通过堆进行信息泄漏

什么叫信息泄漏

在CTF中，Pwn题目一般都是运行在远端服务器上的。因此我们不能获知服务器上的libc.so地址、Heap基地址等地址信息，但是在进行利用的时候往往需要这些地址，此时就需要进行信息泄漏。

信息泄漏的目标

信息泄漏的目标有哪些？我们可以通过观察内存空间来获知这一点

```
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x /home/pwn
0x0000000000600000 0x0000000000601000 0x0000000000000000 r-- /home/pwn
0x0000000000601000 0x0000000000602000 0x0000000000001000 rw- /home/pwn
0x0000000000602000 0x0000000000623000 0x0000000000000000 rw- [heap]
0x00007ffff7a0d000 0x00007ffff7bcd000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcd000 0x00007ffff7dcd000 0x00000000001c0000 --- /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 0x00000000001c0000 r-- /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd1000 0x00007ffff7dd3000 0x00000000001c4000 rw- /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd7000 0x0000000000000000 rw-
0x00007ffff7dd7000 0x00007ffff7dfd000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fdb000 0x00007ffff7fde000 0x0000000000000000 rw-
0x00007ffff7ff6000 0x00007ffff7ff8000 0x0000000000000000 rw-
0x00007ffff7ff8000 0x00007ffff7ffa000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 0x0000000000000000 r-x [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000025000 r-- /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x0000000000026000 rw- /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

首先第一个是主模块的基地址，因为只有在开启PIE(地址无关代码)的情况下主模块的基地址才会发生改变，因此通常情况下主模块的地址不需要泄漏。 第二个是堆地址，堆地址对于进程来说是每次运行都会改变，当然需要控制堆中的数据时可能就需要先泄漏堆基地址。 第三个是libc.so的地址，在很多情况下我们只有通过libc中的system等函数才能实现代码执行，并且malloc_hook、one_gadgets、IO_FILE等结构也都储存在libc中，因此libc的地址也是我们泄漏的目标。

通过什么进行泄漏

通过前面的知识我们知道heap分为unsorted bin、fastbin、smallbin、large bin等，我们逐个考察这些结构来查看如何进行泄漏。

unsorted bin

我们构造两个unsorted bin然后查看它的内存，现在在unsorted bin链表中存在两个块，第一个块的地址是0x602000、第二个块的地址是0x6020f0

```
0x602000:   0x0000000000000000  0x00000000000000d1
0x602010:   0x00007ffff7dd1b78  0x00000000006020f0 <=== 指向下一个块
0x602020:   0x0000000000000000  0x0000000000000000
0x602030:   0x0000000000000000  0x0000000000000000
0x6020f0:   0x0000000000000000  0x00000000000000d1
0x602100:   0x0000000000602000  0x00007ffff7dd1b78 <=== 指向main_arena
0x602110:   0x0000000000000000  0x0000000000000000
0x602120:   0x0000000000000000  0x0000000000000000
```

因此我们知道通过unsorted bin我们可以获取到某个堆块的地址和main_areana的地址。一旦获取到某个堆块的地址就可以通过malloc的size进行计算从而获得堆基地址。一旦获取到main_arena的地址，因为main_arena存在于libc.so中就可以计算偏移得出libc.so的基地址。 因此，通过unsorted bin可以获得：1.libc.so的基地址 2.heap基地址

fastbin

我们构造了两个fastbin然后查看它们的内存，现在在fastbin链表中存在两个块，第一个块的地址是0x602040，第二个块的地址是0x602000

```
0x602000:   0x0000000000000000  0x0000000000000021
0x602010:   0x0000000000000000  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000021
0x602050:   0x0000000000602000  0x0000000000000000 <=== 指向第一个块
```

根据前面的知识我们知道fastbin链表最末端的块fd域为0，此后每个块的fd域指向前一个块。因此通过fastbin只能泄漏heap的基地址

smallbin

我们构造了两个fastbin然后查看它们的内存，现在在fastbin链表中存在两个块，第一个块的地址是0x602000，第二个块的地址是0x6020f0

```
0x602000:   0x0000000000000000  0x00000000000000d1
0x602010:   0x00007ffff7dd1c38  0x00000000006020f0 <=== 下一个块的地址
0x602020:   0x0000000000000000  0x0000000000000000
0x602030:   0x0000000000000000  0x0000000000000000
0x6020f0:   0x0000000000000000  0x00000000000000d1
0x602100:   0x0000000000602000  0x00007ffff7dd1c38 <=== main_arena的地址
0x602110:   0x0000000000000000  0x0000000000000000
0x602120:   0x0000000000000000  0x0000000000000000
```

因此，通过smallbin可以获得：1.libc.so的基地址 2.heap基地址

哪些漏洞可以用于泄漏

通过前面的知识我们可以获知堆中存在哪些地址信息，但是想要获取到这些地址需要通过漏洞来实现 一般来说以下漏洞是可以进行信息漏洞的

- 堆内存未初始化
- 堆溢出
- Use-After-Free
- 越界读
- heap extend

0x01 read UAF

通过，UAF ，泄露 heapbase：

```
p0 = malloc(0x20);
p1 = malloc(0x20);

free(p0);
free(p1);

printf('heap base:%p',*p1);
```

由于 fastbin list 的特性，当我们构造一条fastbin list的时候

```
(0x30)     fastbin[1]: 0x602030 --> 0x602000 --> 0x0
```

存在 chunk 1 -> chunk 0 的现象，如果此时 UAF漏洞存在，我们可以通过 show chunk 1，将chunk 0的地址打印出来

同理，泄露 libc base

```
p0 = malloc(0x100);
free(p0);
printf("libc: %p\n", *p0);
```

- **0x02 overlapping chunks**
- **0x03 Partial Overwrite**
- **0x04 Relative Write**