Babyheap write-up by jaehyeon

19-01-12

Prerequisite

- structure of the heap : Understanding the glibc malloc (ptmalloc2) (en)
- malloc, free sequence : wiki (en), f/oss study (kr)
- 종합 : heap 영역 정리(kr) – 여러 매크로 정리 , malloc & free sequence 플로우 차트

- ptmalloc2의 힙 구조를 알아야 하며 malloc, free 함수의 호출 시 어떤 알고리즘이 수행되는지를 알아야 함
- ptmalloc2 코드와 위의 문서들을 반복해서 읽어볼 것 (glibc malloc code는 exploit 공부하면서 볼 것)

Binary Analysis – main function

```
struct block {
    __int32 is_allocated;    // flag field
    __int32 dummy;           // not used
    __int64 size;            // size of the following content
    __int64 p_content;       // pointer of the content
};
```

```
int main(void) {
    struct block *mmaped_addr;
    __int32 n;

    mmaped_addr = initialization();

    while (1) {
        print_menu();
        n = read_num();
        switch (n) {
            case 1:
                __alloc__(mmaped_addr);
                break;
            case 2:
                __fill__(mmaped_addr);
                break;
            case 3:
                __free__(mmaped_addr);
                break;
            case 4:
                __dump__(mmaped_addr);
                break;
            case 5:
                return 0;
            default:
                continue;
        }
    }
}
```

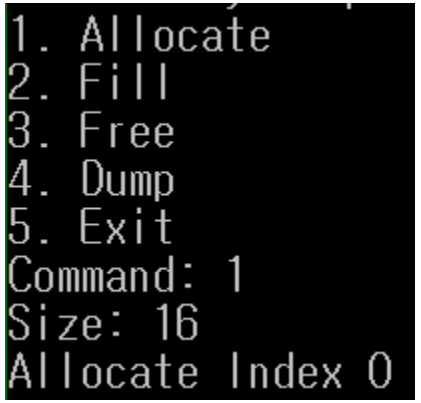
Initialization() :
- set the stdio buffer size
- call mmap that returns an area with enough size

print_menu() : prints the menu

“1. alloc, 2. fill, 3. free, 4. dump, 5. exit”

read_num() : reads a string from stdin and converts it to the corresponding integer

Binary Analysis – __alloc__ function



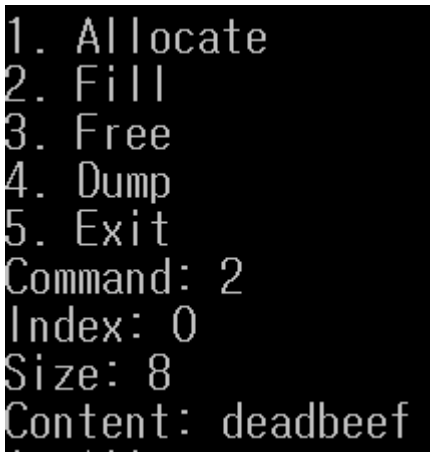
- 주어진 mmaped area를 16개의 block들로 간주한다.
- 따라서 각 블록마다 0 ~ 15 까지의 index가 주어진다.

- Allocate 명령어는 idx : 0 ~ 15 까지의 블록들을 순차 검색하여 .is_allocated flag가 설정되지 않은 첫 번째 블록에게 size 에 대응되는 heap memory space 를 넘겨준다.

- 입력 받는 size는 calloc() 인자로 들어가 calloc(size, 1)을 호출하며 return value를 블록의 .p_content field에 저장한다.

- Allocate 명령어를 통해 size 에 대응되는 heap memory를 받은 블록은 .size field 와 .is_allocated flag 가 설정된다.

Binary Analysis – __fill__ function



- index 에 대응되는 블록이 가지는 heap memory space에 값을 채우는 함수이다.
- 이 때 블록에 명시된 heap memory space의 .size field를 이용하지 않고 임의의 size를 다시 입력 받아서 writing을 진행하기 때문에 heap overflow를 일으킬 수 있다.

Binary Analysis – __free__ function

```
1. Allocate
2. Fill
3. Free
4. Dump
5. Exit
Command: 3
Index: 0
```

- index 에 대응되는 블록이 가지는 heap memory space를 free하고 블록의 .is_allocated flag 를 0으로 만든다.

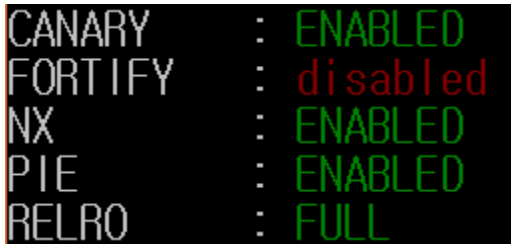
Binary Analysis – __dump__ function

```
1. Allocate
2. Fill
3. Free
4. Dump
5. Exit
Command: 1
Size: 8
Allocate Index 0
1. Allocate
2. Fill
3. Free
4. Dump
5. Exit
Command: 2
Index: 0
Size: 9
Content: deadbeef
1. Allocate
2. Fill
3. Free
4. Dump
5. Exit
Command: 4
Index: 0
Content:
deadbeef
```

- index 에 대응되는 블록이 가지는 heap memory space의 내용을 블록의 (.size field) bytes 만큼 stdout에 출력해준다.

- leak 이 일어날 가능성이 가능 높은 루틴이다.

Binary Analysis – memory protection



덕지덕지 많이도 붙여 놓았다.
GOT overwrite는 애초에 불가능하고
rop chain 생성도 code base leak 이후에나 가능하다.

Exploit

- fill 함수에서 대놓고 heap overflow를 허용해서 이를 이용하여 exploit을 진행할 수 있다.

- 첫 번째로 size x 의 fast chunk를 더 큰 size를 가진 fast chunk 인 것 처럼 free() function을 속여야 한다. 즉 size x 의 fast chunk chunk1의 size field를 x -> y 로 변조하고 (x < y) chunk1을 대상으로 free() 를 call하면 chunk1은 size y 에 대응되는 fastbin 에 저장된다.

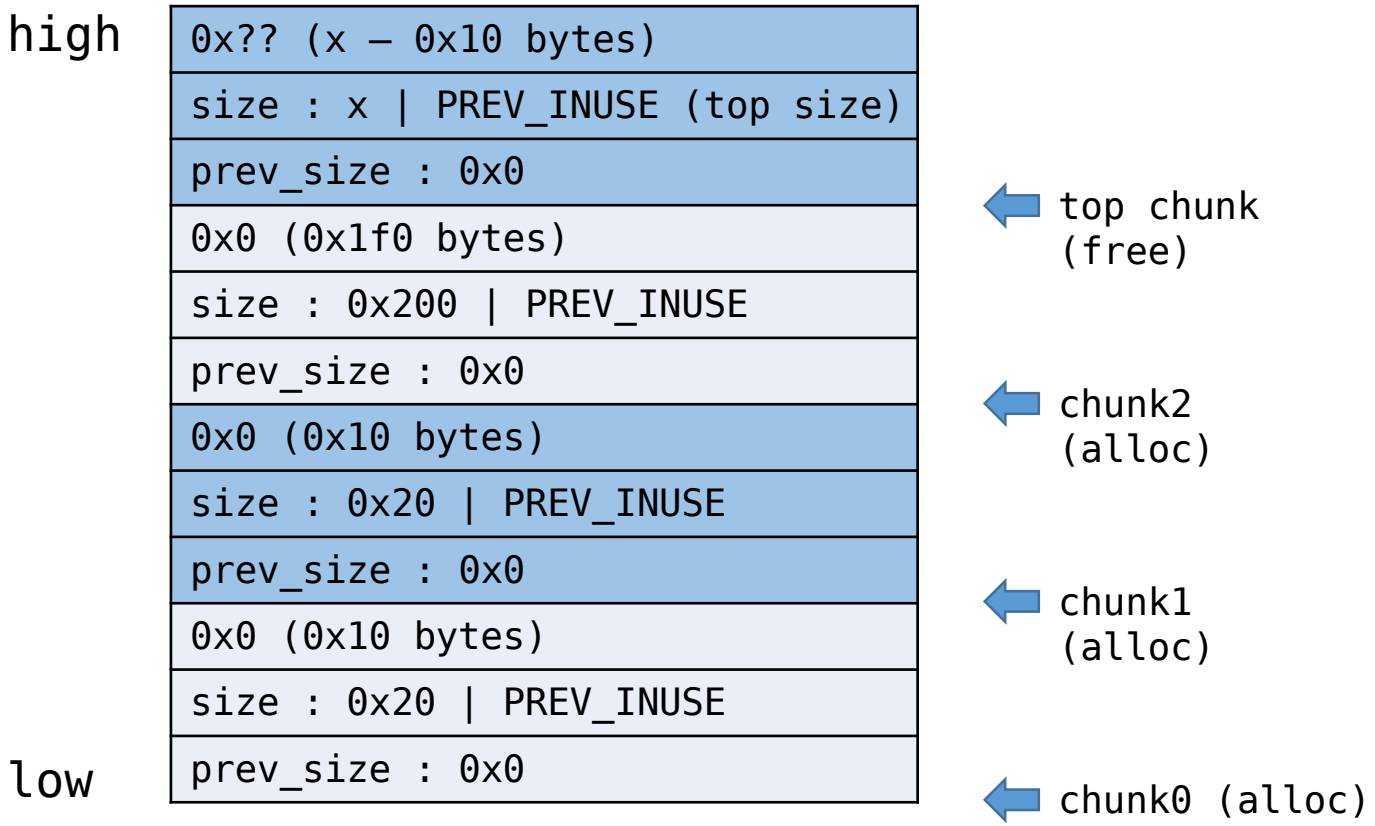
- free() 를 호출할 때 glibc 에서는 free 의 대상이 되는 chunk의 다음 chunk의 size field 값의 유효성을 검사한다. 즉 너무 작거나 너무 큰 경우 free()의 error routine으로 넘어가므로 (1)에서 chunk1의 size field 를 y로 변조할 때 (&chunk1 + y)->size 값이 적절한 범위내에 위치하도록 메모리를 변조해야 한다.

