Heap Exploitation

Modern Binary Exploitation

CSCI 4968 - Spring 2015

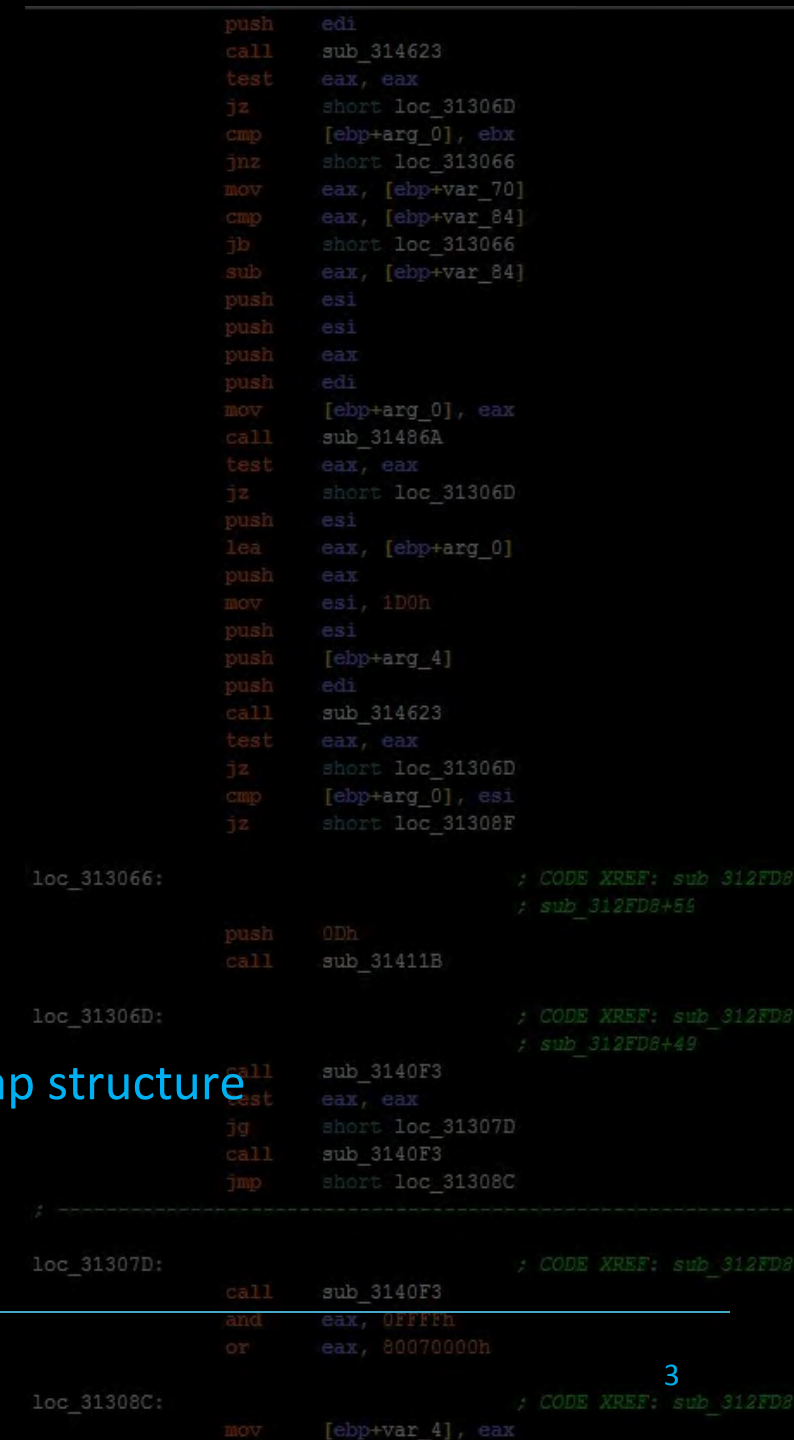
Markus Gaasedelen

Lecture Overview

- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

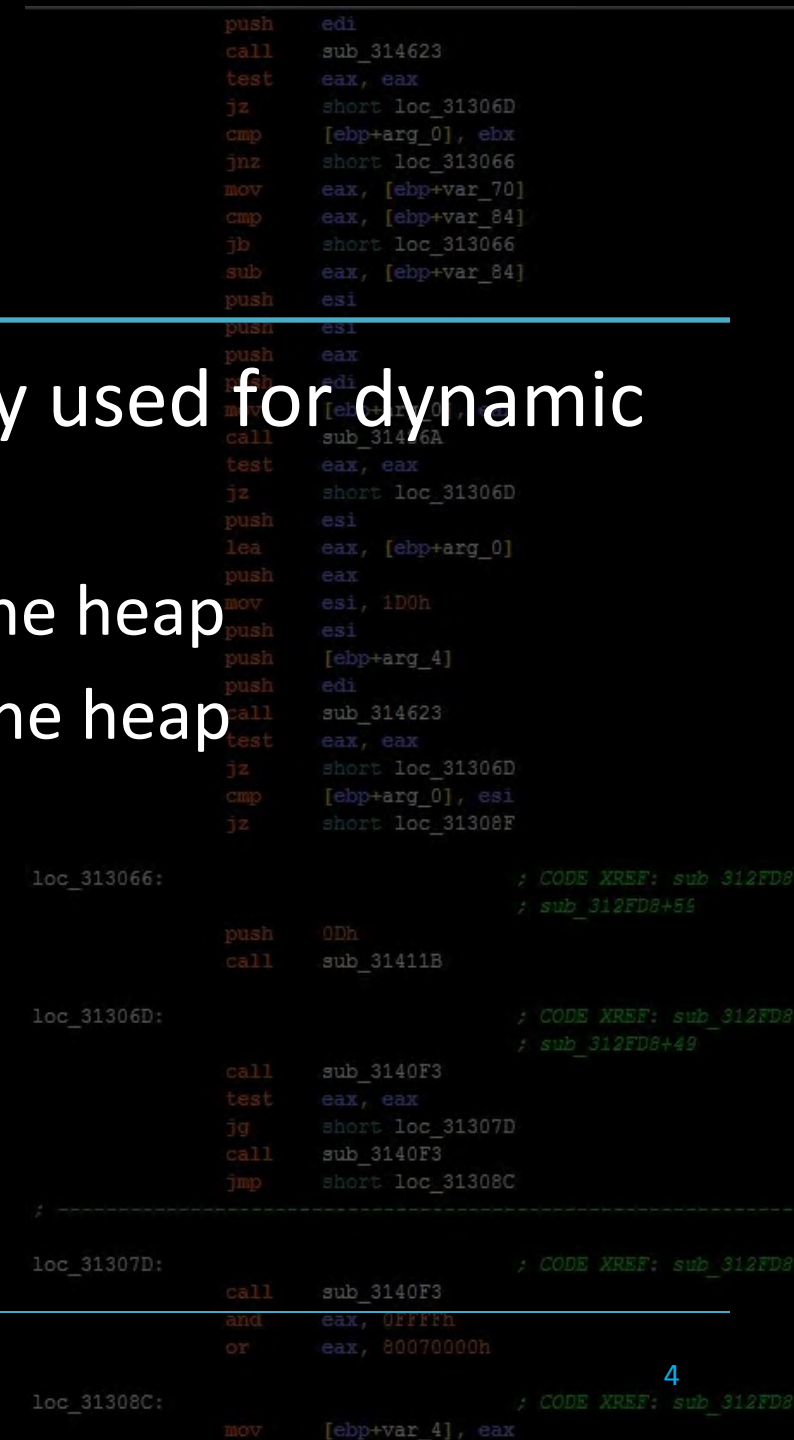
HEAP OVERVIEW

Basic overview on dynamic memory and heap structure

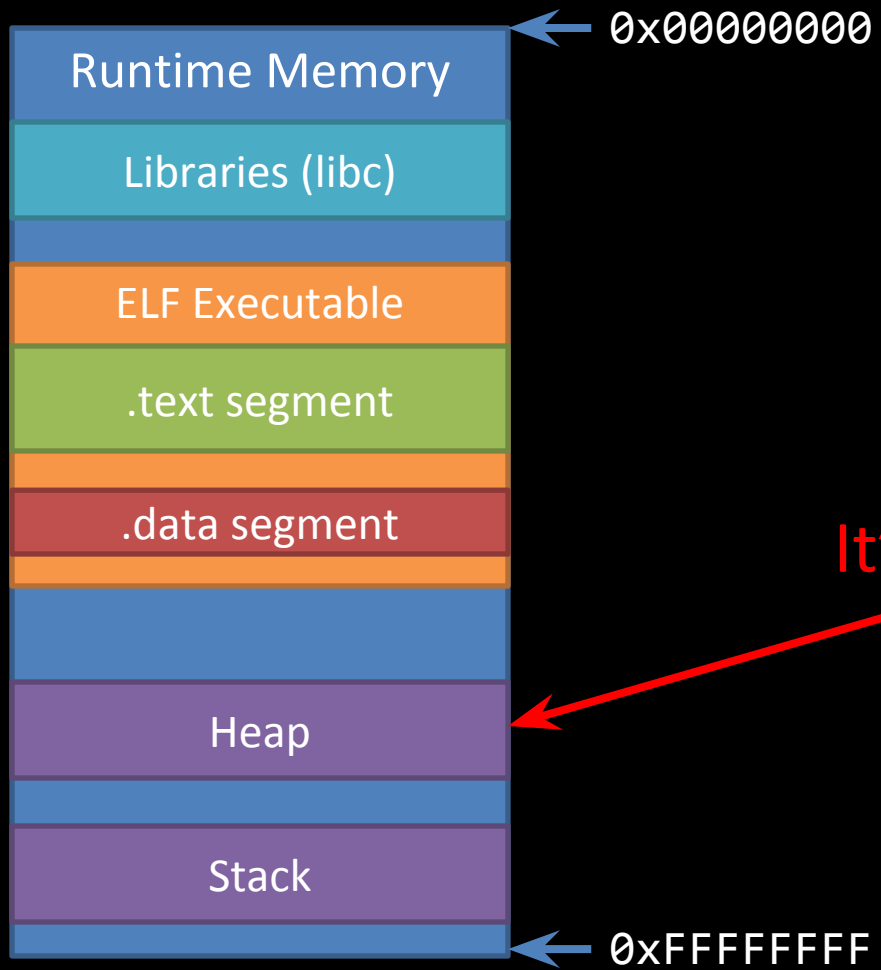


The Heap

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



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;
}
```

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

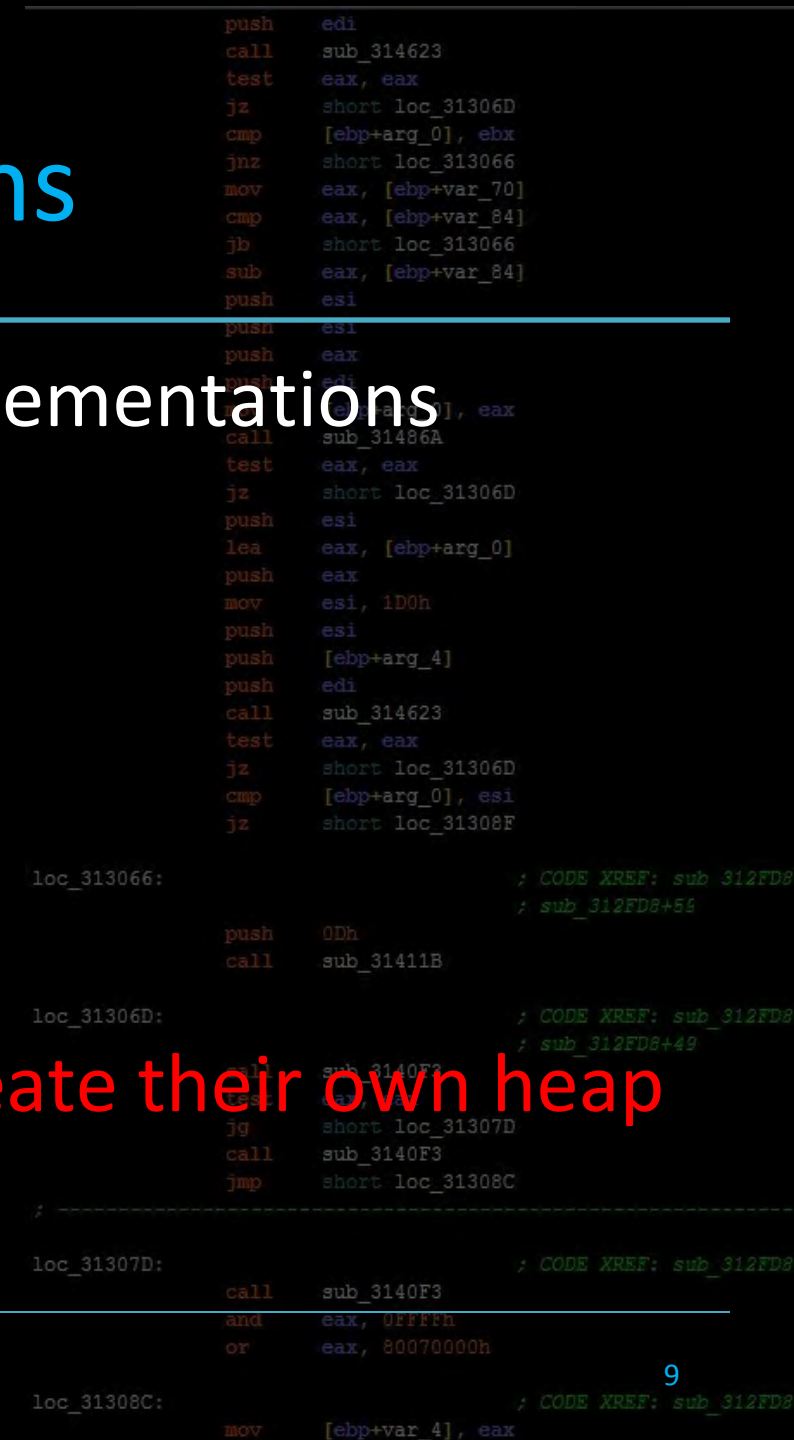
- Fixed memory allocations known at compile time
- Local variables, return addresses, function args
- Fast, Automatic
  - Done by the compiler
  - Abstracts away any concept of allocating/de-allocating

