
void xtea_encode(unsigned int num_rounds, uint32_t origin[2], uint32_t const key[4]) {
    unsigned int i;
    uint32_t v0 = origin[0], v1 = origin[1], sum = 0, delta = 0x9E3779B9;
    for (i = 0; i < num_rounds; i++) {
        v0 += (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
        sum += delta;
        v1 += (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum >> 11) & 3]);
    }
    origin[0] = v0; origin[1] = v1;
}
/*
xtea解密函数
num_rounds          加密轮数
uint32_t* origin    为要加密的数据是两个32位无符号整数
uint32_t* k         为加密解密密钥，为4个32位无符号整数，即密钥长度为128位
*/
void xtea_decode(unsigned int num_rounds, uint32_t origin[2], uint32_t const key[4]) {
    unsigned int i;
    uint32_t v0 = origin[0], v1 = origin[1], delta = 0x12345678, sum = delta * num_rounds;
    for (i = 0; i < num_rounds; i++) {
        v1 -= (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum >> 11) & 3]);
        sum -= delta;
        v0 -= (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
}
    origin[0] = v0; origin[1] = v1;
}


int main()
{
    uint32_t v_part1[2]={0x0EC311F0, 0x45C79AF3}; //动调得到的加密后的值,内存中就是小端储存我们不用再手动改变0EC311F0 4
5C79AF3 EDF5D910 542702CB
    uint32_t v_part2[2]={0xEDF5D910, 0x542702CB};
    uint32_t const k[4]={0x10203, 0x4050607, 0x8090A0B, 0x0C0D0E0F}; //动调得到的key
    unsigned int r=32;
    xtea_decode(r, v_part1, k);
    xtea_decode(r, v_part2, k);
    printf("解密后的数据：%x %x %x %x\n",v_part1[0],v_part1[1], v_part2[0], v_part2[1]);
    return 0;
}
```