- *** tcache 를 사용하지 않는 환경에서 문제풀이를 진행할 것 ***

Exploit –libc leak

- 세 번의 allocation 을 가정하자. alloc(32-8), alloc(32-8), alloc(512-8)

- Then there exist chunk0 with size 32, chunk1 with size 32 and chunk2 with size 512



Exploit –libc leak

- 세 번의 allocation 을 가정하자. alloc(32-8), alloc(32-8), alloc(512-8)

- Then there exist chunk0 with size 32, chunk1 with size 32 and chunk2 with size 512

high

0x?? (x – 0x10 bytes)
size : x | PREV_INUSE (top size)
prev_size : 0x0
0x0 (0x1f0 bytes)
size : 0x200 | PREV_INUSE
prev_size : 0x0
0x0 (0x10 bytes)
size : 0x20 | PREV_INUSE
prev_size : 0x0
0x0 (0x10 bytes)
size : 0x20 | PREV_INUSE
prev_size : 0x0

low

top chunk (free)

chunk2 (alloc)

chunk1 (alloc)

chunk0 (alloc)

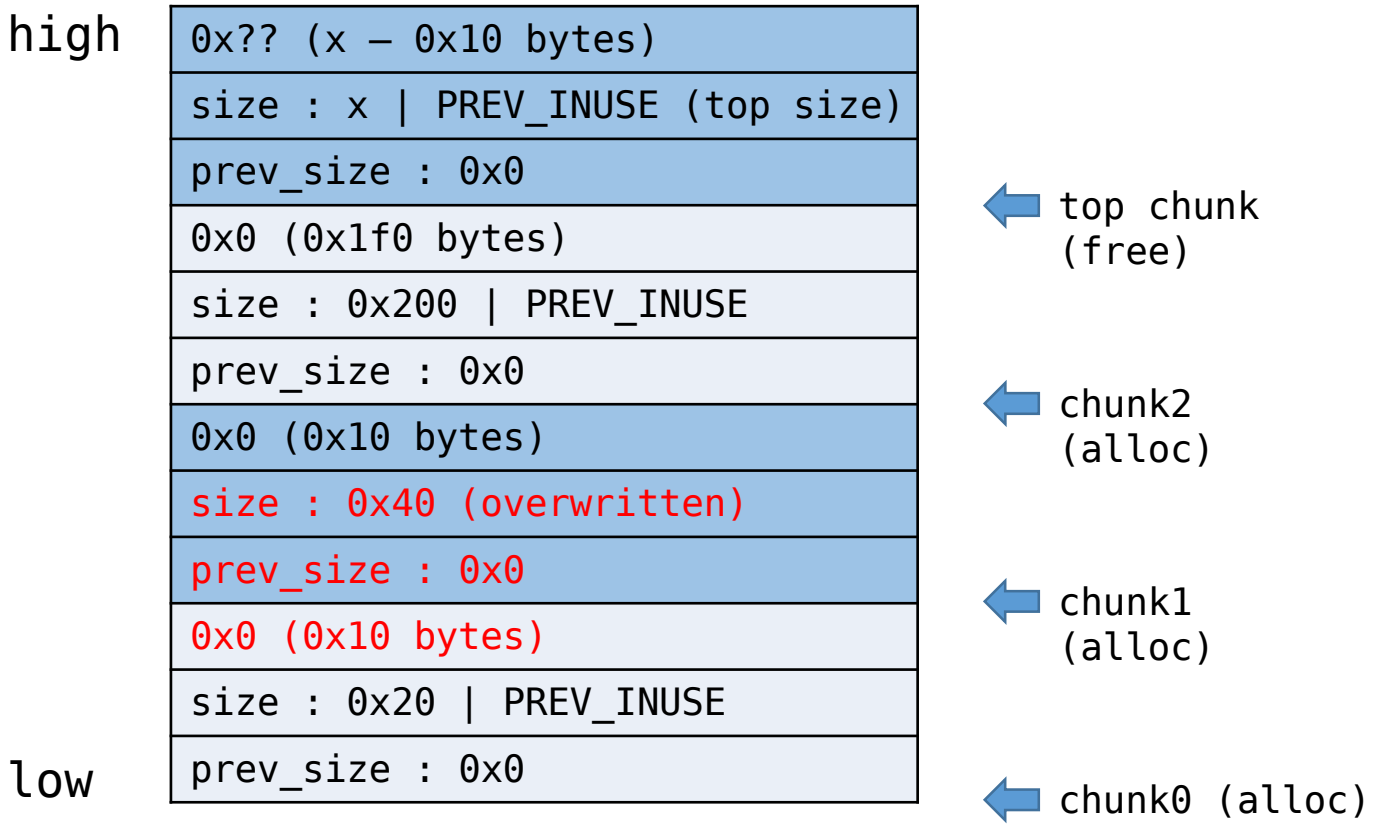
1.index 0 block 을 대상으로 fill 명령어를 수행하자.
입력 길이 제한이 없으므로 chunk1 의 size field 를 원하는 값으로 덮을 수 있다.

Assume that we overwrote it by ‘0x40’

Exploit –libc leak

- 세 번의 allocation 을 가정하자. alloc(32-8), alloc(32-8), alloc(512-8)

- Then there exist chunk0 with size 32, chunk1 with size 32 and chunk2 with size 512



Exploit –libc leak

- 세 번의 allocation 을 가정하자. alloc(32-8), alloc(32-8), alloc(512-8)

- Then there exist chunk0 with size 32, chunk1 with size 32 and chunk2 with size 512

high

0x?? (x – 0x10 bytes)
size : x | PREV_INUSE (top size)
prev_size : 0x0
0x0 (0x1f0 bytes)
size : 0x200 | PREV_INUSE
prev_size : 0x0
0x0 (0x10 bytes)
size : 0x40 (overwritten)
prev_size : 0x0
0x0 (0x10 bytes)
size : 0x20 | PREV_INUSE
prev_size : 0x0

low

⇐ top chunk (free)

⇐ chunk2 (alloc)

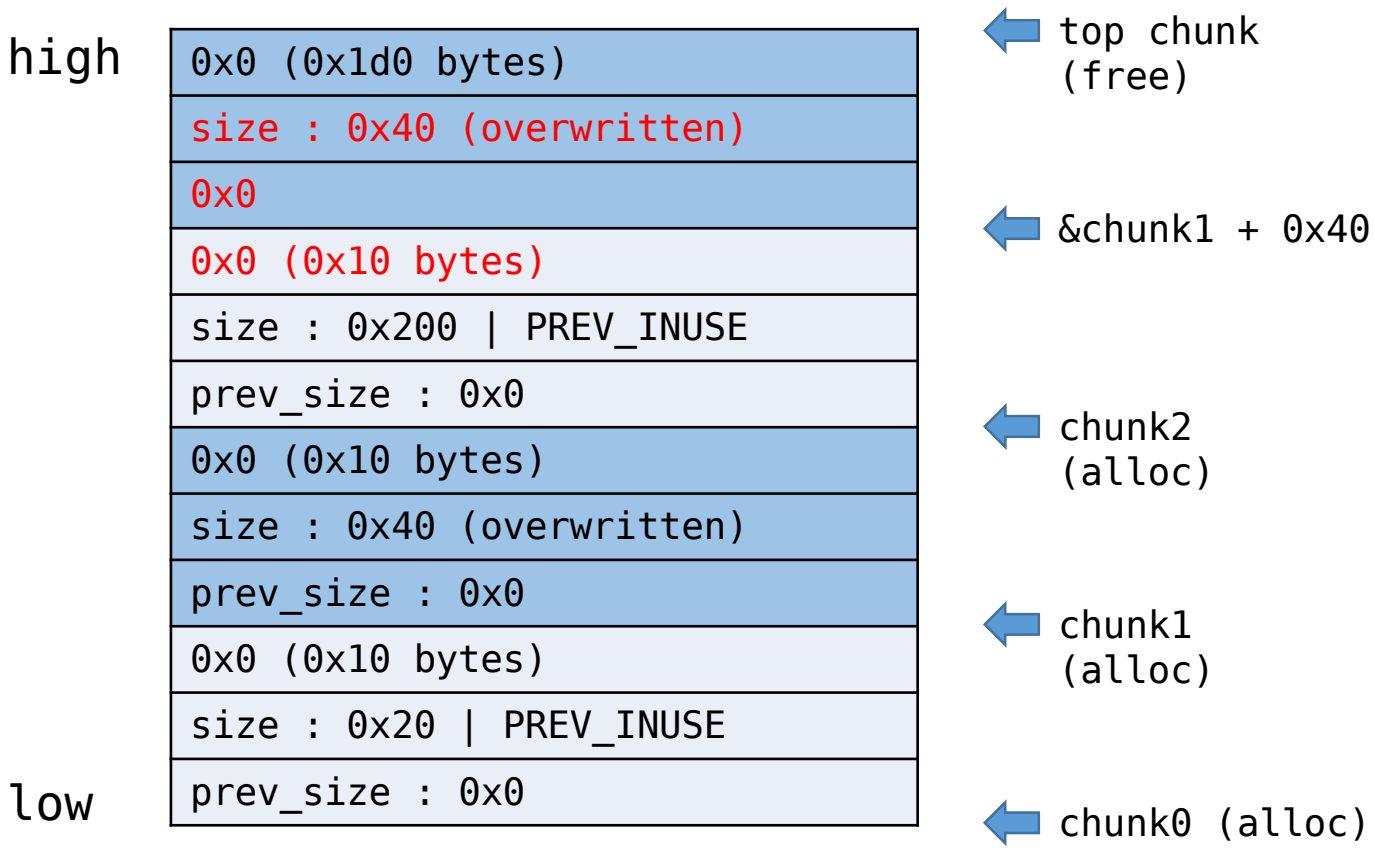
⇐ chunk1 (alloc)

⇐ chunk0 (alloc)

2. *"free(): invalid next size (fast)"* 에러 메세지를 출력 하는 security checking을 우회하기 위해 next chunk의 (measured by &chunk1 + 0x40) size field 가 적당한 value를 가지도록 index 2 block을 대상으로 fill 명령어를 수행하자.

