2017年强网杯hide逆向

原创

追寻520 于 2019-06-19 20:39:32 发布 131 收藏
分类专栏： 逆向 文章标签： 逆向 ctf

1.先用file命令查看文件基本信息，文件为64位ELF文件，剥去了符号表信息。

```
zx@Emmanuel:~$ file /mnt/hgfs/re/强网杯赛题/tmp1/hide
/mnt/hgfs/re/强网杯赛题/tmp1/hide: ELF 64-bit LSB executable, x86-64, version 1
(GNU/Linux), statically linked, stripped
```

2.用checksec查看文件，文件加了upx壳。

```
zx@Emmanuel:~$ checksec /mnt/hgfs/re/强网杯赛题/tmp1/hide
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/zx/.pwntools-cache/
update to 'never'.
[*] You have the latest version of Pwntools (3.14.0.dev0)
[*] '/mnt/hgfs/re/\xe5\xbc\xba\xe7\xbd\x91\xe6\x9d\xaf\xe8\xb5\x9b\xe9\xa2\x98/t
mp1/hide'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
    Packer:   Packed with UPX
```


3.ida静态分析，大概只看到了几个地址和长度。

```
LOAD:000000000044F1C5                 pop     rbx
LOAD:000000000044F1C6
LOAD:000000000044F1C6 loc_44F1C6:                             ; CODE XREF: LOAD:loc_44F151↑j
LOAD:000000000044F1C6                 push    1
LOAD:000000000044F1C8                 push    40000Ch
LOAD:000000000044F1CD                 push    rax
LOAD:000000000044F1CE                 push    2DBB50h
LOAD:000000000044F1D3                 push    rcx
LOAD:000000000044F1D4                 push    r15
LOAD:000000000044F1D6                 mov     edi, 800000h    ; addr
LOAD:000000000044F1DB                 push    7
LOAD:000000000044F1DD                 pop     rdx             ; prot
LOAD:000000000044F1DE                 mov     esi, 2997072    ; len
LOAD:000000000044F1E3                 push    32h
LOAD:000000000044F1E5                 pop     r10             ; flags
LOAD:000000000044F1E7                 sub     r8d, r8d        ; fd
LOAD:000000000044F1EA                 push    9
LOAD:000000000044F1EC                 pop     rax
LOAD:000000000044F1ED                 syscall                 ; LINUX - sys_mmap
LOAD:000000000044F1EF                 cmp     edi, eax
LOAD:000000000044F1F1                 jnz     loc_44F0EB
LOAD:000000000044F1F7                 mov     esi, offset dword_400000
LOAD:000000000044F1FC                 mov     edx, edi
LOAD:000000000044F1FE                 sub     edx, esi
LOAD:000000000044F200                 jz      short loc_44F217
LOAD:000000000044F202                 add     ebp, edx
LOAD:000000000044F204                 add     [rsp+28h+var_20], edx
LOAD:000000000044F208                 add     [rsp+28h+var_10], edx
LOAD:000000000044F20C
LOAD:000000000044F20C loc_44F20C:                             ; CODE XREF: LOAD:000000000044F1A5↑j
```

