Sleepy holder

19-01-17

Background

- 힙에 대한 상식들
- fastbin_dup_consolidate
- unsafe unlink

fastbin_dup_consolidate

- fast chunk (c1) 가 하나 할당되어 있고 언제든지 해제할 수 있어야함 (할당 여부 관계없이)
- c1과 인접한 free chunk가 없어야 함
- 512 bytes 이상의 large chunk를 요청할 수 있어야 함

- 그러면 fast bin 에 c1 이 있고 small bin 에도 c1이 있게 할 수 있다.
- 즉 c1 size 의 chunk 를 두 번 요청한다고 하면 한 번은 fast bin 에서, 또 한 번은 small bin 에서 c1 chunk 를 할당해준다.

fastbin_dup_consolidate

heap

fastbin[x] -> NULL

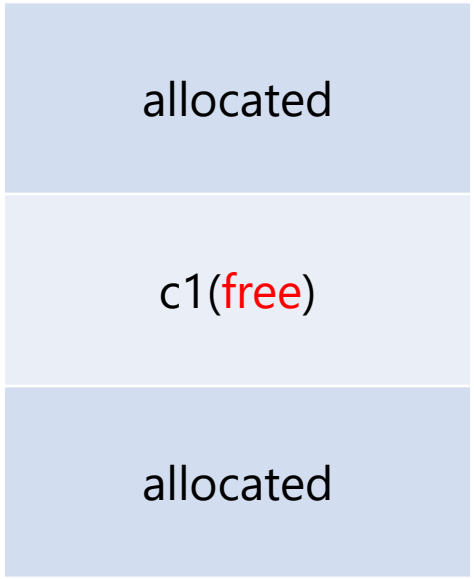
unsorted_bin <-> unsorted_bin

small_bin[x] <-> small_bin[x]

- fast chunk c1 이 할당되어 있다고 가정하자.
- 여기서 c1에 대해 free() 가 이루어지면

fastbin_dup_consolidate

heap



fastbin[x] -> c1 -> NULL

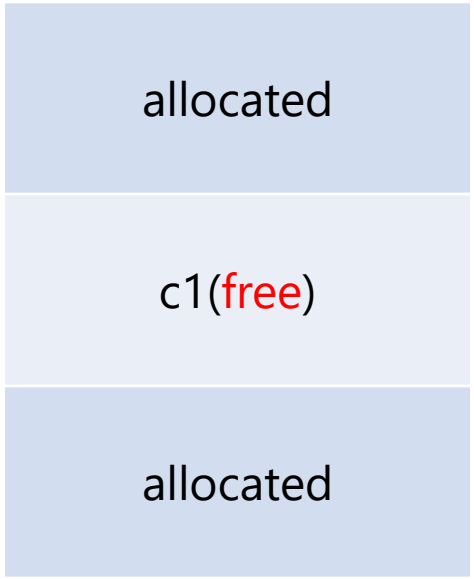
unsorted_bin <-> unsorted_bin

small_bin[x] <-> small_bin[x]

- c1이 fastbin에 추가 된다.
- 여기서 다시 c1에 대해 free() 가 이루어지면 fasttop 이 c1이기 때문에 *"double free or corruption (fasttop)"* 오류 메시지를 출력한다.
- 이 때 512 bytes 이상의 size 's'에 대한 chunk 할당을 시도 하게 되면 _int_malloc() 내부에서 malloc_consolidate() 를 호출한다.
- 그 결과로

fastbin_dup_consolidate

heap



fastbin[x] -> NULL

unsorted_bin <-> c1 <-> unsorted_bin

small_bin[x] <-> small_bin[x]