→ (&chunk1 + 0x40)->size 가 0x40 이 되도록

Exploit –libc leak



Exploit –libc leak

high

0x0 (0x1d0 bytes)
size : 0x40 (overwritten)
0x0
0x0 (0x10 bytes)
size : 0x200 | PREV_INUSE
prev_size : 0x0
0x0 (0x10 bytes)
size : 0x40 (overwritten)
prev_size : 0x0
0x0 (0x10 bytes)
size : 0x20 | PREV_INUSE
prev_size : 0x0

low

top chunk (free)

&chunk1 + 0x40

chunk2 (alloc)

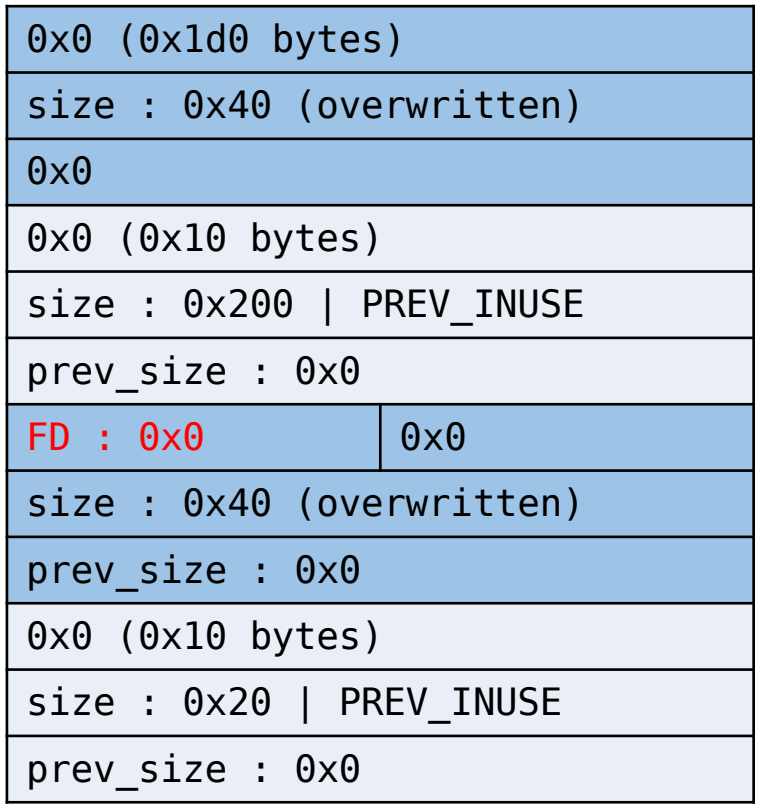
chunk1 (alloc)

chunk0 (alloc)

3. index 1 block을 대상으로 free 명령어를 수행하면 chunk1에 대한 free()가 호출되며 이는 fastbin에 반환된다.

Exploit –libc leak

high



low

top chunk
(free)

&chunk1 + 0x40

chunk2
(alloc)

chunk1
(free)

chunk0 (alloc)

since chunk1.size is 0x40,
fastbinsY[2] -> chunk1 -> 0

(fastbinsY[2] is for chunks
with size : 0x40)

Exploit –libc leak

high

0x0 (0x1d0 bytes)
size : 0x40 (overwritten)
0x0
0x0 (0x10 bytes)
size : 0x200 | PREV_INUSE
prev_size : 0x0
FD : 0x0 | 0x0
size : 0x40 (overwritten)
prev_size : 0x0
0x0 (0x10 bytes)
size : 0x20 | PREV_INUSE
prev_size : 0x0

low

top chunk (free)

&chunk1 + 0x40

chunk2 (alloc)

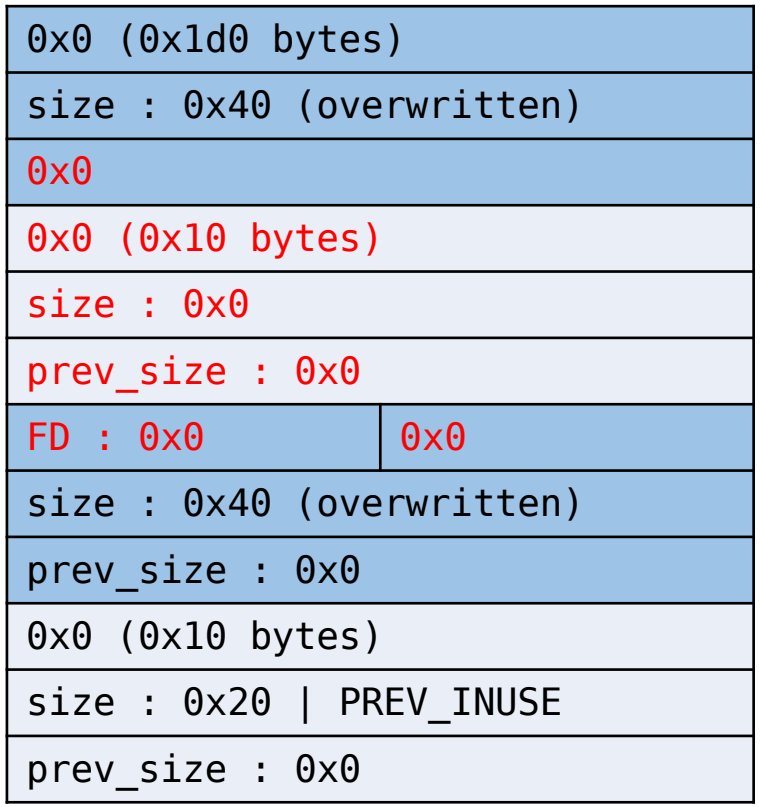
chunk1 (free)

chunk0 (alloc)

4. (3) 이후에 size 를 56으로 하는 allocate 명령어를 수행하게 되면 다시 index 1 block에 chunk1 이 주어진다. calloc이 호출되므로 &chunk1+0x10 부터 next chunk의 prev_size 에 대응되는 영역까지 0으로 초기화 시킨다.

Exploit –libc leak

high



low

top chunk
(free)

&chunk1 + 0x40

chunk2
(alloc)

chunk1
(alloc)

chunk0 (alloc)

이전과 다르게 block1.size 값이 56이므로 index 1 block에 대한 dump 명령어 수행 시 chunk2 의 앞부분을 leak 할 수 있다.

Exploit –libc leak

high

0x0 (0x1d0 bytes)
size : 0x40 (overwritten)
0x0
0x0 (0x10 bytes)
size : 0x0
prev_size : 0x0
FD : 0x0 | 0x0
size : 0x40 (overwritten)
prev_size : 0x0
0x0 (0x10 bytes)
size : 0x20 | PREV_INUSE
prev_size : 0x0

low

top chunk (free)

&chunk1 + 0x40

chunk2 (alloc)

chunk1 (alloc)

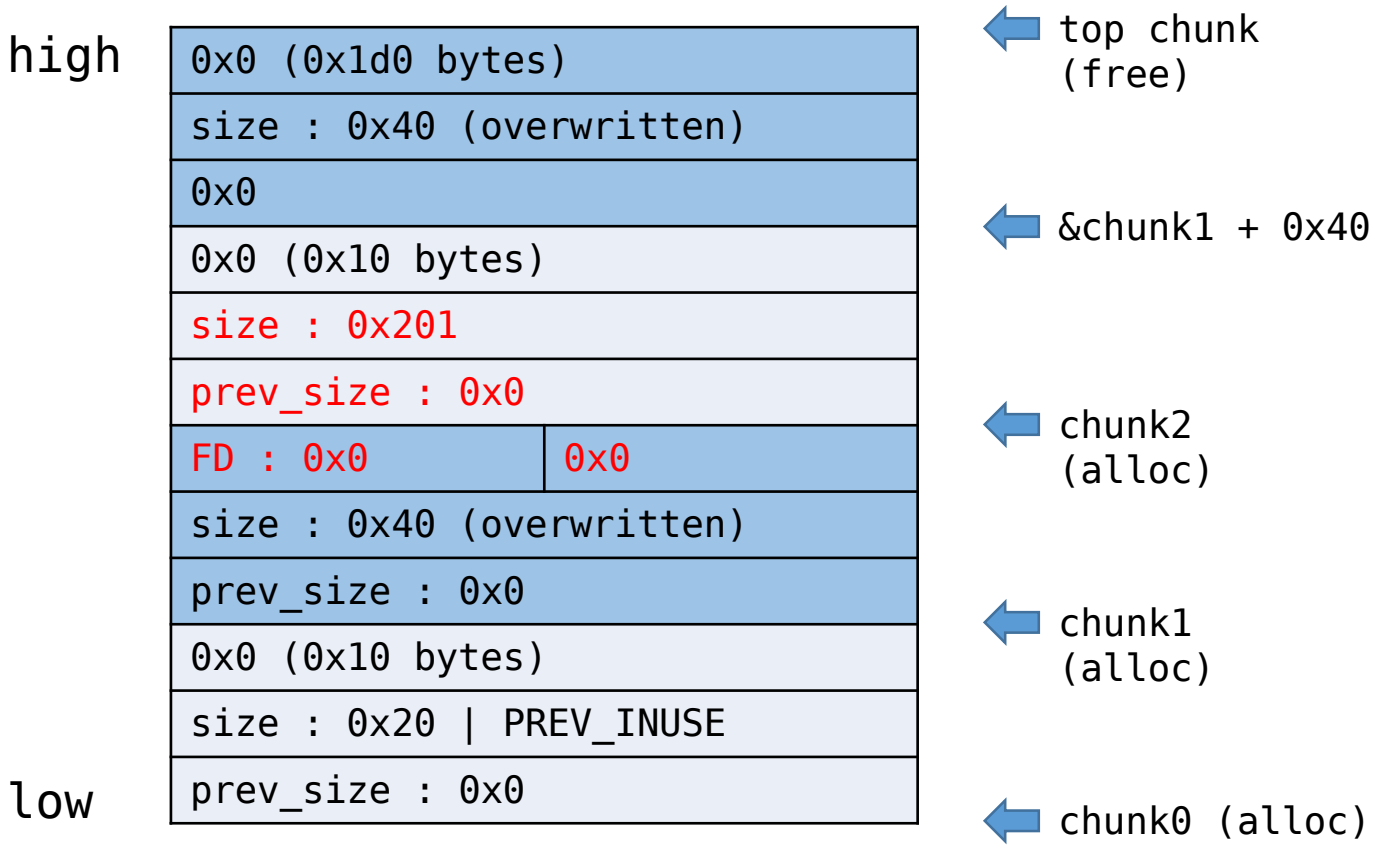
chunk0 (alloc)

5. (4) 이후에 바로 chunk2 에 대한 free() 를 시도하게 되면 *"free(): invalid pointer"* 에러 메세지를 만난다. chunk2.size 값이 0이고 (unsigned)&chunk2 이 0보다 크기 때문이다.