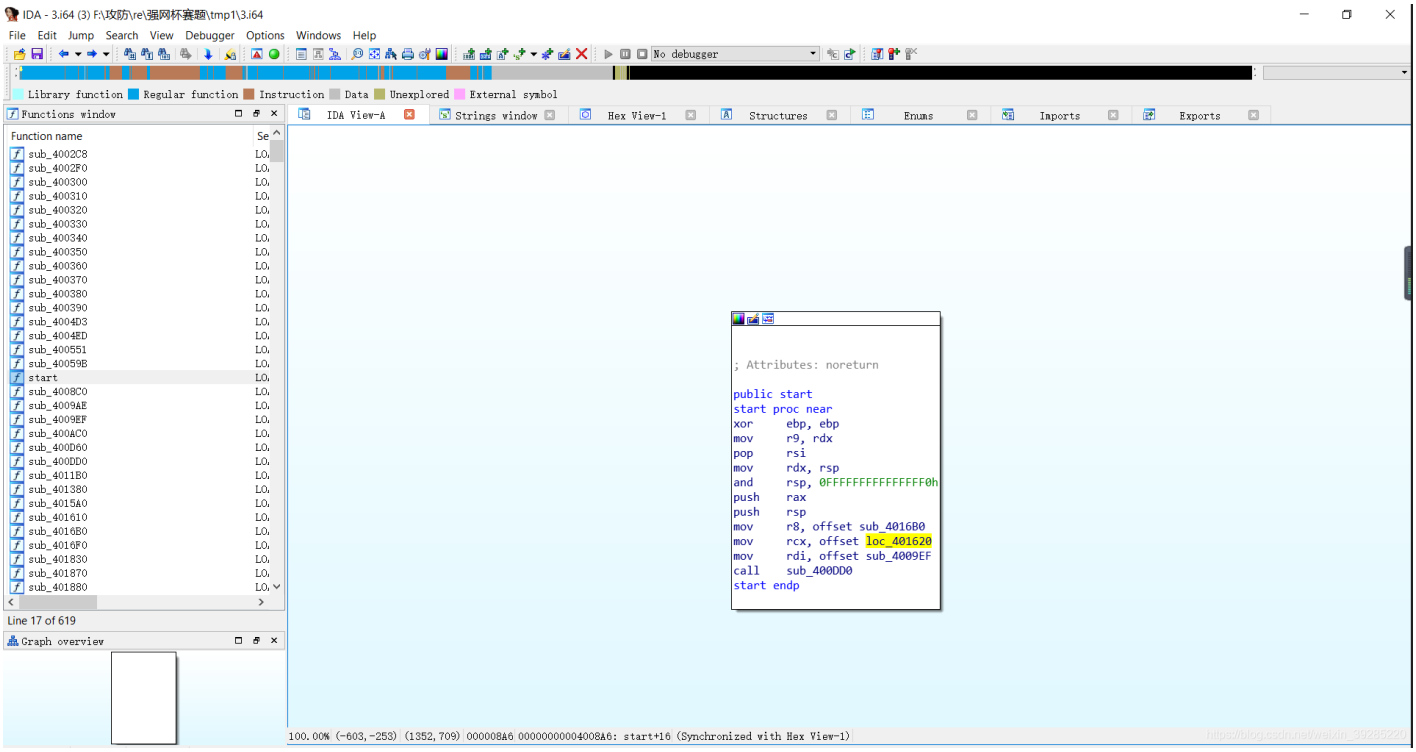
4.gbd动态调试一下，发现起始地址是从0x400000开始的，一运行程序自动退出，应该是开启了反调试，那么下面直接运行程序，把内存dump出来看看。

```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00450000         r-xp      /mnt/hgfs/re/强网杯赛题/tmp1/hide
0x006cc000         0x006cd000         rwxp      /mnt/hgfs/re/强网杯赛题/tmp1/hide
0x00007ffff7ffa000 0x00007ffff7ffd000 r--p      [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 r-xp      [vdso]
0x00007ffffffde000 0x00007ffffffff000 rwxp      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
gdb-peda$ c
Continuing.
[Inferior 1 (process 72893) exited normally]
Warning: not running
gdb-peda$
```


5.直接运行程序，从0x400000开始，尝试dump出来一些内存，字节长度尝试2997072。dump了快10分钟，真的dump了一些东西。

```
zx@Emmanuel:~$ sudo dd if=/proc/9058/mem of=/mnt/hgfs/re/3 bs=1 count=2997972 if
lag=skip_bytes skip=$[0x400000]
dd: /proc/9058/mem: cannot skip to specified offset
dd: error writing '/mnt/hgfs/re/3': Input/output error
1987643+0 records in
1987642+0 records out
1987642 bytes (2.0 MB, 1.9 MiB) copied, 513.647 s, 3.9 kB/s
```