- 먼저 c1을 fastbin에서 제거하며
- c1 을 unsorted_bin에 추가한다.
- 해당 과정에서 c1 의 next chunk 의 prev size field 가 c1 chunk 의 size 로 설정되며 next chunk 의 PREV_IN_USE bit 도 0으로 설정된다.
- _int_malloc() 에서는 malloc_consolidate() 호출을 마치고 나서 unsorted_bin 에서 size ‘s’ 에 대응되는 chunk를 찾는다. unsorted_bin 에 오래 남은 chunk 일수록 먼저 검색 대상이 되는데 size 가 ‘s’가 아니라면 size 에 따라 대응되는 small or large bin으로 이동한다.
- 만약 c1이 검색 대상에 포함된다면

fastbin_dup_consolidate

heap

allocated
c1(free)
allocated

fastbin[x] -> NULL

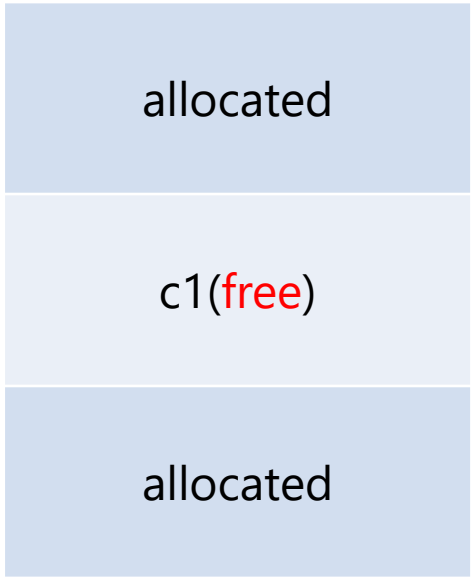
unsorted_bin <-> unsorted_bin

small_bin[x] <-> c1 <-> small_bin[x]

- c1은 unsorted_bin 에서 제거되며 small_bin 으로 옮겨간다.
- c1이 unsorted_bin에 있든 small_bin에 있든 이제 security check에 걸리지 않으면서 double free를 만들 수 있다.
- fasttop 이 c1이 아니므로 c1에 대해서 free()를 호출했을 때 예외 처리가 진행되지 않고 평범하게 루틴이 진행된다.
- 그 결과로

fastbin_dup_consolidate

heap



fastbin[x] -> c1 -> NULL

unsorted_bin <-> unsorted_bin

small_bin[x] <-> c1 <-> small_bin[x]

- c1 에 다시 fastbin에 들어간다. 즉 fast bin , small bin 둘 다 c1 을 리스트 내의 원소로 가지고 있다.
- 이제 실질적인 exploit 이 가능하다.
- 다만 fastbin_dup_into_stack 보다 fake chunk를 생성하는 게 좀 더 까다롭다.
- c1을 fastbin으로 할당 받아 fd, bk 를 조작한다고 해도 smallbin 에서 다시 c1을 할당 받을 때 또다른 security check 가 수행되기 때문에 좀 더 까다로운 면이 있다.
- c1이 unsorted_bin 에 있는 경우에는 한결 수월하다.

Unsafe Unlink

- 많은 조건이 주어져야 가능하다.
- unlink 에 대한 security checking 이 강화된 이후 해당 checking 을 우회하면서 fake chunk를 unlink 하는 기법이 나왔다.
- 대신 unlink 를 통해 바로 got를 덮을 수는 없고 주어진 프로그램에 따라 다른 방식으로 got를 덮고 shell을 획득할 수 있다.
- 자세히 건 여기서....

Binary Analysis

part of keep()

```
if ( v0 == 2 )
{
  if ( !big_allocated )
  {
    p_largeMem = calloc(1uLL, 0xFA0uLL);
    big_allocated = 1;
    puts("Tell me your secret: ");
    read(0, p_largeMem, 0xFA0uLL);
  }
}
else if ( v0 == 3 )
{
  if ( !huge_allocated )
  {
    p_hugeMem = calloc(1uLL, 0x61A80uLL);
    huge_allocated = 1;
    puts("Tell me your secret: ");
    read(0, p_hugeMem, 0x61A80uLL);
  }
}
else if ( v0 == 1 && !small_allocated )
{
  p_smallMem = calloc(1uLL, 0x28uLL);
  small_allocated = 1;
  puts("Tell me your secret: ");
  read(0, p_smallMem, 0x28uLL);
}
```

기본적으로 할당 할 수 있는 chunk의 종류가 세 개로 제한 되어 있다.

small chunk – user heap size : 0x28, chunk size : 0x30
large chunk – user heap size : 0xfa0, chunk size : 0xfb0
huge chunk – user heap size : 0x61a80, chunk size : 0x61a90

keep() 은 할당과 동시에 read() 로 user input 을 받고
renew()는 이미 할당 된 user heap 의 내용을 수정한다.

small chunk 는 fast chunk size 이고
large chunk 는 large chunk size 이고
huge chunk 는 mmap() syscall 을 통해 할당된다.