따라서 index 1 block 에 대해 fill 명령어를 수행하여 chunk2.size 를 0x201 으로 덮어주자.

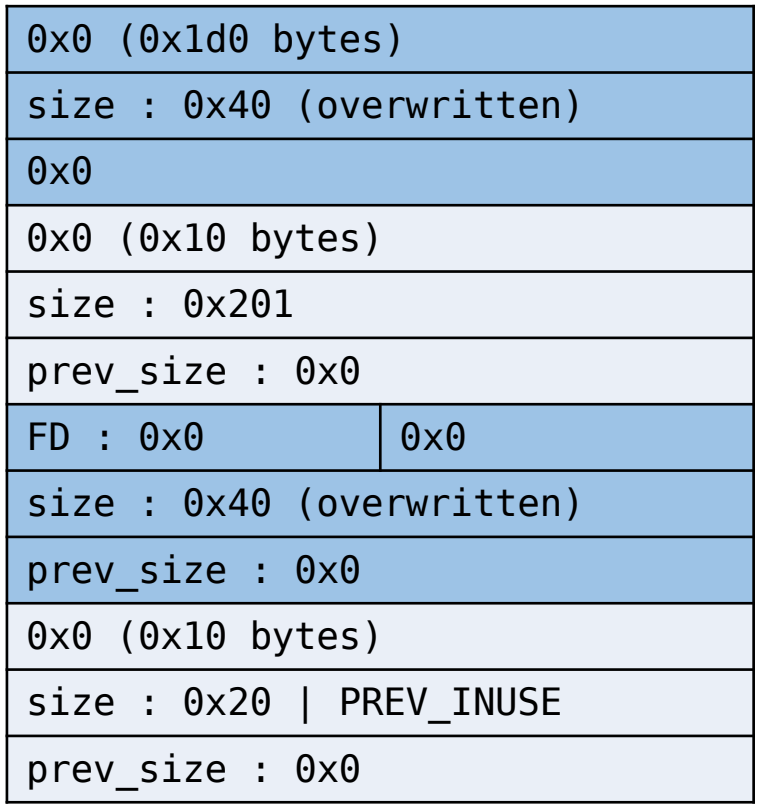
PREV_INUSE bit 를 세팅해주는 이유는 chunk2에 대해 free()를 호출할 때 chunk1 과의 병합을 막기 위해서이다.

Exploit –libc leak



Exploit –libc leak

high



low

top chunk (free)

&chunk1 + 0x40

chunk2 (alloc)

chunk1 (alloc)

chunk0 (alloc)

6. chunk2 는 top chunk와 인접한 large chunk 이기 때문에 chunk2에 대한 free() 호출 시에 top chunk와 병합된다.

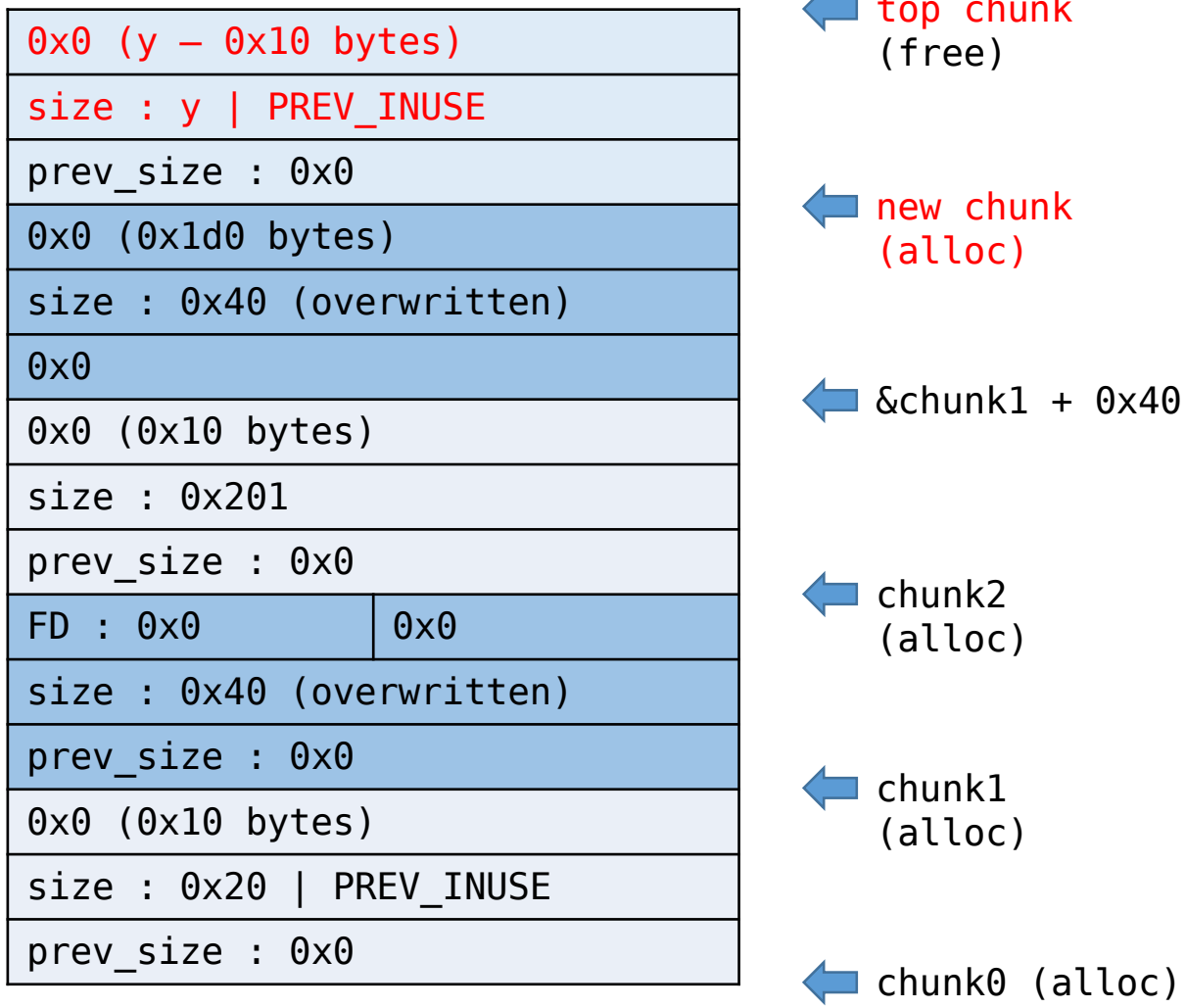
우리의 목적은 main_arena 를 leak 하여 libcbase를 알아내는 것이기 때문에 chunk2가 병합되지 않고 unsorted bin 에 들어가도록 해야 한다.

따라서 allocate를 한번 더 수행하여 chunk2 와 top chunk 사이에 다른 할당된 메모리 공간을 만들자

Exploit –libc leak

high

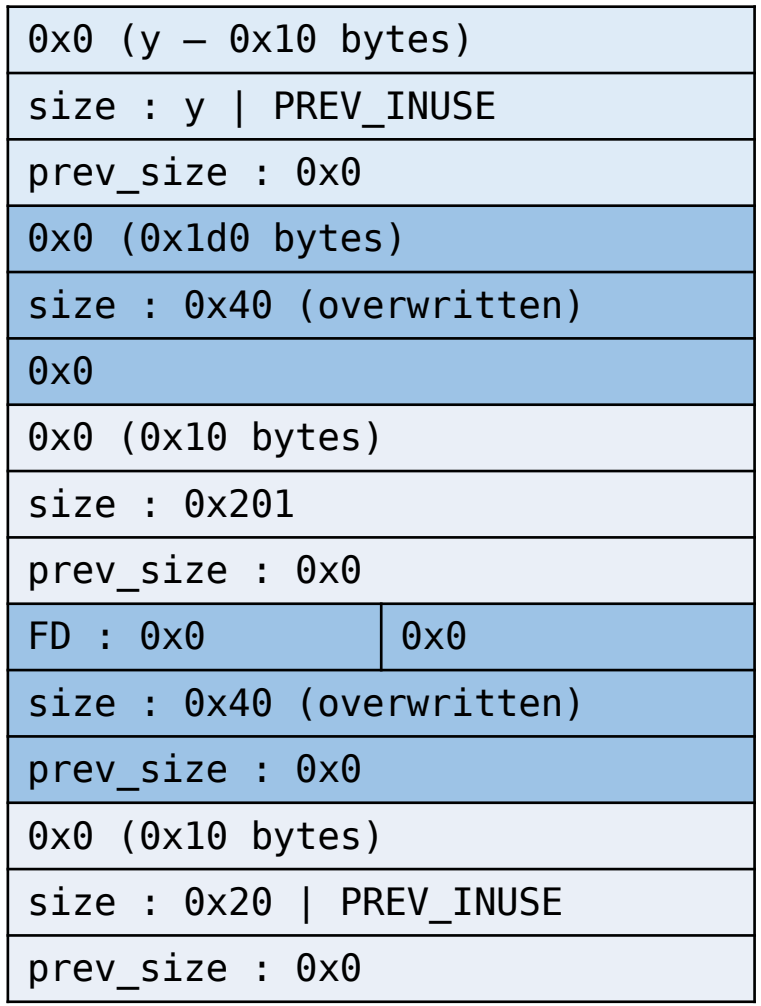
low



Exploit –libc leak

high

low



top chunk (free)

new chunk (alloc)