6.ida分析dump出来的文件



找到start函数，start函数中赋值给rdi的就是main函数了，找到main函数，就好分析了。

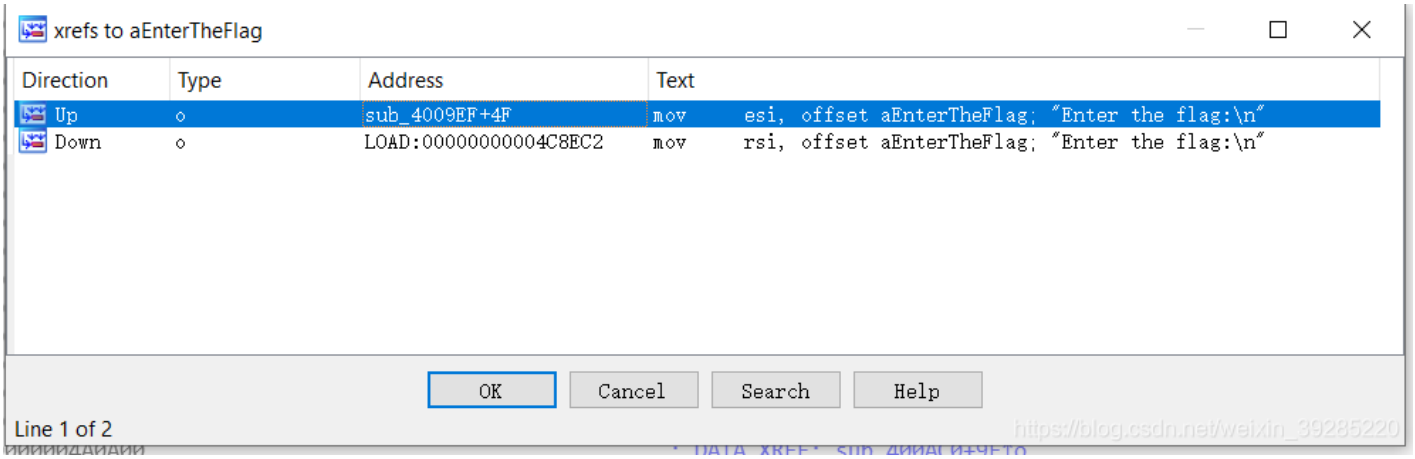
```
__int64 sub_4009EF()
{
  const char *v0; // rsi
  __int64 v1; // rdx
  __int64 result; // rax
  __int64 v3; // rcx
  unsigned __int64 v4; // rt1
  char v5; // [rsp+10h] [rbp-70h]
  unsigned __int64 v6; // [rsp+78h] [rbp-8h]

  v6 = __readfsqword(0x28u);
  if ( sub_43F380(0LL, 0LL, 0LL, 0LL) )
    sub_40EAD0(0LL);
  sub_43E9B0(1LL, (__int64)"Enter the flag:\n");
  sub_43E950(0LL, (__int64)&v5);
  if ( (unsigned int)sub_4009AE((__int64)&v5) != 0 )
  {
    v0 = "You are right\n";
    sub_43E9B0(1LL, (__int64)"You are right\n");
  }
  else
  {
    v0 = "You are wrong\n";
    sub_43E9B0(1LL, (__int64)"You are wrong\n");
  }
  result = 0LL;
  v4 = __readfsqword(0x28u);
  v3 = v4 ^ v6;
  if ( v4 != v6 )
    sub_442480(1LL, v0, v1, v3);
  return result;
}
```