Heap Implementations

- Tons of different heap implementations
  - dlmalloc
  - ptmalloc
  - tcmalloc
  - jemalloc
  - nedmalloc
  - Hoard

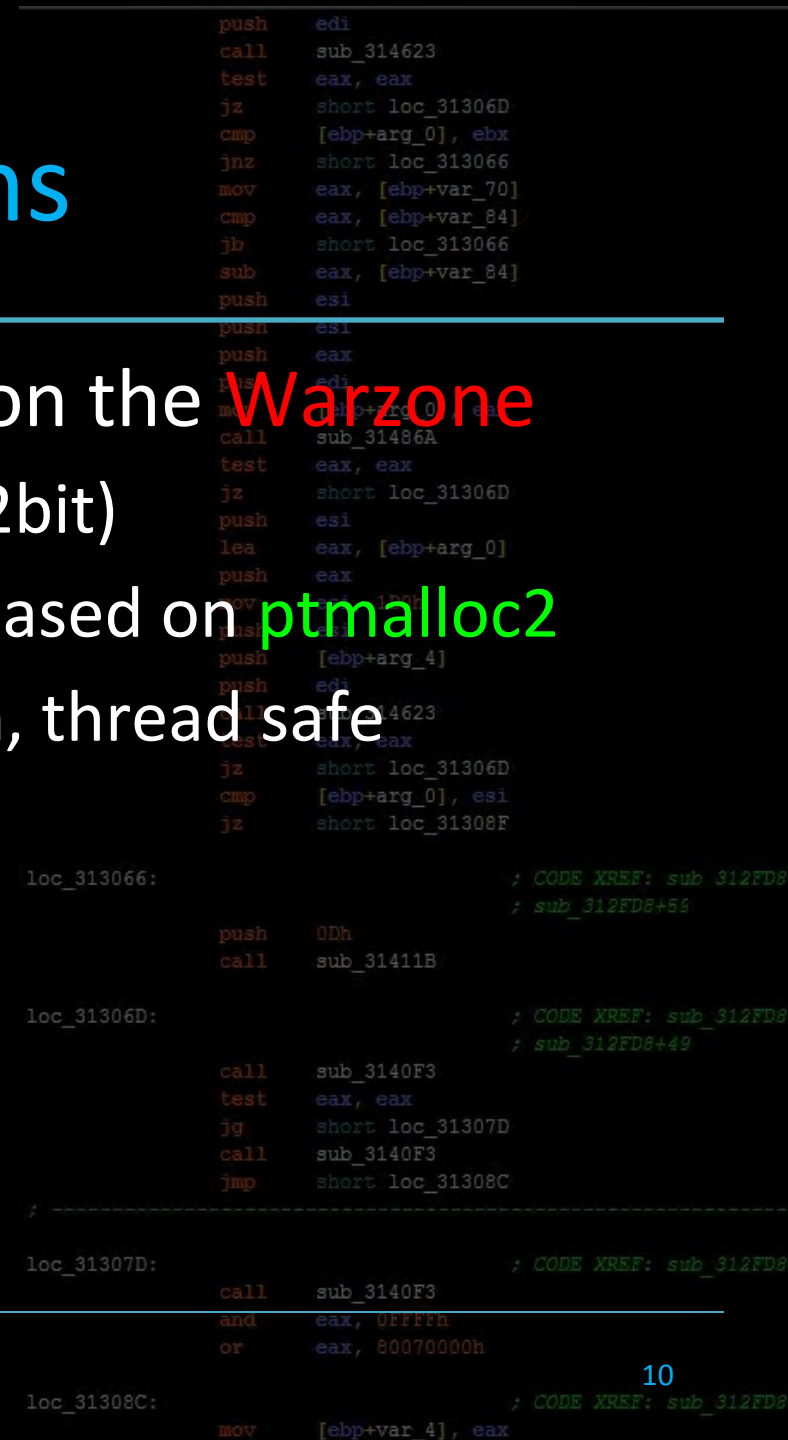
Heap Implementations

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



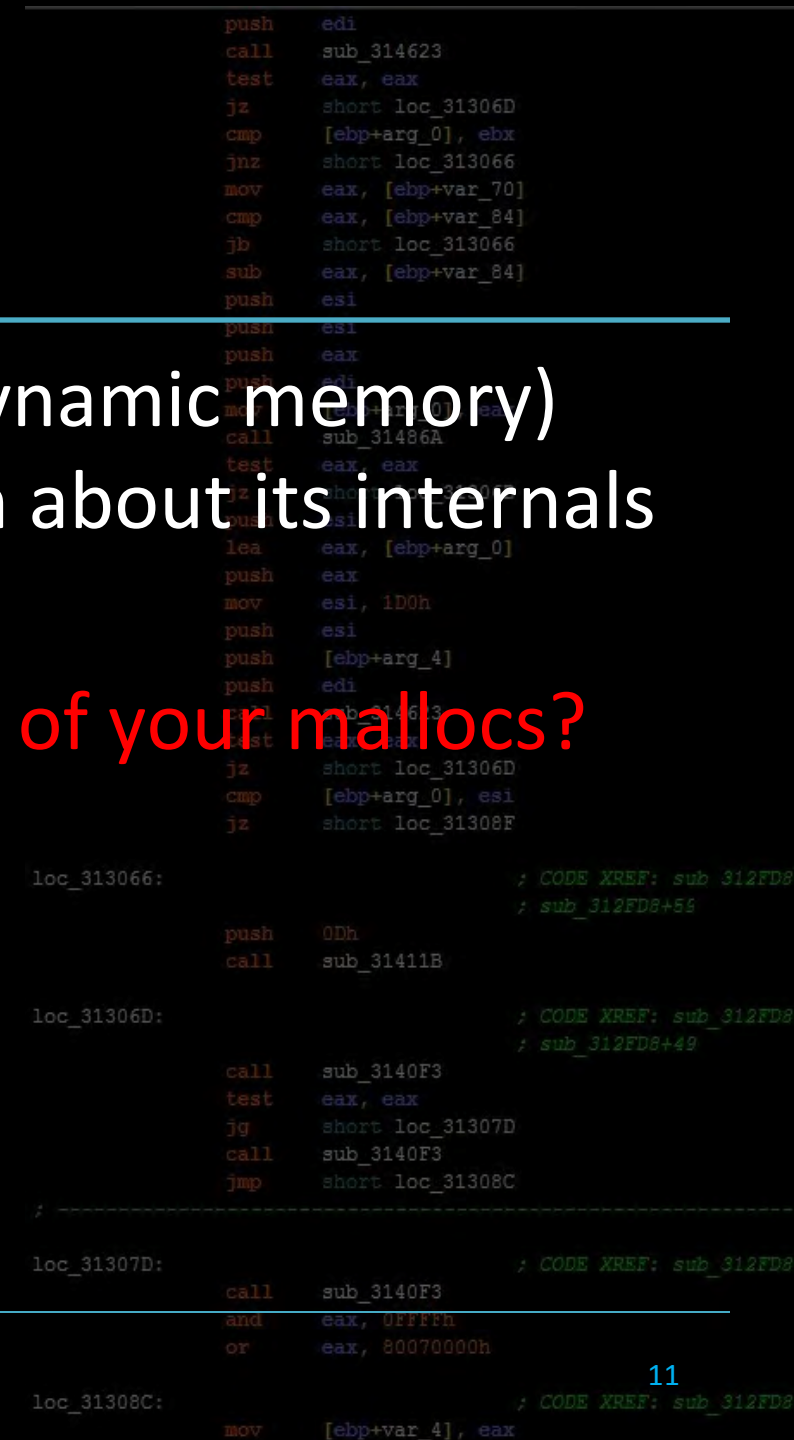
Heap Implementations

- glibc 2.19 is what we have on the Warzone
  - Default for Ubuntu 14.04 (32bit)
  - Its heap implementation is based on ptmalloc2
  - Very fast, low fragmentation, thread safe



Know Thy Heap

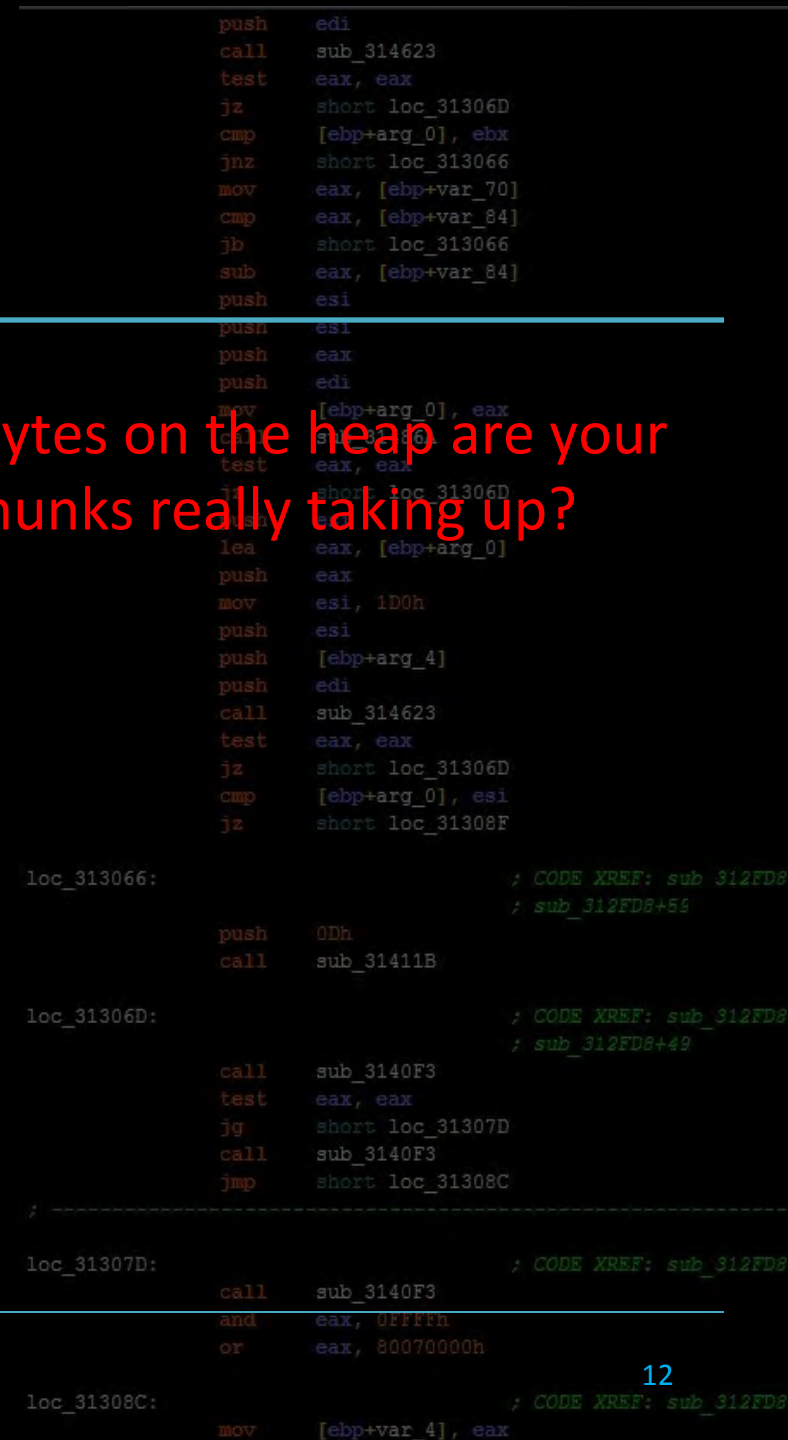
- Everyone uses the heap (dynamic memory) but few usually know much about its internals
- Do you even know the cost of your mallocs?



Malloc Trivia

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