&chunk1 + 0x40

chunk2 (alloc)

chunk1 (alloc)

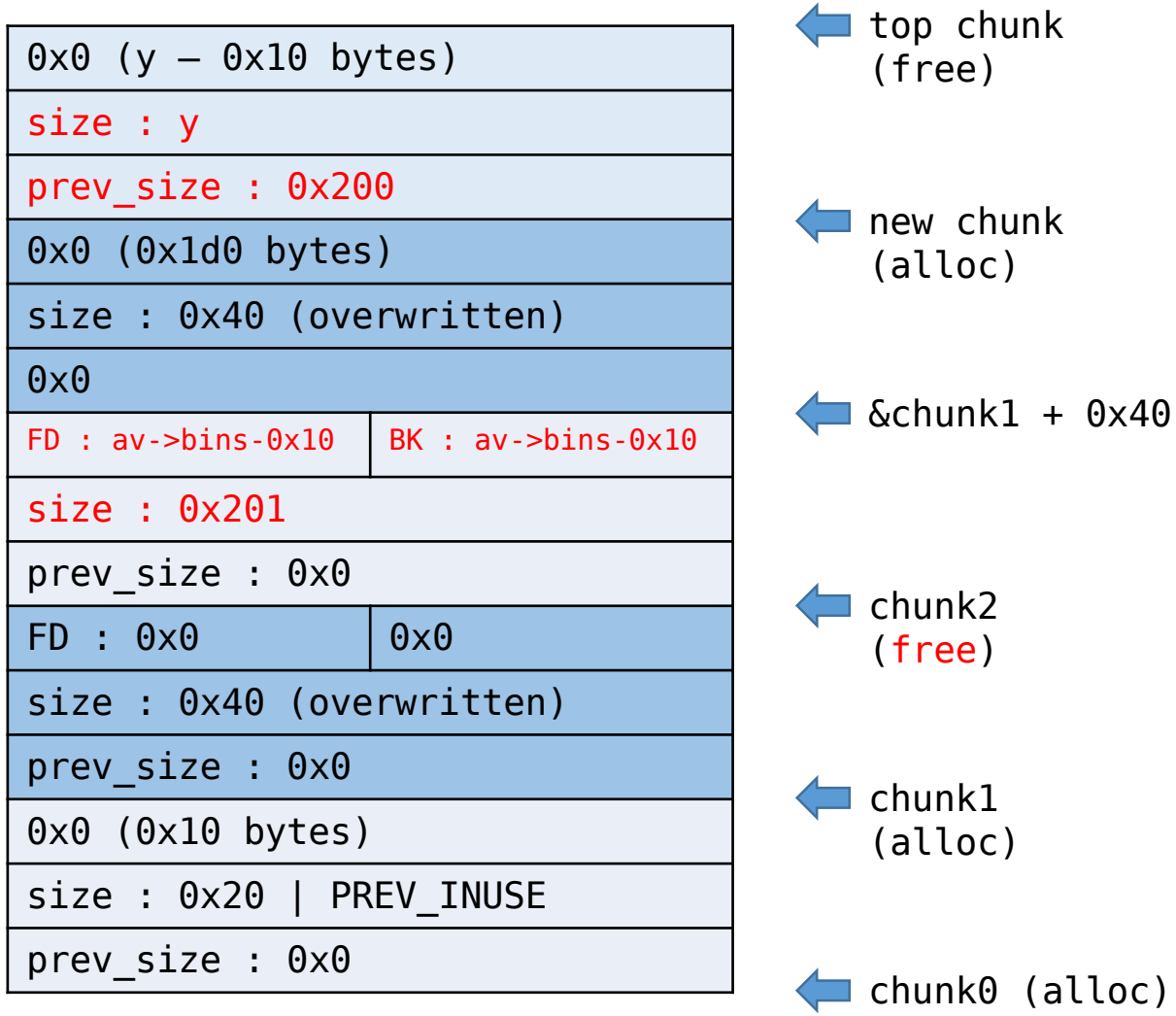
chunk0 (alloc)

7. 이제 chunk2에 대한 free()를 호출하면 chunk2 는 unsorted bin 에 연결되고 freed 상태의 chunk2 는 (unsorted bin 의 주소 – 0x10) 값을 .fd, .bk field에 저장한다.

Exploit –libc leak

high

low



Exploit –libc leak

high

0x0 (y – 0x10 bytes)
size : y
prev_size : 0x200
0x0 (0x1d0 bytes)
size : 0x40 (overwritten)
0x0
FD : av->bins-0x10 | BK : av->bins-0x10
size : 0x201
prev_size : 0x0
FD : 0x0 | 0x0
size : 0x40 (overwritten)
prev_size : 0x0
0x0 (0x10 bytes)
size : 0x20 | PREV_INUSE
prev_size : 0x0

low

⬅ top chunk (free)

⬅ new chunk (alloc)

⬅ &chunk1 + 0x40

⬅ chunk2 (free)

⬅ chunk1 (alloc)

⬅ chunk0 (alloc)

8. 이제 index 1 block 에 대해서 dump 명령어를 수행하면 &chunk1 + 0x10 부터 56 bytes 를 stdout에 출력해준다.

dumped :
p64(0) + p64(0) + p64(0) + p64(0x201) + p64(chunk1.fd) + p64(chunk1.bk) + p64(0)

이를 통해 우리는 main_arena 주소를 얻어냈고 오프셋을 이용해 libcbase를 구할 수 있다.

Exploit – overwriting the __malloc_hook variable

libc의 main_arena 구조체의 시작 주소로 부터 2 * SIZE_SZ bytes 떨어진 곳에는 __malloc_hook 함수 포인터를 위한 공간이 마련되어 있다.

user 가 malloc() 을 호출할 때 __malloc_hook 함수 포인터가 NULL 이 아니라면 해당 포인터가 가리키는 customized malloc 이 실행된다.
(calloc() 도 마찬가지로 __malloc_hook 변수를 먼저 확인하고 __malloc_hook()을 호출한 후 할당된 메모리에 대해서 memset() 을 수행한다.)

우리는 __malloc_hook 에 one-shot gadget 의 주소를 덮음으로써 다음 allocate 명령어가 수행될 때 쉘을 획득할 수 있다.

```
gdb-peda$ x/10gx (char*)&main_arena-0x10
0x7fc4fdbf4b10 <__malloc_hook>: 0x0000000000000000      0x0000000000000000
0x7fc4fdbf4b20 <main_arena>:    0x0000000000000000      0x0000000000000000
0x7fc4fdbf4b30 <main_arena+16>: 0x0000000000000000      0x00007ffff3cc4020
0x7fc4fdbf4b40 <main_arena+32>: 0x0000000000000000      0x0000000000000000
0x7fc4fdbf4b50 <main_arena+48>: 0x0000000000000000      0x0000000000000000
```

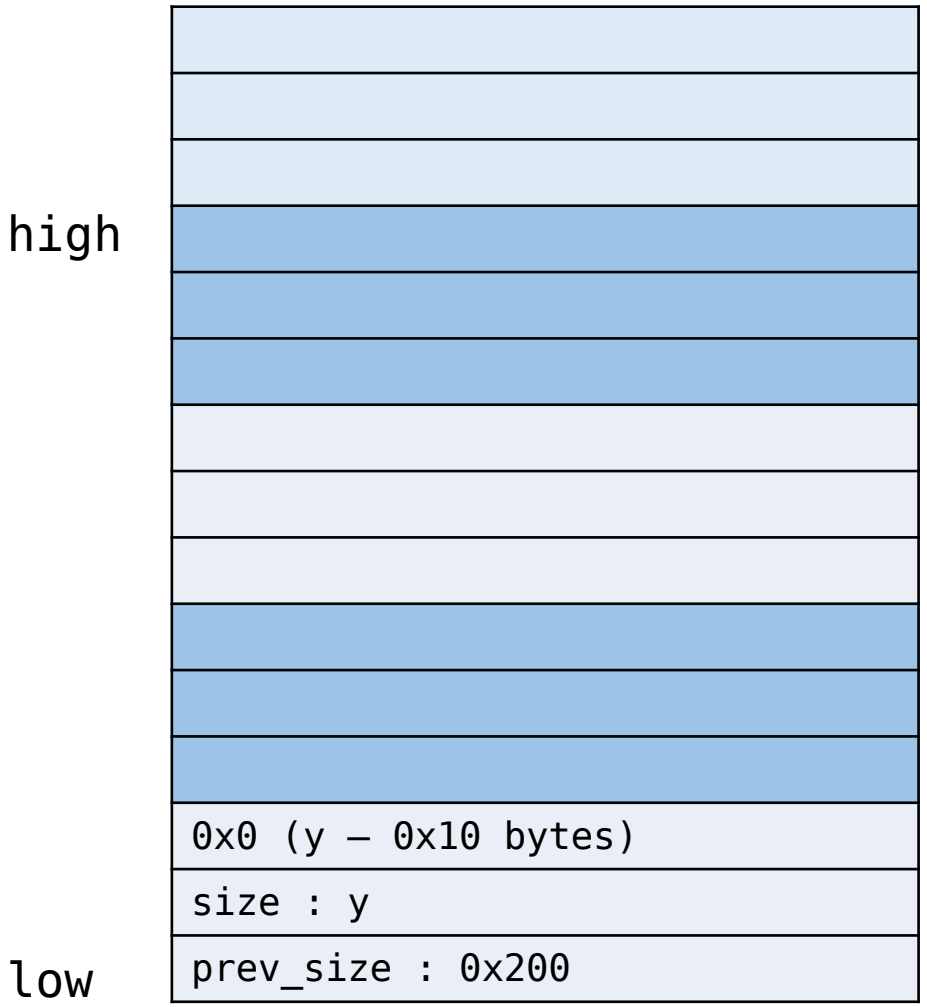
```
Void_t*
public_mALLOc(size_t bytes)
{
  mstate ar_ptr;
  Void_t *victim;

  __malloc_ptr_t (*hook) __MALLOC_P ((size_t, __const __malloc_ptr_t)) =
    __malloc_hook;
  if (hook != NULL)
    return (*hook)(bytes, RETURN_ADDRESS (0));
```

```
  if (hook != NULL) {
    sz = bytes;
    mem = (*hook)(sz, RETURN_ADDRESS (0));
    if(mem == 0)
      return 0;
#ifdef HAVE_MEMCPY
    return memset(mem, 0, sz);
```

in calloc()