看这段代码，关键就是要分析sub_4009AE函数，结果点开以后，直接告诉flag是错误的。

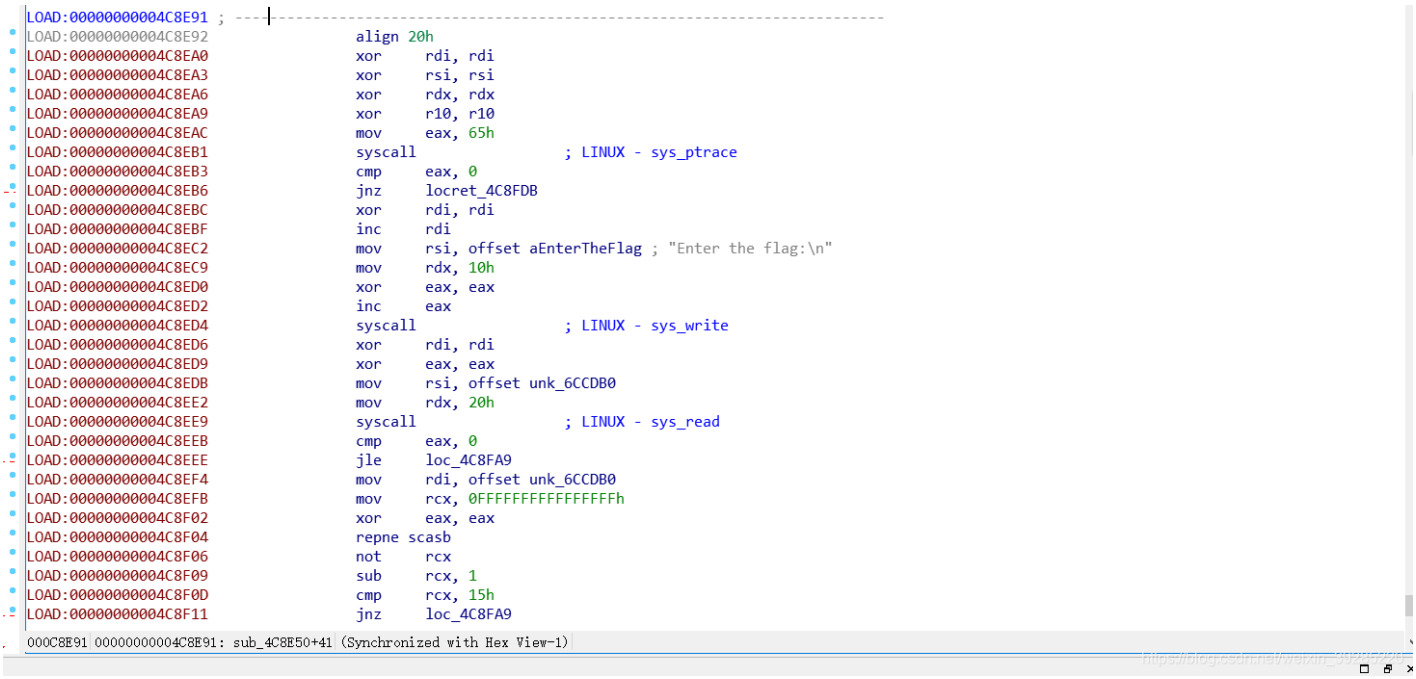
IDA View-A | Pseudocode-C | Pseudocode-B | Pseudocode-A | Strings

```
_BOOL8 __fastcall sub_4009AE(__int64 a1)
{
  return (unsigned int)sub_400360(a1, (__int64)"qwb{this_is_wrong_flag}") == 0;
}
```

思路一下就断了，这个函数应该是估计是个假函数，那么就用那个关键字符串再搜索一下，看看有没有其他地方还有提示。查找发现“Enter the flag：”出现了两次，第一次就是上面的函数，那现在来看看下面那个地方。