Binary Analysis

```
.bss:00000000006020C0 p_largeMem      dq ?
.bss:00000000006020C0
.bss:00000000006020C8 ; void *p_hugeMem
.bss:00000000006020C8 p_hugeMem       dq ?
.bss:00000000006020C8
.bss:00000000006020D0 ; void *p_smallMem
.bss:00000000006020D0 p_smallMem      dq ?
.bss:00000000006020D0
.bss:00000000006020D8 big_allocated   dd ?
.bss:00000000006020D8
.bss:00000000006020DC huge_allocated  dd ?
.bss:00000000006020DC
.bss:00000000006020E0 small_allocated dd ?
```

할당이 될 때 calloc() 의 return 값은 bss 영역에 저장된다. free() 할 때 역시 bss 영역에 저장된 포인터를 인자로 사용한다.

heap 영역에 대한 포인터를 저장하는 공간 이후에는 실제로 할당되었는지 확인하는 flag가 종류별로 존재한다.

Binary Analysis

```
unsigned __int64 wipe()
{
  int v0; // eax
  char s[4]; // [rsp+10h] [rbp-10h]
  unsigned __int64 v3; // [rsp+18h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  puts("Which Secret do you want to wipe?");
  puts("1. Small secret");
  puts("2. Big secret");
  memset(s, 0, 4uLL);
  read(0, s, 4uLL);
  v0 = atoi(s);
  if ( v0 == 1 )
  {
    free(p_smallMem);
    small_allocated = 0;
  }
  else if ( v0 == 2 )
  {
    free(p_largeMem);
    big_allocated = 0;
  }
  return __readfsqword(0x28u) ^ v3;
}
```

wipe 루틴에서 메모리 할당 여부를
체크하지 않고 바로 free 를 하다 보니
exploit 에 취약할 수 밖에 없다.

먼저 keep, wipe 을 반복적으로 사용하여
fastbin_dup_consolidate 를 일으킬 수 있다.
fast chunk 를 fastbin 에 그리고 smallbin 에
위치하도록 조작할 수 있다.

eXPLOIT – fastbin_dup_consolidate

```
KEEP_SMALL

KEEP_LARGE

WIPE_SMALL

KEEP_HUGE

WIPE_SMALL
```

왼 쪽과 같은 sequence 로
fastbin_dup_consolidate 를 일으킬 수 있다.

Keep 에서의 secret 내용은 중요하지 않다.

eXPLOIT – fastbin_dup_consolidate

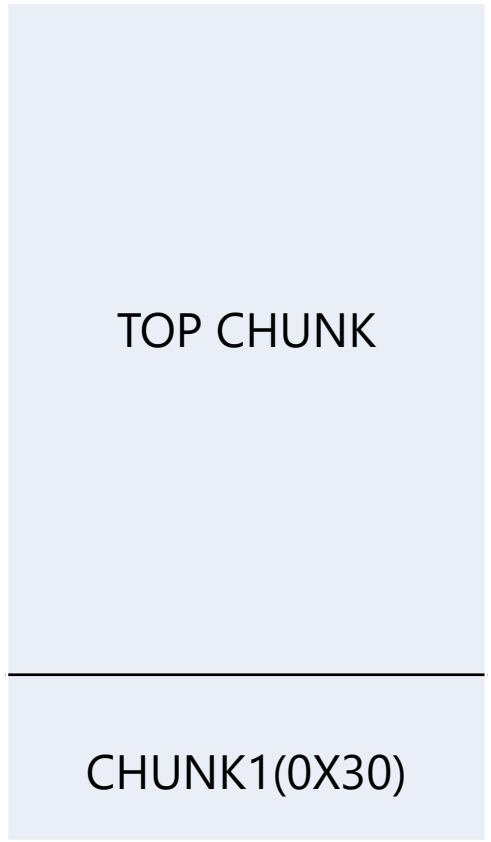
KEEP_SMALL

KEEP_LARGE

WIPE_SMALL

KEEP_HUGE

WIPE_SMALL



fastbin[x] -> NULL

unsorted_bin <-> unsorted_bin

small_bin[x] <-> small_bin[x]