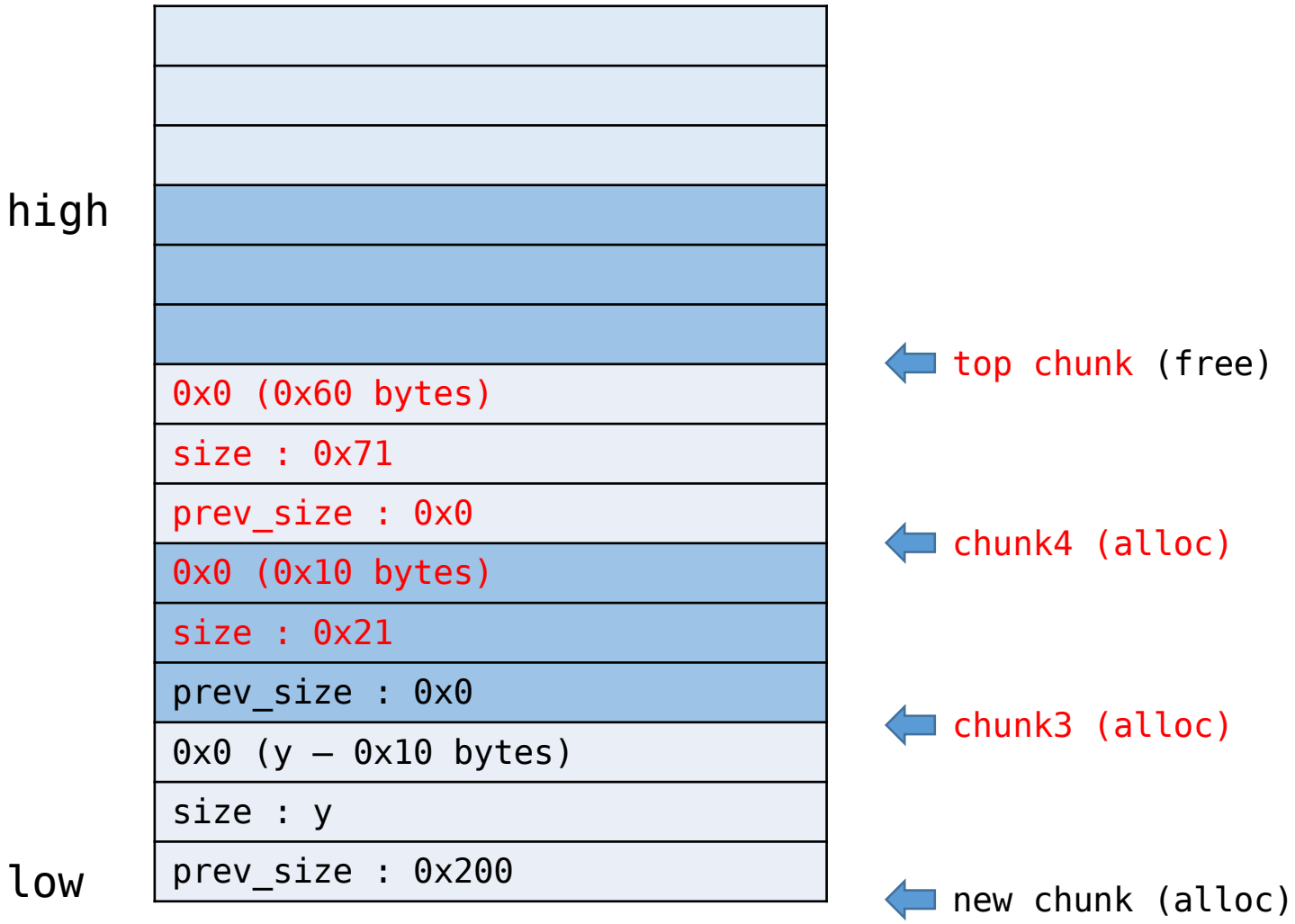
Exploit – overwriting the __malloc_hook variable



1. 이제 new chunk 부터 다시 위로 fast chunk를 쌓자. (libc leak에 사용되었던 chunk2는 unsorted bin 으로부터 미리 할당받자)

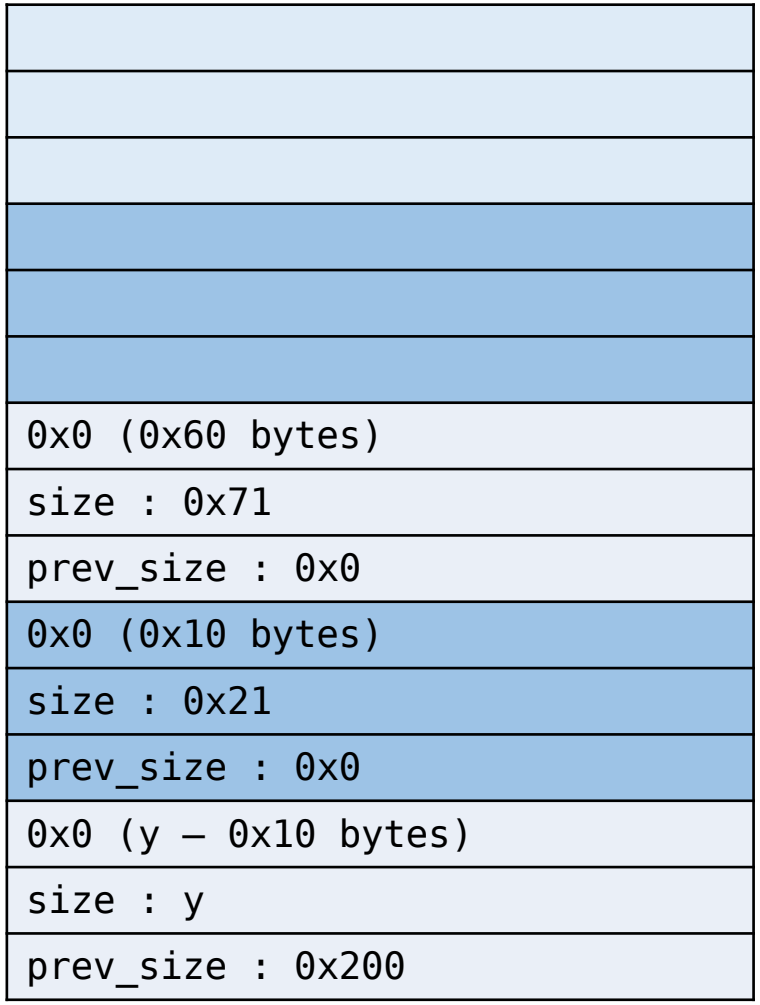
new chunk 할당 이후로 각각 0x20bytes, 0x70bytes 의 청크들을 top chunk로부터 분리하여 할당하자 (by allocate with the sizes 0x18, 0x68)

Exploit – overwriting the __malloc_hook variable



Exploit – overwriting the __malloc_hook variable

high



low

top chunk (free)

chunk4 (alloc)

chunk3 (alloc)

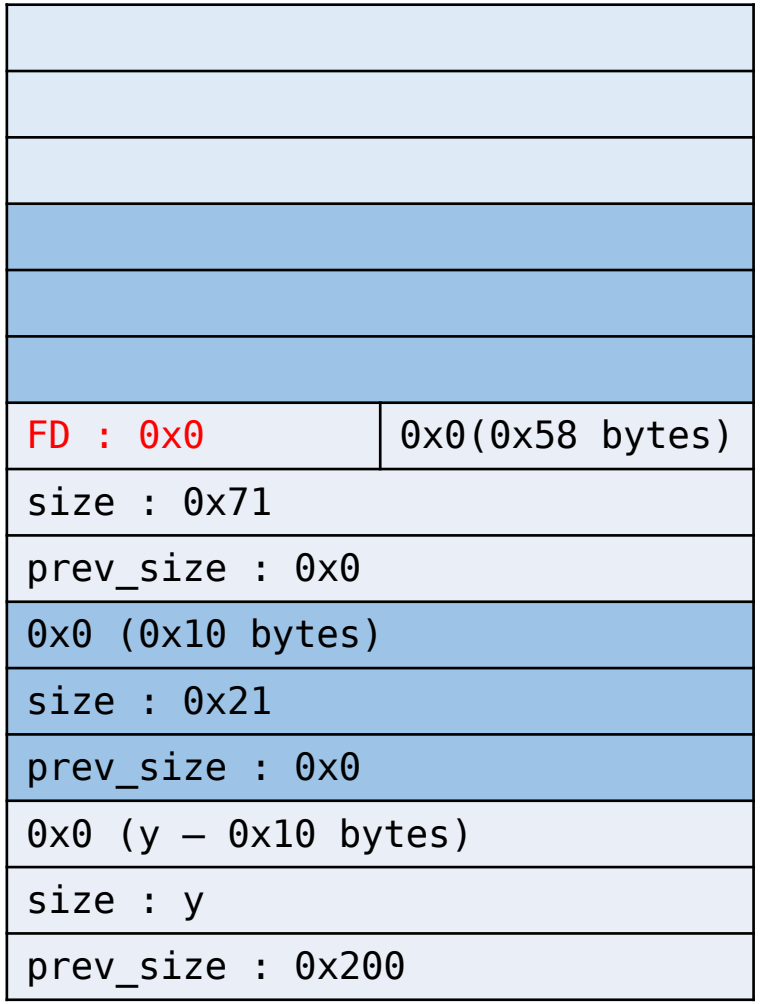
new chunk (alloc)

2. chunk4 에 대해서 free() 를 호출하자. (굳이 block structure 에 대한 얘기를 꺼낼 필요는 없을 것 같다.)

fastbin 배열에 아무런 chunk가 없으니 chunk4.fd 에는 0이 대입될 것 이다.