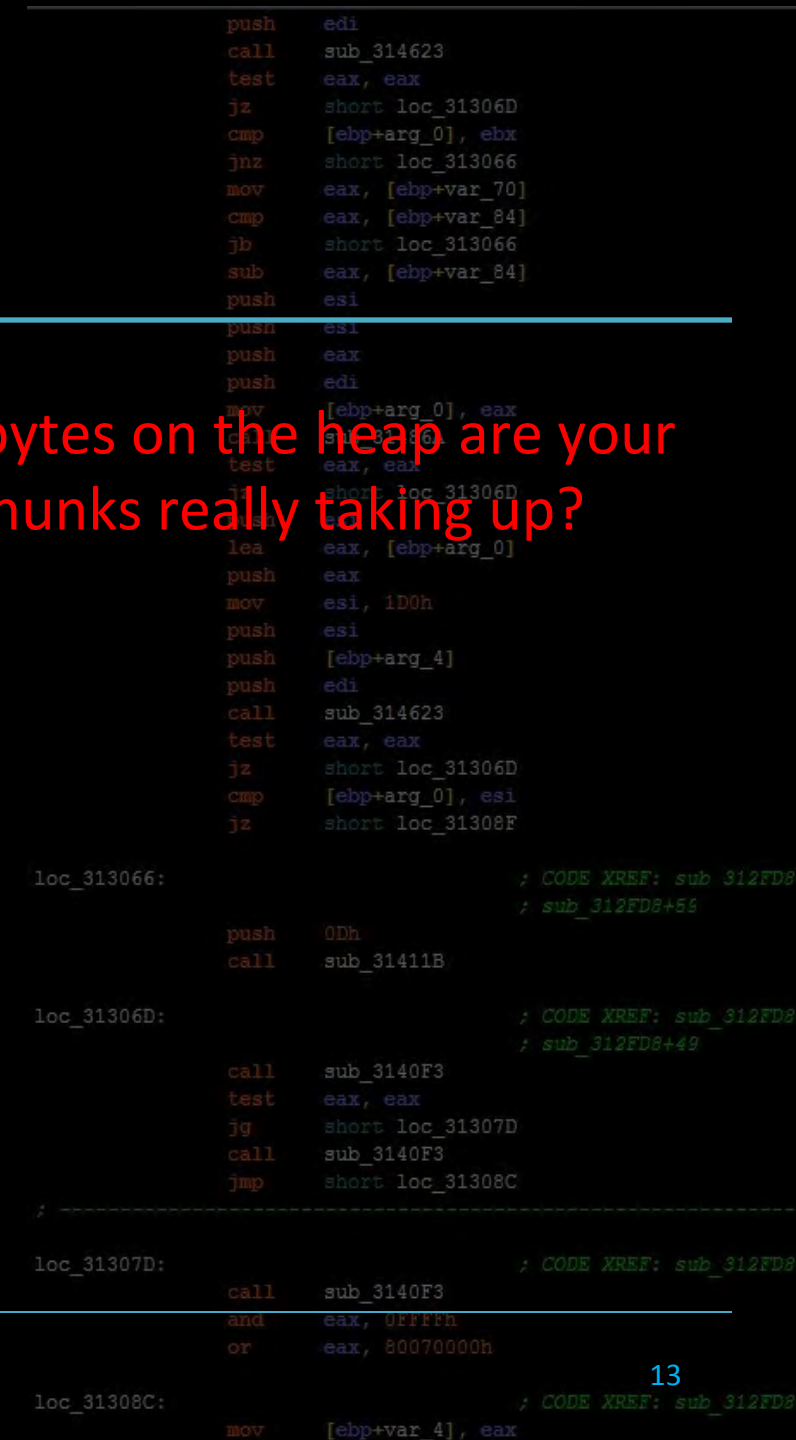
How many bytes on the heap are your malloc chunks really taking up?



Malloc Trivia

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

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

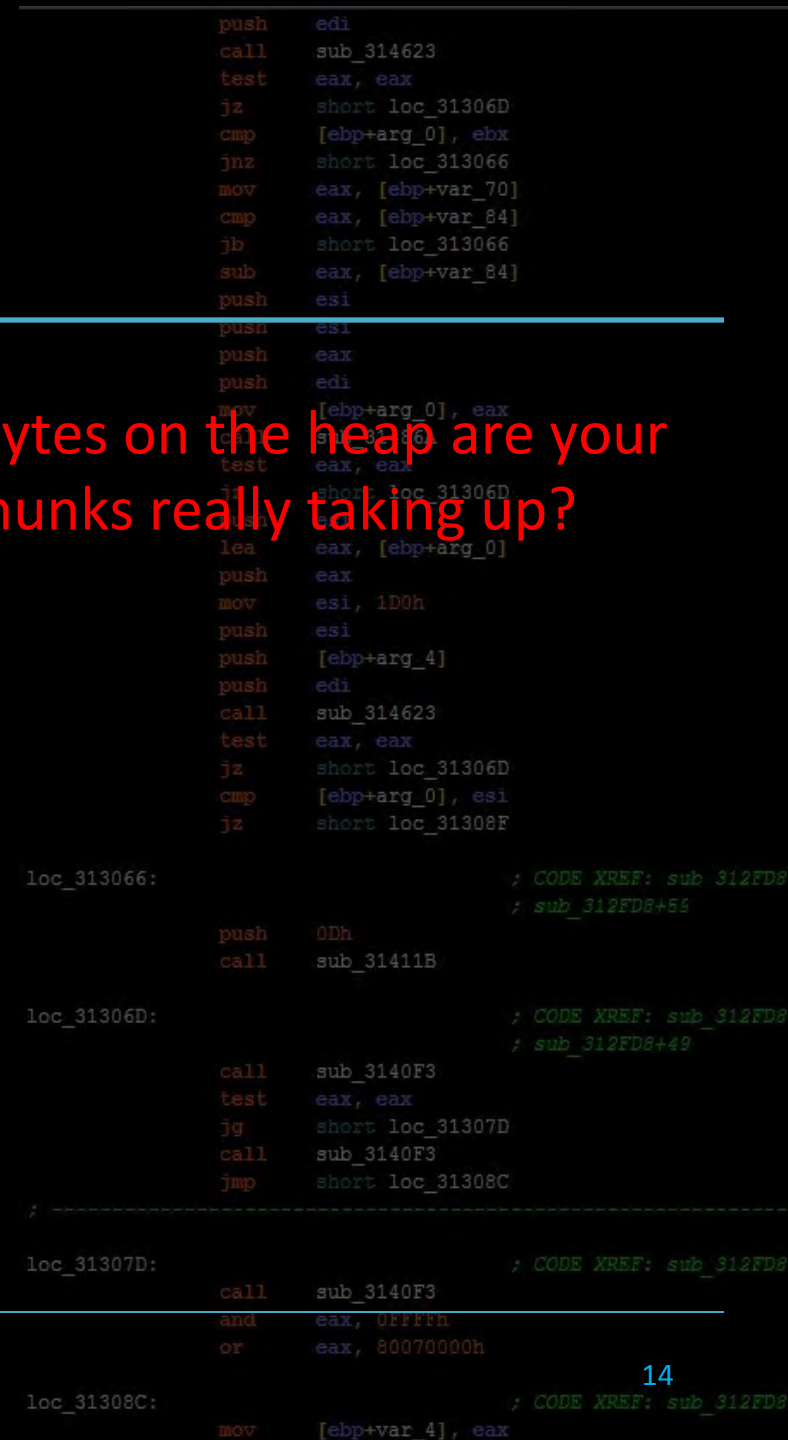


Malloc Trivia

- malloc(32);
  - 40 bytes
- malloc(4);
  - 16 bytes
- malloc(20);

- malloc(0);

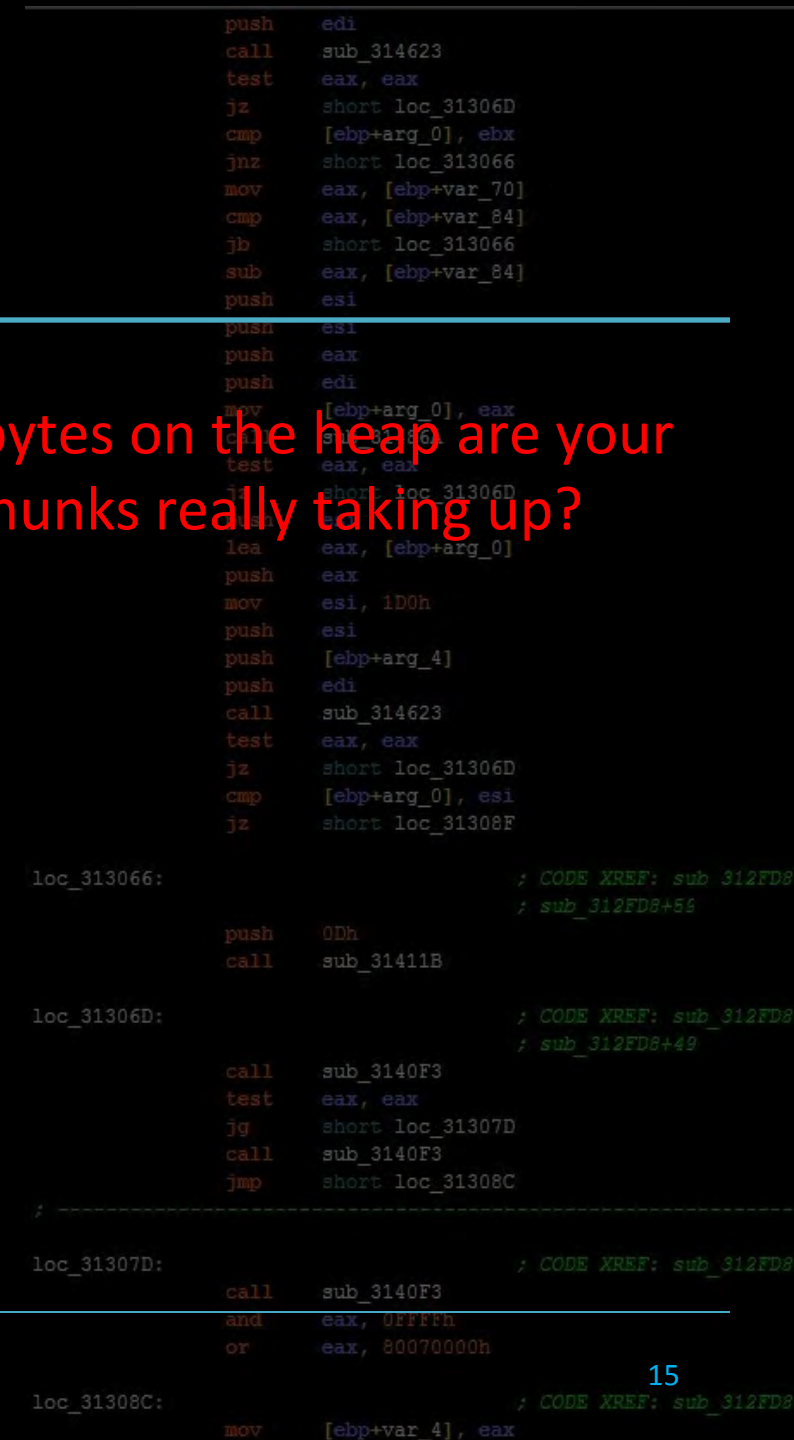
How many bytes on the heap are your malloc chunks really taking up?



Malloc Trivia

- malloc(32);
  - 40 bytes
- malloc(4);
  - 16 bytes
- malloc(20);
  - 24 bytes
- malloc(0);

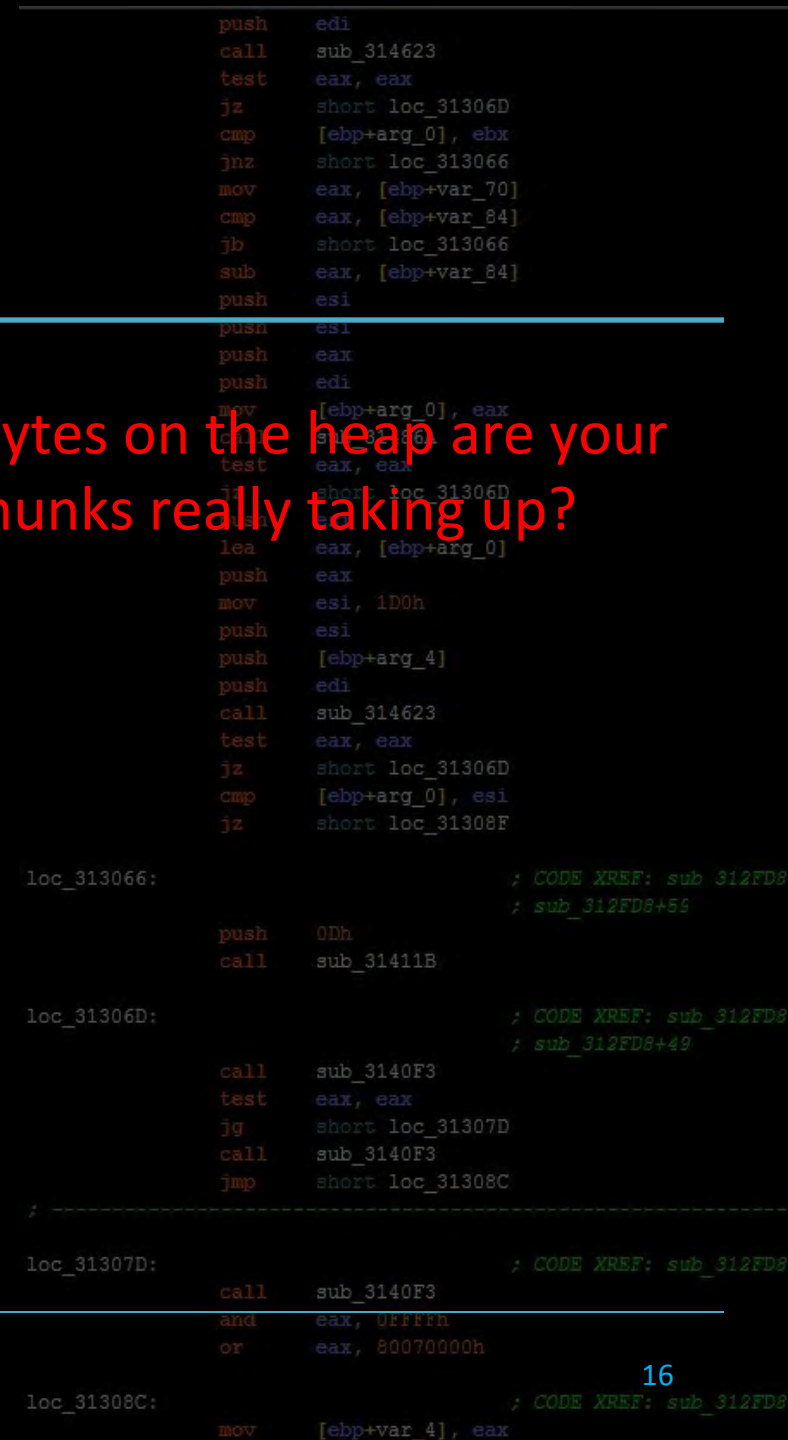
How many bytes on the heap are your malloc chunks really taking up?



Malloc Trivia

- malloc(32);
  - 40 bytes