eXPLOIT – fastbin_dup_consolidate

KEEP_SMALL

KEEP_LARGE

WIPE_SMALL

KEEP_HUGE

WIPE_SMALL

fastbin[x] -> NULL

unsorted_bin <-> unsorted_bin

small_bin[x] <-> small_bin[x]

eXPLOIT – fastbin_dup_consolidate

KEEP_SMALL

KEEP_LARGE

WIPE_SMALL

KEEP_HUGE

WIPE_SMALL

fastbin[x] -> CHUNK1-> NULL

unsorted_bin <-> unsorted_bin

small_bin[x] <-> small_bin[x]

eXPLOIT – fastbin_dup_consolidate

```
KEEP_SMALL

KEEP_LARGE

WIPE_SMALL

KEEP_HUGE : large request

WIPE_SMALL
```

fastbin[x] -> NULL

unsorted_bin <-> unsorted_bin

small_bin[x] <-> CHUNK1 <-> small_bin[x]

eXPLOIT – fastbin_dup_consolidate

KEEP_SMALL

KEEP_LARGE

WIPE_SMALL

KEEP_HUGE

WIPE_SMALL : double free

fastbin[x] -> CHUNK1 -> NULL

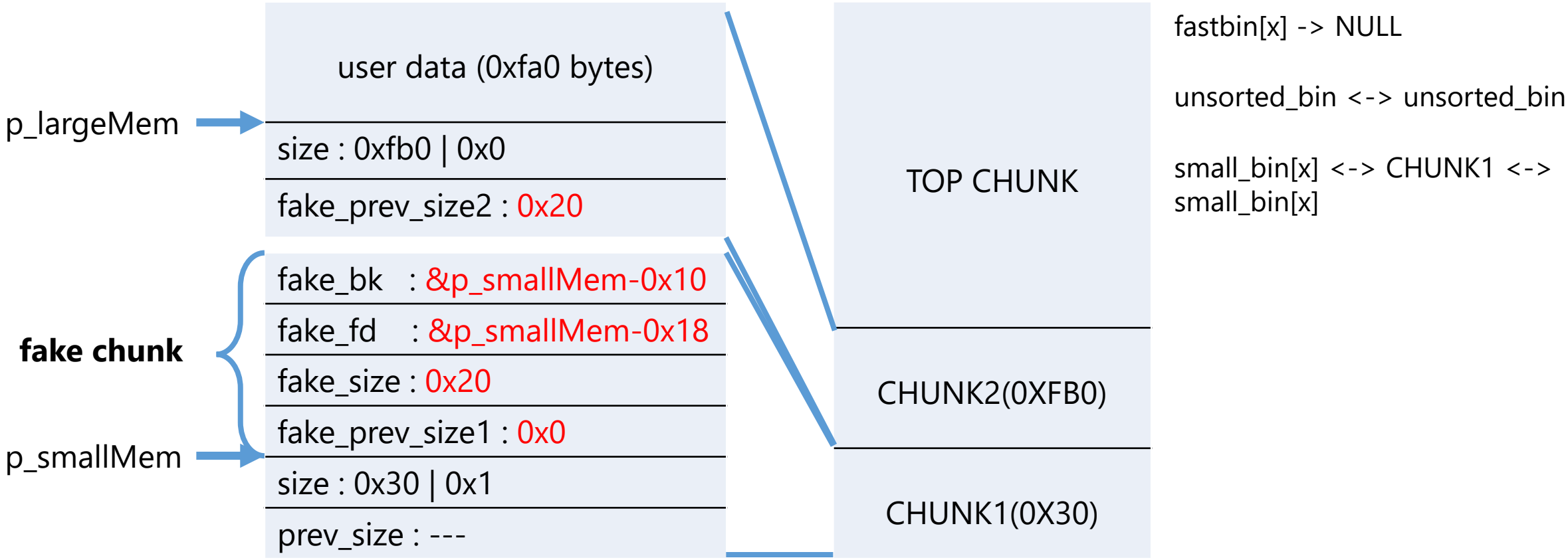
unsorted_bin <-> unsorted_bin

small_bin[x] <-> CHUNK1 <-> small_bin[x]