分析这段汇编代码，发现用户输入的字符串存到了unk_6CCDB0中。



对这段代码进行反编译，提示没有函数，那么新建一个函数，在jle跳转指令之后添加一个函数。这样按f5就能反编译了。

```
LOAD:00000000004C8EF4
LOAD:00000000004C8EF4
LOAD:00000000004C8EF4 sub_4C8EF4      proc near
LOAD:00000000004C8EF4                 mov     rdi, offset unk_6CCDB0
LOAD:00000000004C8EFB                 mov     rcx, 0FFFFFFFFFFFFFFFFh
LOAD:00000000004C8F02                 xor     eax, eax
LOAD:00000000004C8F04                 repne scasb
LOAD:00000000004C8F06                 not     rcx
LOAD:00000000004C8F09                 sub     rcx, 1
LOAD:00000000004C8F0D                 cmp     rcx, 15h
LOAD:00000000004C8F11                 jnz     loc_4C8FA9
LOAD:00000000004C8F17                 mov     rdi, offset unk_6CCDB0
LOAD:00000000004C8F1E                 cmp     byte ptr [rdi+1], 77h
LOAD:00000000004C8F22                 jnz     loc_4C8FA9
LOAD:00000000004C8F28                 cmp     byte ptr [rdi+2], 62h
LOAD:00000000004C8F2C                 jnz     short loc_4C8FA9
LOAD:00000000004C8F2E                 cmp     byte ptr [rdi+3], 7Bh
LOAD:00000000004C8F32                 jnz     short loc_4C8FA9
LOAD:00000000004C8F34                 cmp     byte ptr [rdi+14h], 7Dh
LOAD:00000000004C8F38                 jnz     short loc_4C8FA9
LOAD:00000000004C8F3A                 mov     rdi, offset unk_6CCDB4
LOAD:00000000004C8F41                 call    sub_4C8CC0
LOAD:00000000004C8F46                 mov     rdi, offset unk_6CCDB4
LOAD:00000000004C8F4D                 call    sub_4C8E50
LOAD:00000000004C8F52                 mov     rdi, offset unk_6CCDB4
LOAD:00000000004C8F59                 call    sub_4C8CC0
LOAD:00000000004C8F5E                 mov     rdi, offset unk_6CCDB4
LOAD:00000000004C8F65                 call    sub_4C8E50
LOAD:00000000004C8F6A                 mov     rdi, offset unk_6CCDB4
LOAD:00000000004C8F71                 call    sub_4C8CC0
LOAD:00000000004C8F76                 mov     rdi, offset unk_6CCDB4
LOAD:00000000004C8F7D                 call    sub_4C8E50
LOAD:00000000004C8F82                 mov     rsi, offset qword_4C8CB0
LOAD:00000000004C8F89                 mov     rdx, 0
LOAD:00000000004C8F90
```


看到了核心算法部分。输入的字符串格式应该为qwb{XXXXXX}，中间有16个字符.输入字符串存储在unk_6CCDB0中，而后面处理的数组为unk_6CCDB4, 刚好是中间那16个字符。

```
signed __int64 sub_4C8EF4()
{
  _BYTE *v0; // rdi
  __int64 *v1; // rsi
  unsigned __int64 v2; // rdx
  signed __int64 result; // rax

  if ( strlen((const char *)&unk_6CCDB0) == 21
    && *((_BYTE *)&unk_6CCDB0 + 1) == 'w'
    && *((_BYTE *)&unk_6CCDB0 + 2) == 'b'
    && *((_BYTE *)&unk_6CCDB0 + 3) == '{'
    && *((_BYTE *)&unk_6CCDB0 + 20) == '}' )
  {
    sub_4C8CC0((__int64)&unk_6CCDB4);
    sub_4C8E50((__int64)&unk_6CCDB4);
    sub_4C8CC0((__int64)&unk_6CCDB4);
    sub_4C8E50((__int64)&unk_6CCDB4);
    sub_4C8CC0((__int64)&unk_6CCDB4);
    v0 = &unk_6CCDB4;
    sub_4C8E50((__int64)&unk_6CCDB4);
    v1 = qword_4C8CB0;
    v2 = 0LL;
    while ( v2 < 0x10 && *v0 == *(_BYTE *)v1 )
    {
      ++v2;
      ++v0;
      v1 = (__int64 *)((char *)v1 + 1);
    }
  }
  __asm { syscall; LINUX - sys_write }
  result = 60LL;
  __asm { syscall; LINUX - sys_exit }
  return result;
}
```



```
.tls:00000000006CCDB0 unk_6CCDB0      db    0                 ; DATA XREF: LOAD:00000000004C8EDB↑o
.tls:00000000006CCDB0                                         ; sub_4C8EF4↑o ...
.tls:00000000006CCDB1                 db    0
.tls:00000000006CCDB2                 db    0
.tls:00000000006CCDB3                 db    0
.tls:00000000006CCDB4 unk_6CCDB4      db    0                 ; DATA XREF: sub_4C8EF4+46↑o
.tls:00000000006CCDB4                                         ; sub_4C8EF4+52↑o ...
.tls:00000000006CCDB5                 db    0
.tls:00000000006CCDB6                 db    0
.tls:00000000006CCDB7                 db    0
.tls:00000000006CCDB8                 db    0
.tls:00000000006CCDB9                 db    0
.tls:00000000006CCDBA                 db    0
.tls:00000000006CCDBB                 db    0
.tls:00000000006CCDBC                 db    0
.tls:00000000006CCDBD                 db    0
.tls:00000000006CCDBE                 db    0
.tls:00000000006CCDBF                 db    0
.tls:00000000006CCDC0                 db    0
.tls:00000000006CCDC1                 db    0
.tls:00000000006CCDC2                 db    0
.tls:00000000006CCDC3                 db    0
.tls:00000000006CCDC4                 db    0
.tls:00000000006CCDC5                 db    0
.tls:00000000006CCDC6                 db    0
.tls:00000000006CCDC7                 db    0
.tls:00000000006CCDC8                 db    0
.tls:00000000006CCDC9                 db    0
.tls:00000000006CCDCA                 db    0
```