- malloc(4);
  - 16 bytes
- malloc(20);
  - 24 bytes
- malloc(0);
  - 16 bytes

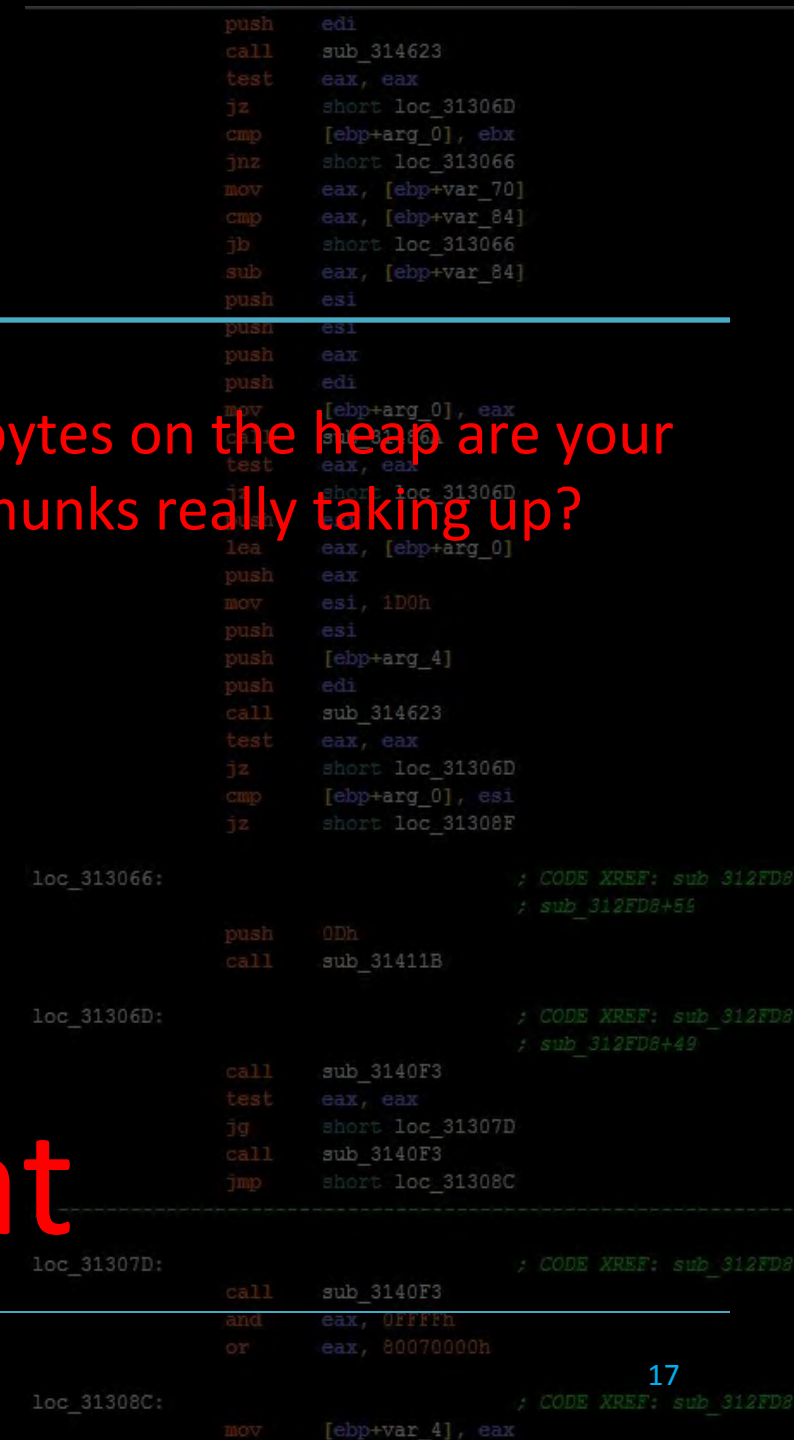
How many bytes on the heap are your malloc chunks really taking up?



Malloc Trivia

- malloc(32);
  - 40 bytes
- malloc(4);
  - 16 bytes
- malloc(20);
  - 24 bytes
- malloc(0);
  - 16 bytes ← lolwat

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



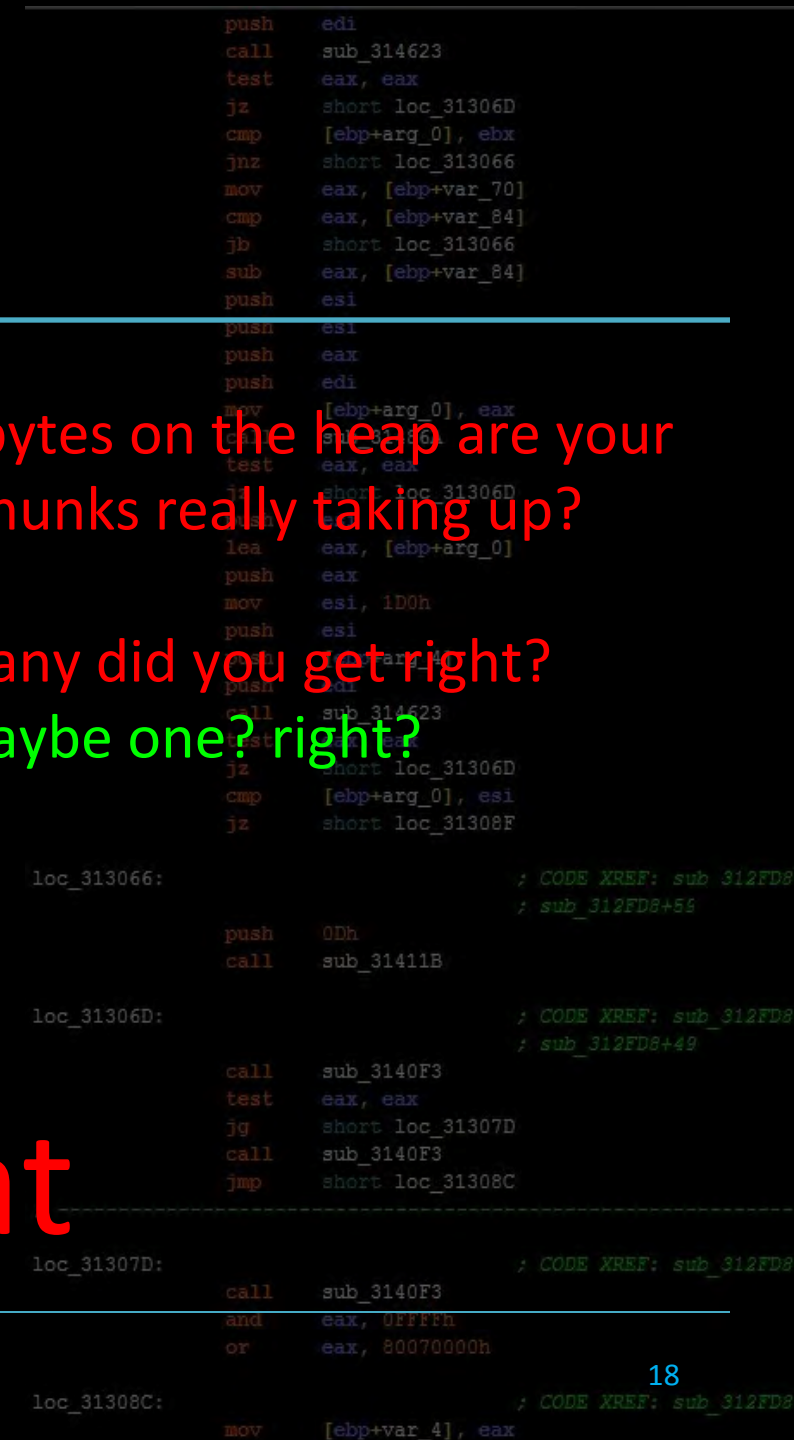
Malloc Trivia

- malloc(32);
  - 40 bytes
- malloc(4);
  - 16 bytes
- malloc(20);
  - 24 bytes
- malloc(0);
  - 16 bytes ← lolwat

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

How many did you get right?

Maybe one? right?



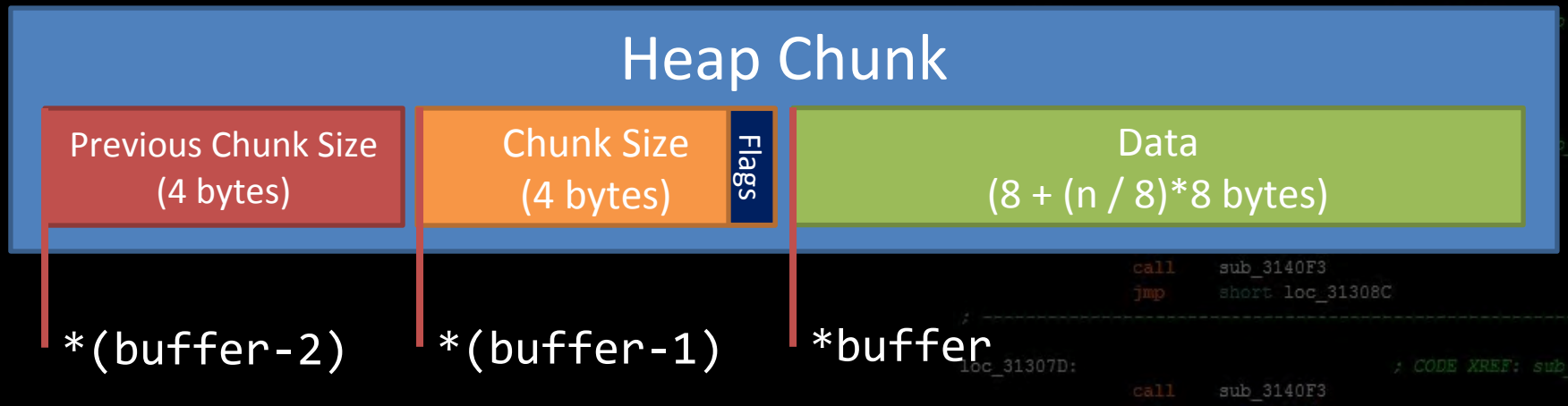
/levels/lecture/heap/sizes

prints distance between mallocs (size of chunk)