exploit – unsafe unlink

- chunk2 의 PREV_IN_USE bit 가 malloc_consolidate() 로 인해 0의 값을 가지므로 chunk2 에 대한 free() 가 호출되면 backward consolidate 를 시도한다.
- 이 때 우리는 unlink 의 대상이 chunk1 이 아니라 chunk1 보다 조금 앞에 있는 fake chunk 가 되게끔 fake chunk를 만들 수 있다.
- 먼저 chunk1 을 먼저 할당 받고 다음과 같은 내용을 채워 넣자.
- p64(0) + p64(0x20) + p64(&p_smallMem-0x18) + p64(&p_smallMem-0x10) + p64(0x21) (0x28b)
- 이 때 &p_smallMem 이 가리키는 영역은 small request 를 위한 calloc의 return value 가 저장되는 영역이다. (the return value of calloc for small request points to the heap area)
- 이를 자세히 나타내면

exploit – unsafe unlink



exploit – unsafe unlink

- 이제 large chunk 에 대한 free() 를 호출하면 fake chunk 와 chunk2의 병합이 진행되고 fake chunk에 대한 unlink 가 수행된다.
- 이후 large chunk with fake chunk 와 top chunk 가 인접하기 때문에 다시 두 청크가 합쳐져서 새로운 top chunk가 된다. 따라서 free() 에서 heap 영역의 메모리가 변하는 부분은 fake_size field 밖에 없다. (‘새로운 top chunk 의 크기 | 0x1’ 이 대입됨)
- 또한 freelist 역시 이전과 다를 게 없다.
- 우리가 관심있는 부분은 fake chunk에 대한 unlink 이다.
- 일단 fake chunk 가 unlink 가 수행되는 동안의 security checking 들을 어떻게 회피하는지 알아보려 한다.

exploit – unsafe unlink

```
if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))
  malloc_printerr ("corrupted size vs. prev_size");
```

- 해당 조건문은 fake chunk 의 size field 와 next chunk 의 prev_size field (fake prev_size2) 이 동일하게 0x20 이므로 pass 할 수 있다.

```
if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
  malloc_printerr ("corrupted double-linked list");
```

- 먼저 P 는 fake chunk 의 시작주소로 p_smallMem 전역변수에 저장된 포인터이다.
- FD->bk : *(FD+0x18) : *(&p_smallMem-0x18+0x18) : p_smallMem
- BK->fd : *(BK+0x10) : *(&p_smallMem-0x10+0x10) : p_smallMem
- 따라서 해당 security checking 을 아주 멋지게 pass 한다.

exploit – unsafe unlink

```
FD->bk = BK;
BK->fd = FD;
```

- 실제 unlink를 진행하면 위와 같은 operation이 진행되고 다음과 같이 해석된다.
- since FD : &p_smallMem-0x18, BK : &p_smallMem-0x10
- FD->bk = BK; ⇔ p_smallMem = &p_smallMem-0x10;
- BK->fd = FD; ⇔ p_smallMem = &p_smallMem-0x18;
- 즉 p_smallMem 변수의 담긴 값의 변화는 다음과 같다.
- p_smallMem : &fake_chunk -> &p_smallMem-0x10 -> &p_smallMem-0x18

exploit – unsafe unlink

- 이제 중요한 사실은 아까 small secret 에 대한 keep 을 호출하고 large secret 에 대한 wipe만 호출했기때문에 renew 명령어로 p_smallMem 이 가리키는 영역의 0x28 bytes를 맘대로 채워 놓을 수 있다.
- 이 때 p_smallMem 에 &p_smallMem-0x18 이 들어있기 때문에 우리가 원하는 전역 변수를 우리가 원하는 값으로 변경할 수 있다. (big_allocated 변수 까지)
- if we put the contents, then