Exploit – overwriting the __malloc_hook variable

high



low

top chunk (free)

chunk4 (free)

chunk3 (alloc)

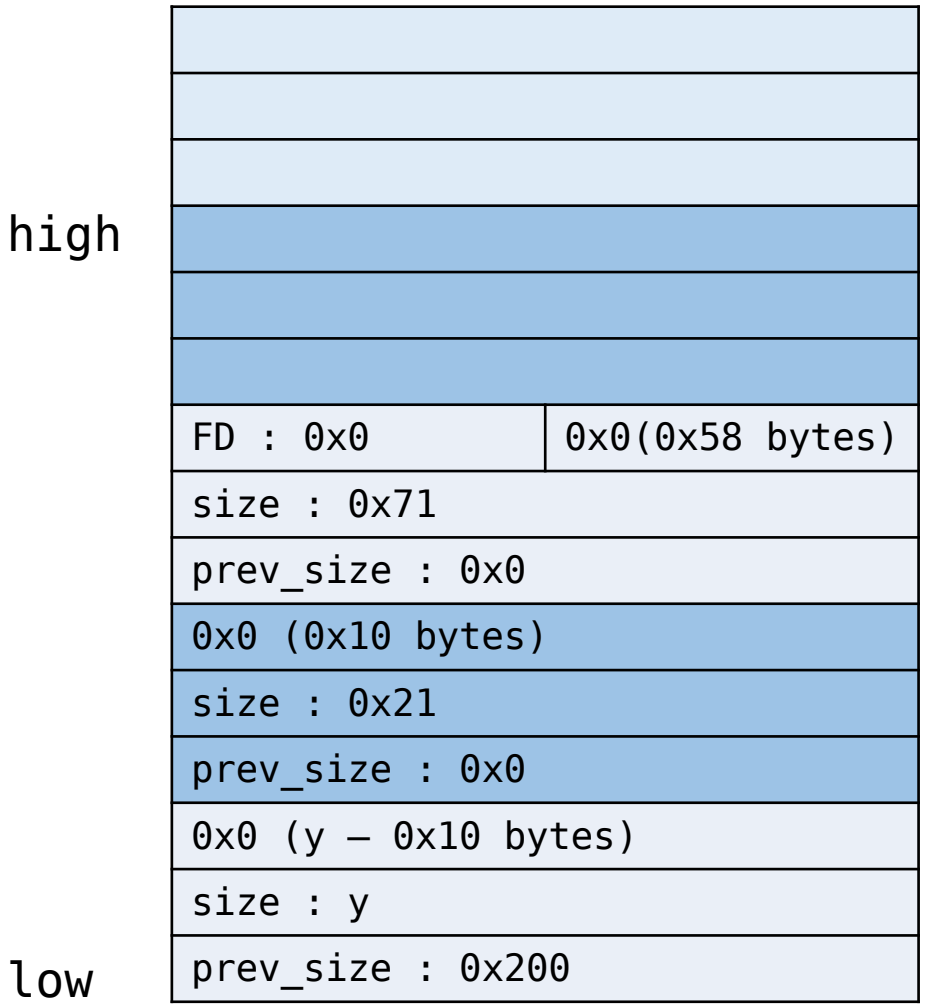
new chunk (alloc)

토막 상식

chunk4 는 fast chunk size 를 가지므로 해당 청크에 대한 free() 호출 시 인접한 chunk 와 병합되지 않는다.

또한 fast chunk free 시 다른 범주의 chunk들과 다르게 next chunk 의.prev_size field 와 PREV_IN_USE bit 의 setting 또한 이루어지지 않는다.

Exploit – overwriting the __malloc_hook variable



3. 우리의 원래 목적은 __malloc_hook 변수를 우리가 원하는 값으로 덮어씌우는 것이었다.
따라서 처음 떠오르는 sequenc는 다음과 같다.
a. chunk4 의 fd를 &__malloc_hook – 0x10 으로 덮고
b. chunk4 를 할당한다. 그러면 chunk4 를 가졌던 fastbin이 우리가 만든 fake chunk를 가리킨다.
c. fake chunk를 할당받아 __malloc_hook 변수를 one shot gadget으로 덮는다.

Exploit – overwriting the __malloc_hook variable

결과적으로 위의 시나리오로 진행을 하면 fastbin 으로부터 메모리를 할당할 때 실행되는 security checking 에 의해 다음과 같은 에러를 뱉는다.
"malloc(): memory corruption (fast)"

```
if (__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0))
  {
    errstr = "malloc(): memory corruption (fast)";
  errout:
    malloc_printerr (check_action, errstr, chunk2mem (victim));
    return NULL;
  }
```

왜냐? fake chunk의 .size field 값에 대한 fastbin_index() macro 반환 값이 fake chunk 가 들어있던 fastbin 배열의 index 값과 다르기 때문이다. 즉 fake chunk를 할당하기 위해서는 fake chunk 가 속한 fastbin 에서 기본적으로 처리하는 청크의 size 와 fake chunk 의 .size field 값과 같도록 조작을 해야 한다.

```
/* offset 2 to use otherwise unindexable first 2 bins */
#define fastbin_index(sz) \
  ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)
```

Exploit – overwriting the __malloc_hook variable

chunk4 의 fd를 &__malloc_hook – 0x10으로 덮는다고 가정하자. fake chunk의 할당을 시도하려고 하면 애초에 .size field 가 너무 큰 값이라 security checking 를 통과하지 못한다.

```
gdb-peda$ x/10gx (char*)&main_arena-0x40
0x7f1cc65f4ae0 <_IO_wide_data_0+288>:   0x0000000000000000      0x0000000000000000
0x7f1cc65f4af0 <_IO_wide_data_0+304>:   0x00007f1cc65f3260      0x0000000000000000
0x7f1cc65f4b00 <__memalign_hook>:       0x00007f1cc62b5e20      0x00007f1cc62b5a00
0x7f1cc65f4b10 <__malloc_hook>: 0x0000000000000000      0x0000000000000000
0x7f1cc65f4b20 <main_arena>:    0x0000000000000000      0x0000000000000000
```

그러나 오른쪽 해당 메모리 공간을 통으로 보면 '\x00' byte 들이 연속적으로 중첩되어 있는 것들을 볼 수 있다. 즉 fastbin range (0x20 ~ 0x80) 내의 값을 가지는 QWORD 를 size field 로 사용하여 fake chunk를 할당할 수 있다.

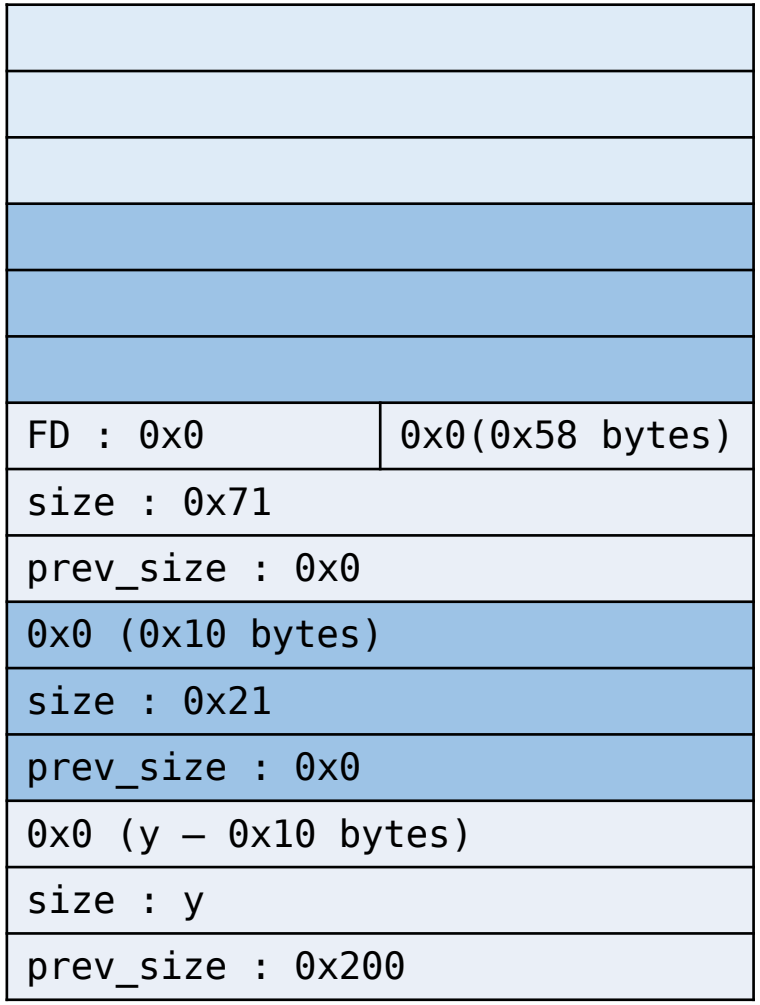
```
gdb-peda$ x/10gx (char*)&__malloc_hook-0x23
0x7f1cc65f4aed <_IO_wide_data_0+301>:   0x1cc65f3260000000      0x000000000000007f
0x7f1cc65f4afd: 0x1cc62b5e20000000      0x1cc62b5a0000007f
0x7f1cc65f4b0d <__realloc_hook+5>:      0x000000000000007f      0x0000000000000000
0x7f1cc65f4b1d: 0x0000000000000000      0x0000000000000000
0x7f1cc65f4b2d <main_arena+13>: 0x0000000000000000      0xffd612a020000000
```

앞에서 chunk4 의 size를 0x70으로 한 것도 fake chunk의 size field를 0x7f 로 설정할 수 있기 때문이다.

fastbin_index(0x70) = 5
fastbin_index(0x7f) = 5
이므로 이전의 security check 를 통과한다.
(size >> 4 를 수행한 후 비교하기 때문에 heap segment의 시작주소가 0x7f 가 아닌 0x5d 라든가 해도 대부분 적절한 fastbin 을 찾을 수 있다.)