```
lecture@warzone:/levels/lecture/heap$ ./sizes
malloc(32) is at 0x086a6008,  40 bytes to the next pointer
malloc( 4) is at 0x086a6030,  16 bytes to the next pointer
malloc(20) is at 0x086a6040,  24 bytes to the next pointer
malloc( 0) is at 0x086a6058,  16 bytes to the next pointer
malloc(64) is at 0x086a6068,  72 bytes to the next pointer
malloc(32) is at 0x086a60b0,  40 bytes to the next pointer
malloc(32) is at 0x086a60d8,  40 bytes to the next pointer
malloc(32) is at 0x086a6100,  40 bytes to the next pointer
malloc(32) is at 0x086a6128,  40 bytes to the next pointer
lecture@warzone:/levels/lecture/heap$
```

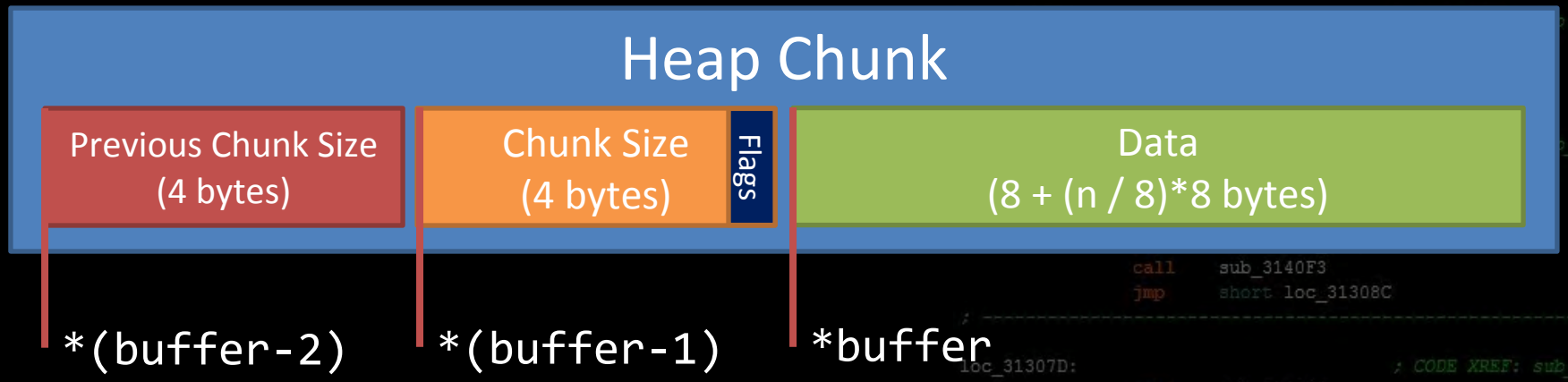
Heap Chunks

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

//Out comes a heap chunk
```