  p_largeMem <- puts@got

  p_smallMem <- free@got

  big_allocated <- 0x1

  } 이 둘이 바뀌면 안됨

```
if ( v0 == 1 )
{
  if ( small_allocated )
  {
    puts("Tell me your secret: ");
    read(0, p_smallMem, 0x28uLL);
  }
}
```

```
00000000006020C0 ; void *p_largeMem
00000000006020C0 p_largeMem      dq ?
00000000006020C0
00000000006020C8 ; void *p_hugeMem
00000000006020C8 p_hugeMem       dq ?
00000000006020C8
00000000006020D0 ; void *p_smallMem
00000000006020D0 p_smallMem      dq ?
00000000006020D0
00000000006020D8 big_allocated   dd ?
00000000006020D8
00000000006020DC huge_allocated  dd ?
00000000006020DC
00000000006020E0 small_allocated dd ?
```

exploit – overwriting global variables & got

- p_smallMem 변수에 free@got 가 들어있으므로 smallMem 에 대한 renew 명령어를 호출해서 free@got 의 값을 바꿀 수 있다.
- leak을 위해서 free@got 에 puts@plt를 넣어주자. 이제 free() 호출 시 puts가 호출된다.
- p_largeMem 에 puts@got 가 들어있으므로 wipe_large 를 통해 free(p_largeMem) 를 호출하고 puts_addr을 leak 할 수 있다. (free(p_largeMem) -> puts(puts@got))
- 이제 다시 smallMem 에 대한 renew 명령어를 호출하여 free@got 에 system_addr을 넣어주자. (free() 호출 시 system()이 호출됨)
- keep_large 를 통해 top chunk로부터 memory를 할당 받아 “sh\0” 이란 내용을 채워 넣은 후 wipe_large 를 수행하면 free(p_largeMem) : system(p_largeMem) : system(“sh\0”) 으로 쉘을 획득할 수 있다. (★)

exploit – overwriting global variables & got

- exploit 마지막 문단 (★) 첫 부분에의 ‘keep_large 를 통해 top chunk로부터 memory를 할당 받아’ 이 절은
- unsafte unlink이후 p_largeMem <- puts@got & p_smallMem <- free@got 로 전역변수들의 값들을 덮어씌우는 과정에서 둘의 역할이 왜 바뀌면 안되는지를 간접적으로 보여준다. 만약 p_largeMem 에 free@got, p_smallMem 에 puts@got 가 대입되어 그 역할이 바뀐다면 exploit의 마지막에도 keep_small 을 통해 메모리를 할당 받고 “sh\0” 문자열을 채워 넣을 것으로 예상된다. 그러나 이 때 keep_small 은 top chunk로 부터 memory를 할당 받지 않고 small_bin에 남아있는 chunk1을 할당하려고 시도 한다.
- 즉 victim 은 chunk1이 되고 security check를 하던 도중 segmentation fault를 일으킨다.

exploit – overwriting global variables & got

```
bck = victim->bk;
if (__glibc_unlikely (bck->fd != victim))
  malloc_printerr ("malloc(): smallbin double linked list corrupted");
```

- 위는 security checking 조건문으로 chunk1 에 대한 검사를 한다.
- bck = victim->bk // equal to 0x20
- 따라서 bck->fd : *(bck+0x10) : *(0x30) 이므로 비교하기도 전에 segmentation fault를 일으킨다.

fake_bk : &p_smallMem-0x10
fake_fd : &p_smallMem-0x18
fake_size : 0x20
fake_prev_size1 : 0x0
size : 0x30 | 0x1
prev_size : ---

chunk1

reference

- write-up1 : 걍 처음부터 다 봄;; write-up 보면서 unlink 보호기법 우회 이해한듯
- ex.py : 너무 감이 안 잡혔을 때 참고한 소스. 맨처음에는 unlink 관련 security checking이 없는 구버전 libc 로 exploit 진행하는 줄 알았음;;;
- glibc malloc source & ptmalloc source 이거 필수인듯

environment information

- In WSL!
- glibc version : libc-2.23.so (do not use tcache)
- used socat to run the program in the port 10003
- debugging with peda

exploit code

- link (github)