Exploit – overwriting the __malloc_hook variable

high



low

top chunk (free)

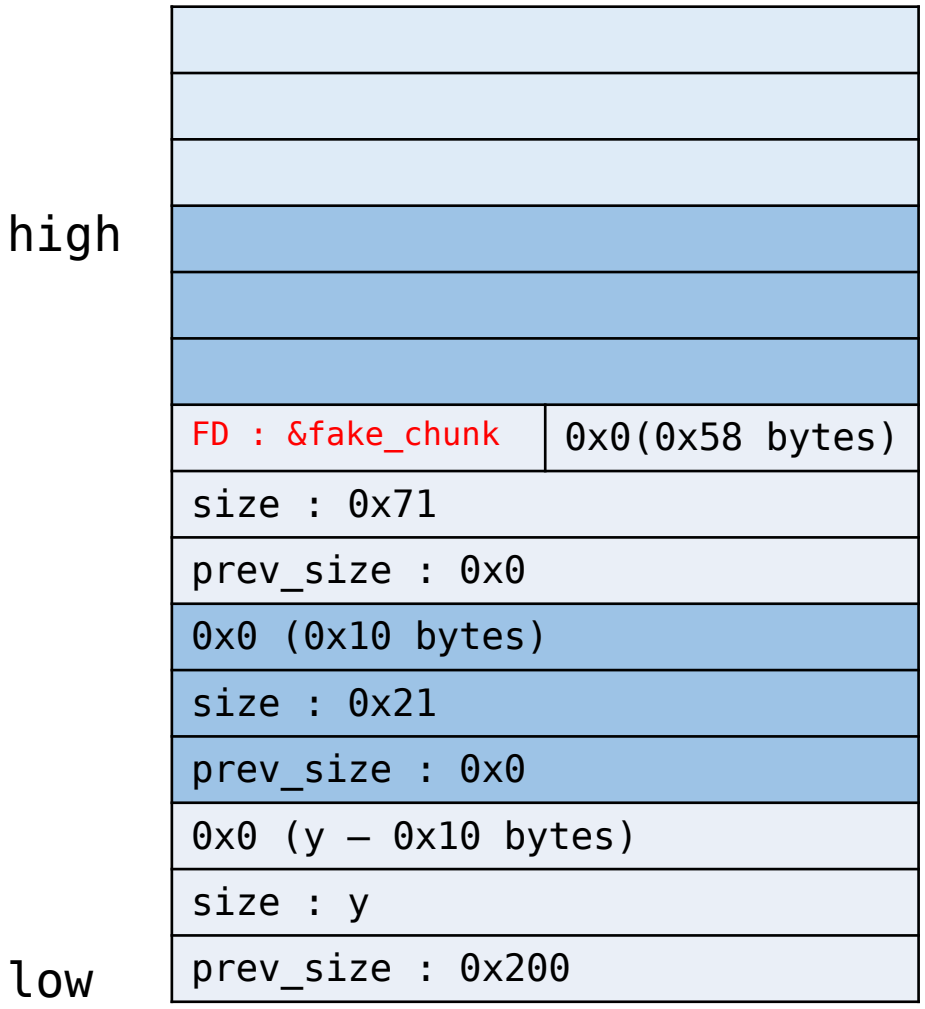
chunk4 (free)

chunk3 (alloc)

new chunk (alloc)

3’. chunk4.size >> 4 == fake_chunk.size >> 4 : true
가정하에 chunk4 의 fd를
&fake_chunk->size – 0x8 로 덮자.

Exploit – overwriting the __malloc_hook variable



top chunk (free)

chunk4 (free)

chunk3 (alloc)

new chunk (alloc)

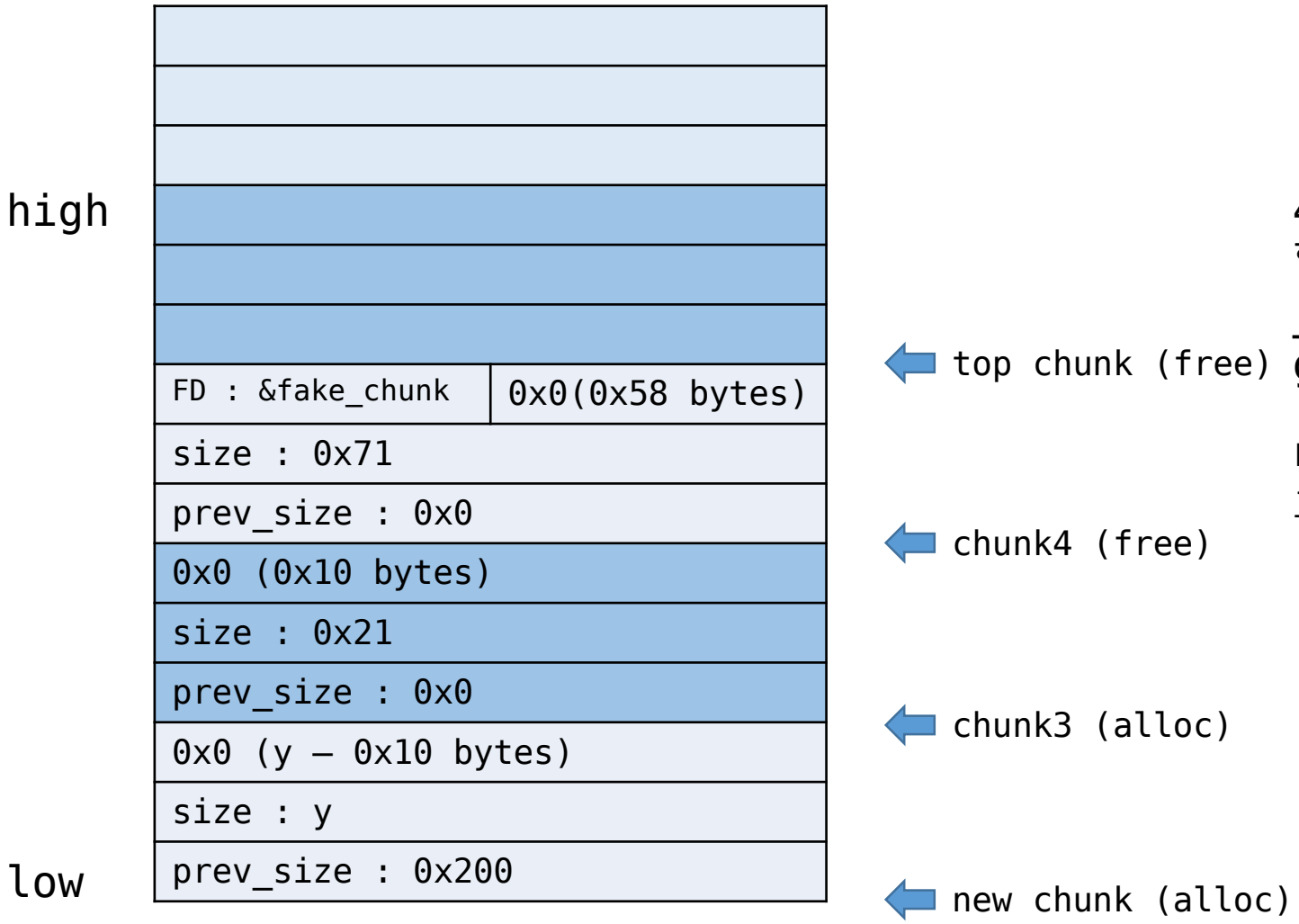
이 경우 chunk4 가 속한 fastbin 은 다음과 같이 나타낼 수 있다.

fastbin[fastbin_index(chunk4.size)] -> chunk4 -> fake_chunk -> ????

fake_chunk에 대한 메모리 뷰는 다음과 같다.

```
<fake_chunk+0> 0x????????
<fake_chunk+8> 0x0000007f
....
<__malloc_hook-α> ......
....
<__malloc_hook+β> ......
```

Exploit – overwriting the __malloc_hook variable



4. chunk4를 할당 받고 fake_chunk를 할당 받아 Fill operation을 통해 __malloc_hook 변수를 one shot gadget 의 주소로 덮으면 된다.

마지막으로 calloc() 을 한 번 더 호출해주면 shell 을 획득할 수 있다.

reference

one gadget : for installing the tools to find the offset of one gadgets

write-up1 : libc leak 할 때 '작은 size 의 chunk를 큰 size의 chunk 인 것 처럼 속인다는 것' 여기서 실마리를 얻었음

write-up2 : malloc_hook overwriting 이 어떻게 이루어지는지 참고

glibc malloc source : 여러 security checking 이 어떻게 이루어지는지 참고

ptmalloc2 malloc source : malloc / free algorithm 세부사항 확인

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)