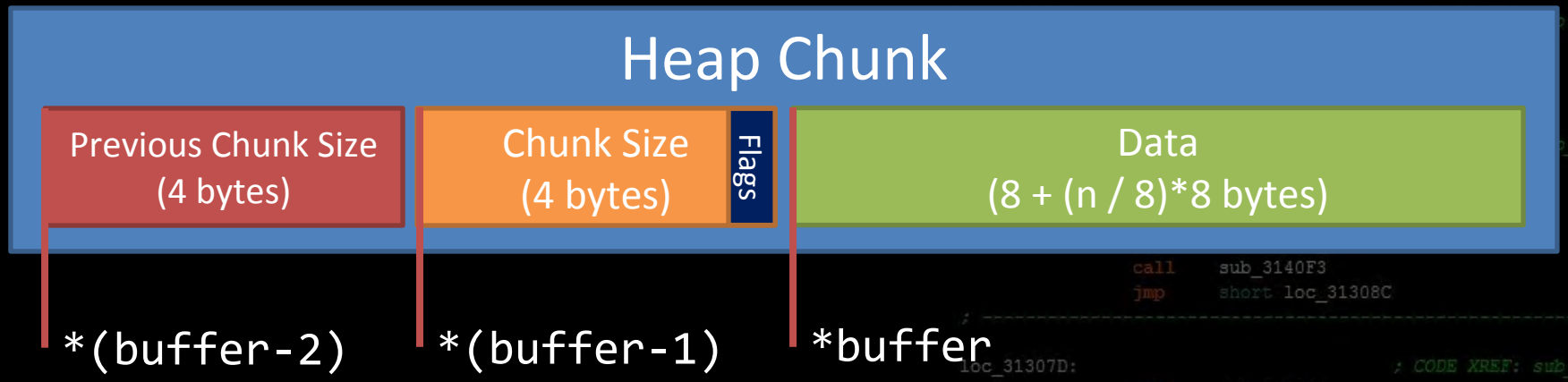
Heap Chunks

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



Heap Chunks

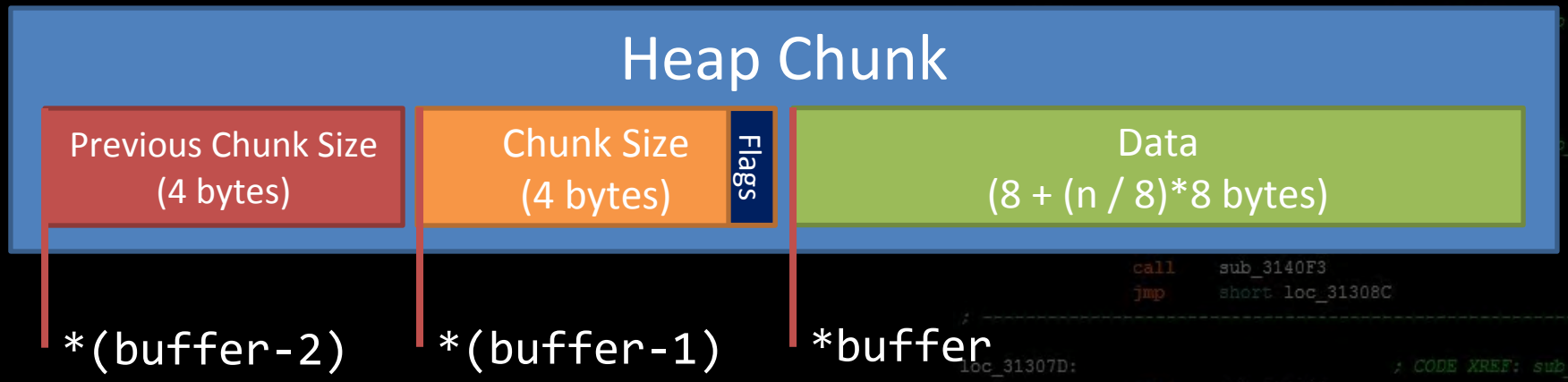
- Data
  - Your newly allocated memory / ptr returned by malloc



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
```



/levels/lecture/heap/heap_chunks

prints heap chunks fields