分析代码发现，数组主要进过两个函数处理，共处理六次，最后得到的加密数组qword_4C8CB0逐位比较，相等即可。

```
LOAD:00000000004C8CB0 qword_4C8CB0     dq 1BF28C357F13B852h, 311E4F73D28663F4h
```

qword_4C8CB0为16字节采用小端存储，所以恢复成字节数组时，要逆序。

```
byte target[] = { 0x52, 0xb8, 0x13, 0x7f, 0x35, 0x8c, 0xf2, 0x1b, 0xf4, 0x63, 0x86, 0xd2, 0x73, 0x4f, 0x1e, 0x31 };
```

函数sub_4C8CC0 主要是通过中间的for循环进行加密。

```
unsigned __int64 __fastcall sub_4C8CC0(__int64 input_str)
{
  unsigned __int64 result; // rax
  unsigned __int64 v2; // rt1
  unsigned int v3; // [rsp+18h] [rbp-48h]
  __int64 v4; // [rsp+1Ch] [rbp-44h]
  signed int i; // [rsp+24h] [rbp-3Ch]
  signed int j; // [rsp+28h] [rbp-38h]
  int v7; // [rsp+40h] [rbp-20h]
  int v8; // [rsp+44h] [rbp-1Ch]
  int v9; // [rsp+48h] [rbp-18h]
  int v10; // [rsp+4Ch] [rbp-14h]
  unsigned __int64 v11; // [rsp+58h] [rbp-8h]

  v11 = __readfsqword(0x28u);
  v7 = 1883844979;
  v8 = 1165112144;
  v9 = 2035430262;
  v10 = 861484132;
  for ( i = 0; i <= 1; ++i )
  {
    v3 = *(_DWORD *)(8 * i + input_str);
    v4 = *(unsigned int *)(input_str + 4 + 8 * i);
    for ( j = 0; j <= 7; ++j )
    {
      v3 += (*(&v7 + (BYTE4(v4) & 3)) + HIDWORD(v4)) ^ ((((unsigned int)v4 >> 5) ^ 16 * v4) + v4);
      HIDWORD(v4) += 0x676E696C;
      LODWORD(v4) = ((*(&v7 + ((HIDWORD(v4) >> 11) & 3)) + HIDWORD(v4)) ^ (((v3 >> 5) ^ 16 * v3) + v3)) + v4;
    }
    *(_DWORD *)(input_str + 8 * i) = v3;
    *(_DWORD *)(input_str + 4 + 8 * i) = v4;
  }
  v2 = __readfsqword(0x28u);
  result = v2 ^ v11;
  if ( v2 != v11 )
    result = ((__int64 (*)(void))loc_4C8B9A)();
  return result;
}
```


sub_4C8E50函数进行逐字节异或处理。

```
_BYTE *__fastcall sub_4C8E50(__int64 a1)
{
  _BYTE *result; // rax
  signed int i; // [rsp+14h] [rbp-4h]

  for ( i = 0; i <= 15; ++i )
  {
    result = (_BYTE *)(i + a1);
    *result ^= i;
  }
  return result;
}
```