```
lecture@warzone:/levels/lecture/heap$
lecture@warzone:/levels/lecture/heap$ ./heap_chunks
mallocing...
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08276008) -------> ... ] - from malloc(0)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08276018) -------> ... ] - from malloc(4)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08276028) -------> ... ] - from malloc(8)
[ prev - 0x00000000 ][ size - 0x00000019 ][ data buffer (0x08276038) -------> ... ] - from malloc(16)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x08276050) -------> ... ] - from malloc(24)
[ prev - 0x00000000 ][ size - 0x00000029 ][ data buffer (0x08276070) -------> ... ] - from malloc(32)
[ prev - 0x00000000 ][ size - 0x00000049 ][ data buffer (0x08276098) -------> ... ] - from malloc(64)
[ prev - 0x00000000 ][ size - 0x00000089 ][ data buffer (0x082760e0) -------> ... ] - from malloc(128)
[ prev - 0x00000000 ][ size - 0x00000109 ][ data buffer (0x08276168) -------> ... ] - from malloc(256)
[ prev - 0x00000000 ][ size - 0x00000209 ][ data buffer (0x08276270) -------> ... ] - from malloc(512)
[ prev - 0x00000000 ][ size - 0x00000409 ][ data buffer (0x08276478) -------> ... ] - from malloc(1024)
[ prev - 0x00000000 ][ size - 0x00000809 ][ data buffer (0x08276880) -------> ... ] - from malloc(2048)
[ prev - 0x00000000 ][ size - 0x00001009 ][ data buffer (0x08277088) -------> ... ] - from malloc(4096)
[ prev - 0x00000000 ][ size - 0x00002009 ][ data buffer (0x08278090) -------> ... ] - from malloc(8192)
[ prev - 0x00000000 ][ size - 0x00004009 ][ data buffer (0x0827a098) -------> ... ] - from malloc(16384)
lecture@warzone:/levels/lecture/heap$
```

Pseudo Memory Map

0x00000000 – Start of memory

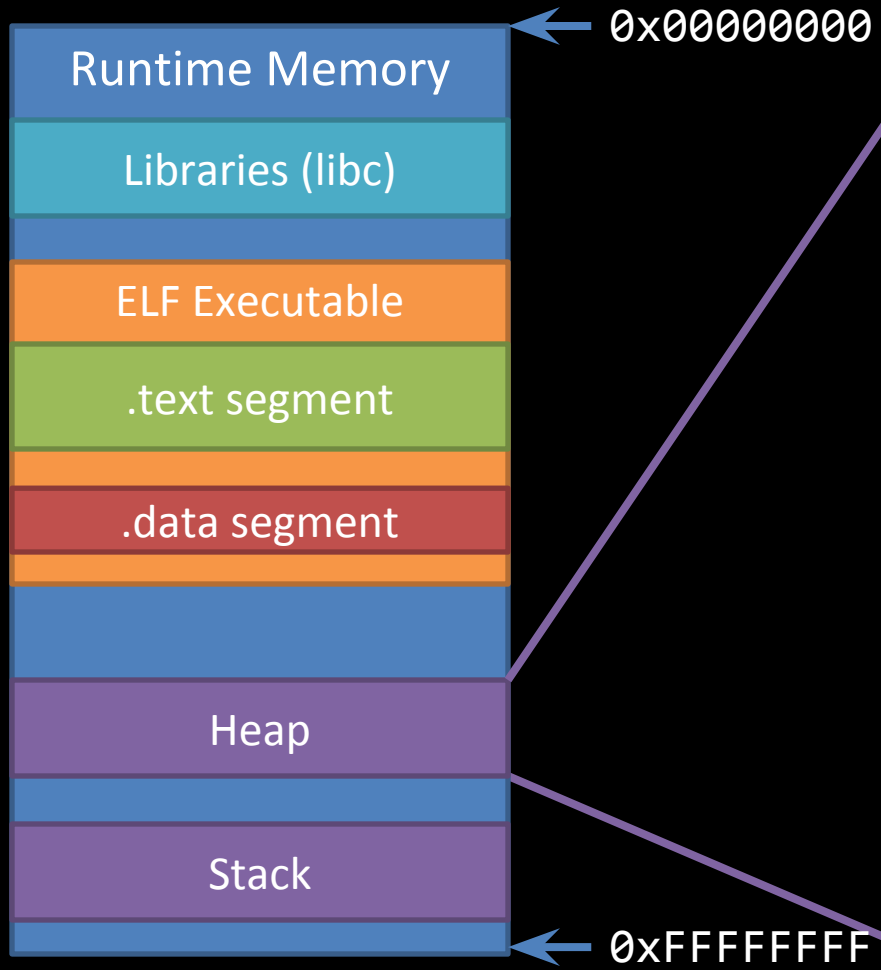
0x08048000 – Start of .text Segment

0xb7ff0000 – Top of heap

0xbfff0000 – Top of stack

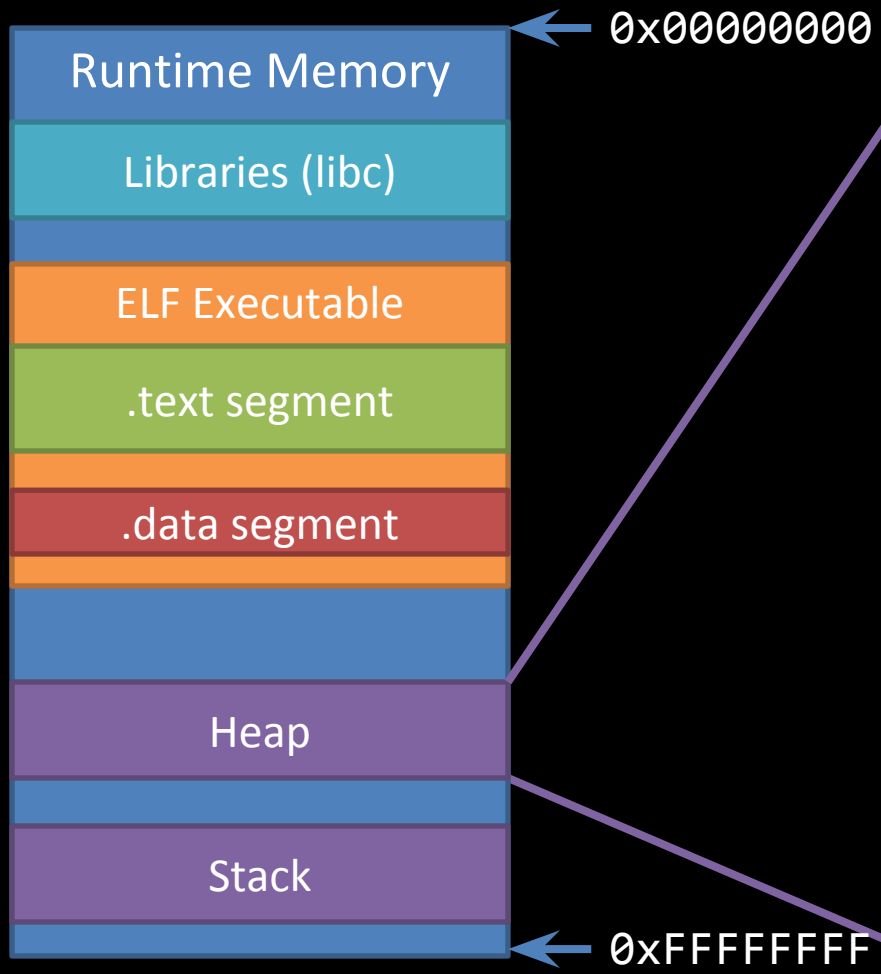
0xFFFFFFFF – End of memory

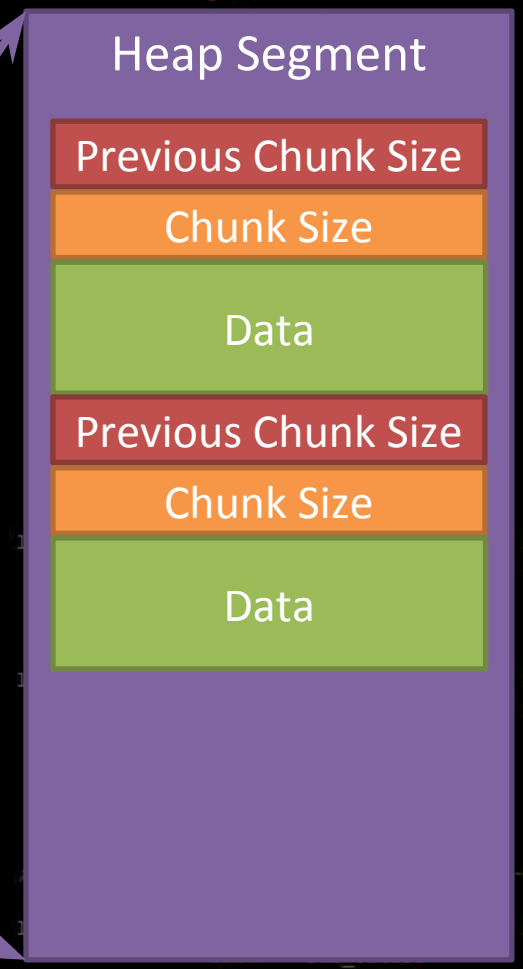
Heap Allocations



Grows towards higher memory

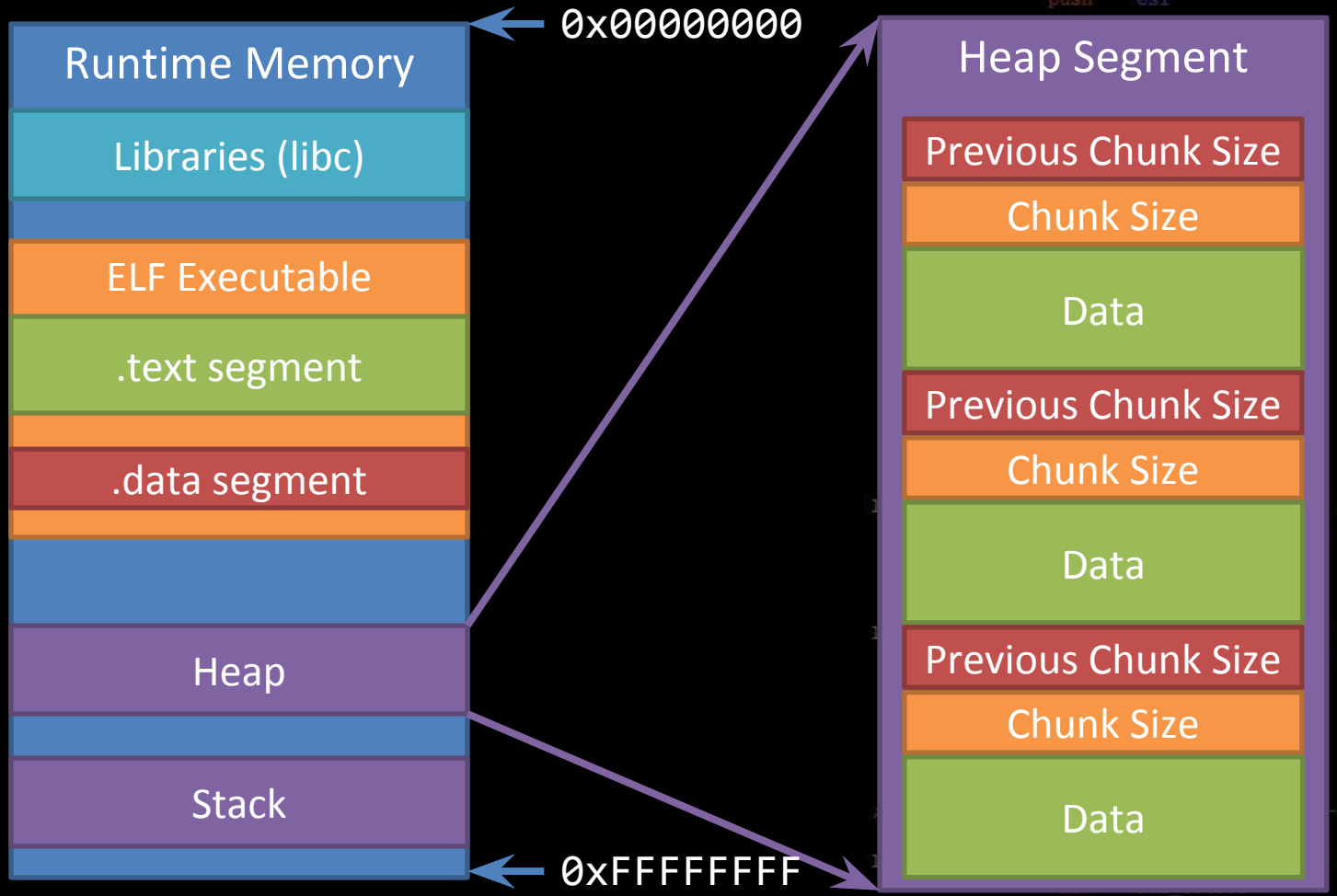