最后加密要得到的目标数组即为qword_4C8CB0。

至此程序已经分析完毕。

7.加密算法逆向。

```
#include <stdio.h>
#include "windows.h"
#include <string.h>
#include <stdlib.h>
#include "defs.h"
uint64 v4_4_arr[9];
int array[4] = { 0x70493173, 0x45723350, 0x79523376, 0x33593464 };

void xor16(byte byte_arr_16[]){
    for (int i = 0; i < 16; i++)
        byte_arr_16[i] = byte_arr_16[i]^i;
}
int bytesToInt(byte* bytes, int size = 4)
{
    int addr = bytes[0] & 0xFF;
    addr |= ((bytes[1] << 8) & 0xFF00);
    addr |= ((bytes[2] << 16) & 0xFF0000);
    addr |= ((bytes[3] << 24) & 0xFF000000);
    return addr;
}
void  intToByte(int i, byte *bytes, int size = 4)

{
    memset(bytes, 0, sizeof(byte)*  size);
    bytes[0] = (byte)(0xff & i);
    bytes[1] = (byte)((0xff00 & i) >> 8);
    bytes[2] = (byte)((0xff0000 & i) >> 16);
    bytes[3] = (byte)((0xff000000 & i) >> 24);
    return;
}
```



```
void re_block(byte byte_arr_8[]){
    int i = 0;
    uint32 v3 = bytesToInt(byte_arr_8);
    uint32 v4 = bytesToInt(byte_arr_8 + 4);
    for (i = 7; i > -1; i--)
    {
        uint64  v30 = (v3 << 4) & 0xffffffff;
        uint64 v2c = v3 >> 5;
        uint64 edx = v30 ^ v2c;
        v30 = (v3 + edx) & 0xffffffff;
        uint64 v28 = (v4_4_arr[i + 1] >> 11) & 3;
        edx = array[v28];
        v2c = (v4_4_arr[i + 1] + edx) & 0xffffffff;
        v4 = (v4  - (v30 ^ v2c)) & 0xffffffff;
        v30 = (v4 << 4) & 0xffffffff;
        v2c = v4 >> 5;
        edx = v30 ^ v2c;
        v30 = (v4 + edx) & 0xffffffff;
        v28 = v4_4_arr[i] & 3;
        edx = array[v28];
        v2c = (v4_4_arr[i] + edx) & 0xffffffff;
        v3 = (v3  - (v30 ^ v2c)) & 0xffffffff;
        intToByte(v3, byte_arr_8);
        intToByte(v4, byte_arr_8 + 4);
    }
}
```

```
byte * re_all(byte str16[]){
    byte byte_arr[16];
    memcpy(byte_arr, str16, sizeof(byte)*16);
    xor16(byte_arr);
    re_block(byte_arr);
    re_block(byte_arr + 8);
    xor16(byte_arr);
    re_block(byte_arr);
    re_block(byte_arr + 8);
    xor16(byte_arr);
    re_block(byte_arr);
    re_block(byte_arr + 8);
    return byte_arr;

}


int main()
{


    for (int i = 1; i < 9; i++)
        v4_4_arr[i] = (v4_4_arr[i - 1] + 0x676E696C) & 0xFFFFFFFF;

    byte target[] = { 0x52, 0xb8, 0x13, 0x7f, 0x35, 0x8c, 0xf2, 0x1b, 0xf4, 0x63, 0x86, 0xd2, 0x73, 0x4f, 0x1e, 0x31 };
    byte rs[16];
    memcpy(rs, re_all(target), sizeof(byte)* 16);
    for (int i = 0; i < 16; i++)
        printf("%c", rs[i]);



}
```