Heap Allocations





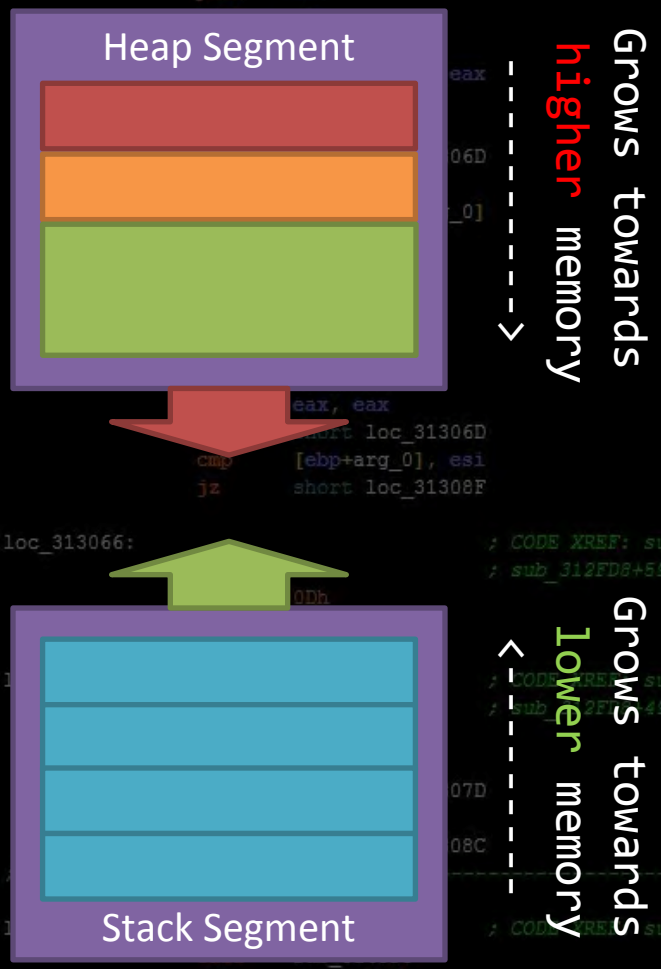
Grows towards higher memory

Heap Allocations

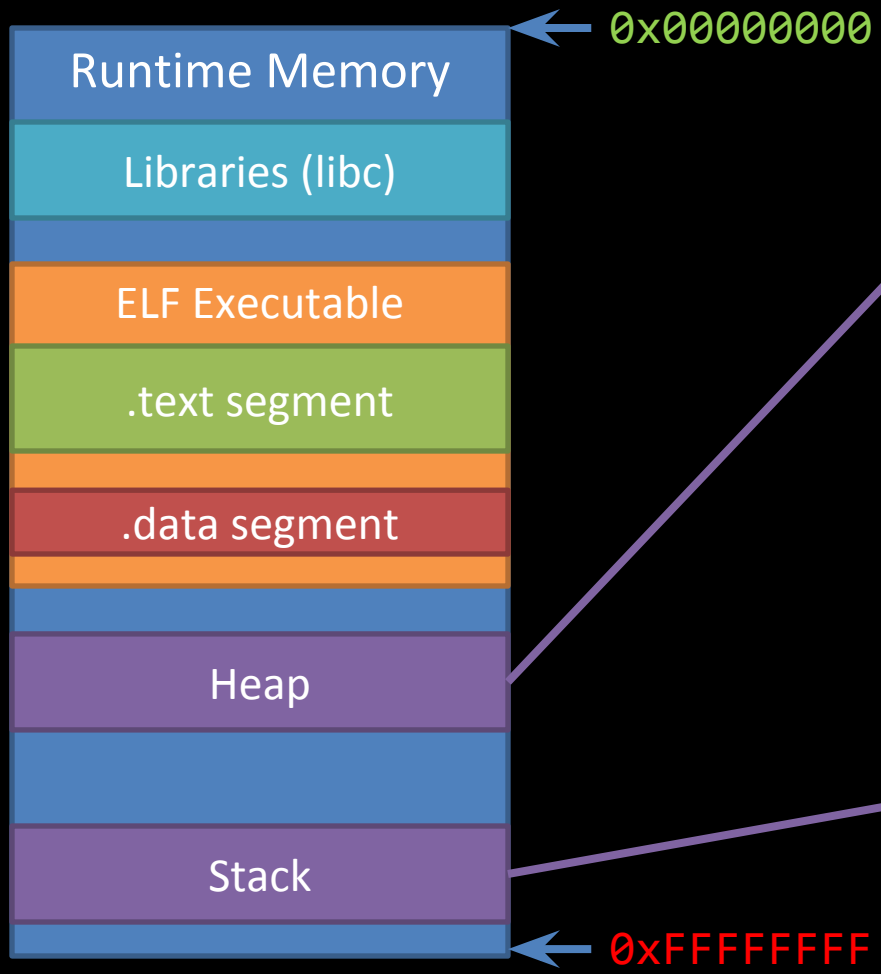


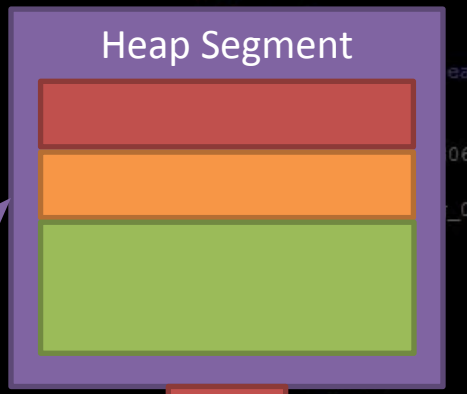
Segment Growth

- Heap grows DOWN towards higher memory
- Stack grows UP towards lower memory

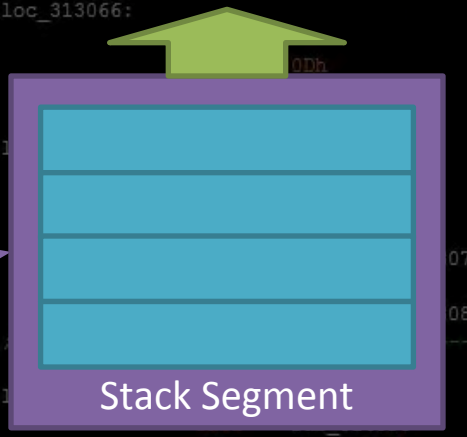


Segment Growth





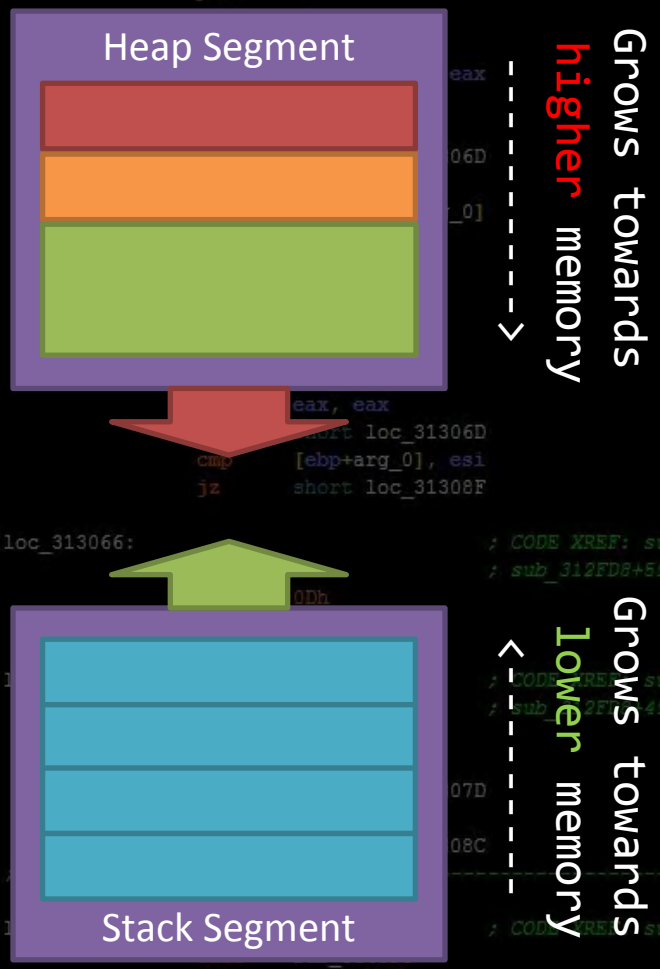
Grows towards higher memory



Grows towards lower memory

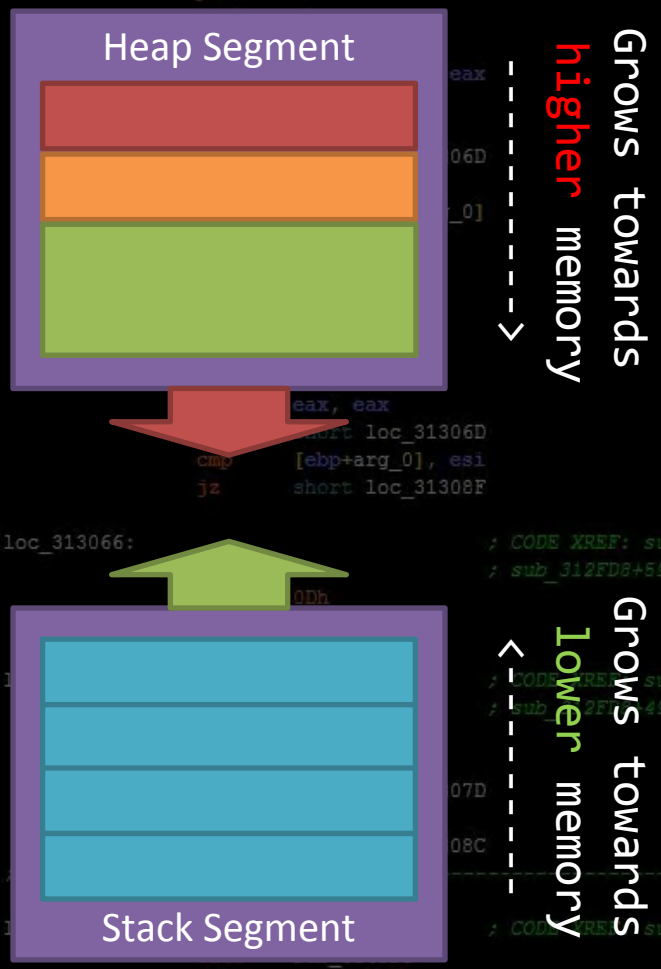
Segment Growth

- Heap grows DOWN towards higher memory
- Stack grows UP towards lower memory
- Any ideas why?



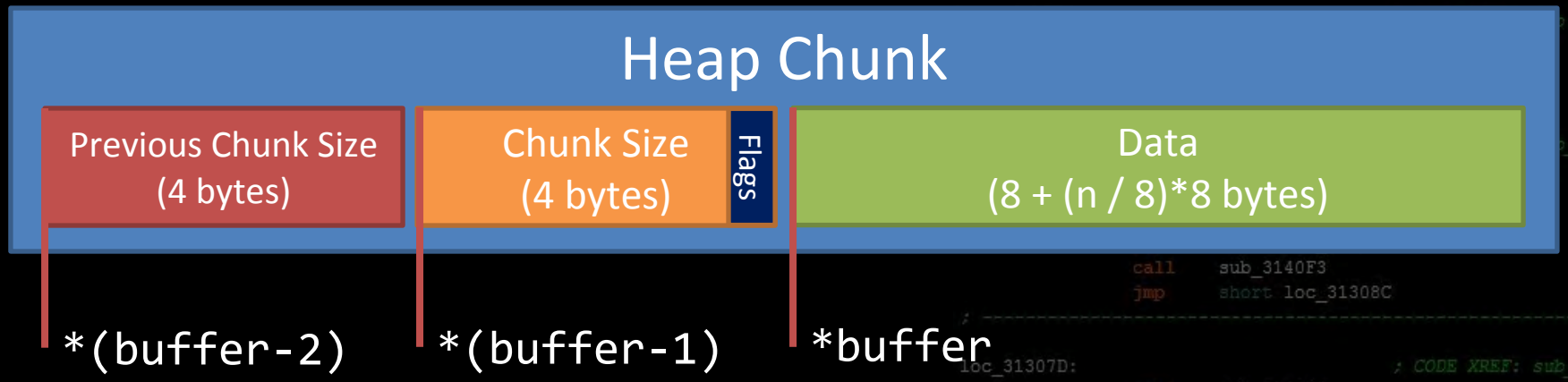
Segment Growth

- 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

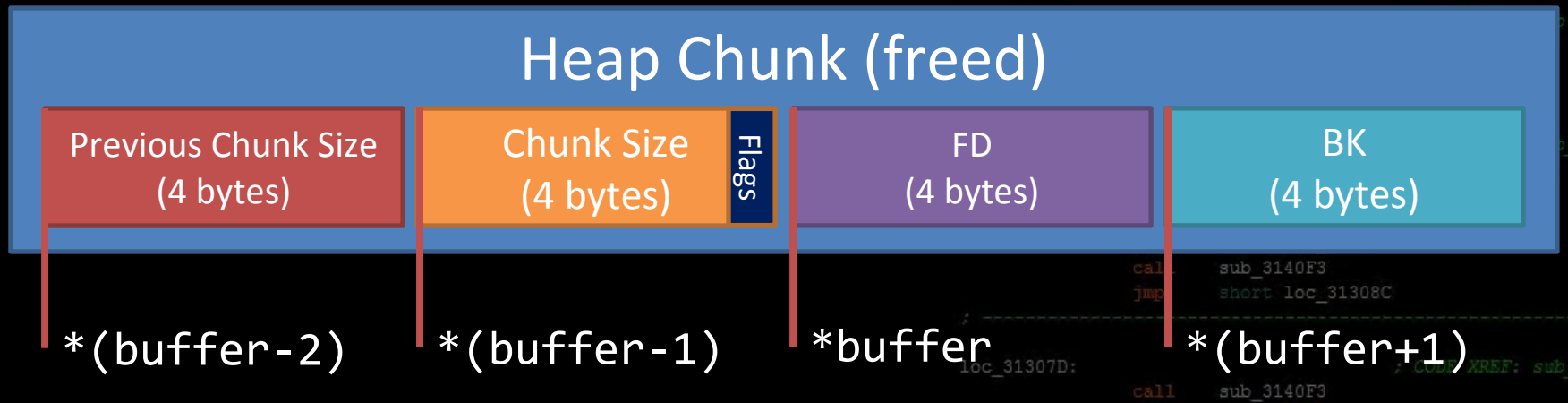
- Heap chunks exist in two states
  - in use (malloc'd)
  - free'd



Heap Chunks – Freed

```
free(buffer);
```

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



/levels/lecture/heap/print_frees

```
lecture@warzone:/levels/lecture/heap$ ./print_frees
mallocing...
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08265008) ----> ... ] - Chunk 0x08265000 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08265018) ----> ... ] - Chunk 0x08265010 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08265028) ----> ... ] - Chunk 0x08265020 - In use
[ prev - 0x00000000 ][ size - 0x00000019 ][ data buffer (0x08265038) ----> ... ] - Chunk 0x08265030 - In use
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x08265050) ----> ... ] - Chunk 0x08265048 - In use
[ prev - 0x00000000 ][ size - 0x00000029 ][ data buffer (0x08265070) ----> ... ] - Chunk 0x08265068 - In use
[ prev - 0x00000000 ][ size - 0x00000049 ][ data buffer (0x08265098) ----> ... ] - Chunk 0x08265090 - In use
[ prev - 0x00000000 ][ size - 0x00000089 ][ data buffer (0x082650e0) ----> ... ] - Chunk 0x082650d8 - In use
[ prev - 0x00000000 ][ size - 0x00000109 ][ data buffer (0x08265168) ----> ... ] - Chunk 0x08265160 - In use
[ prev - 0x00000000 ][ size - 0x00000209 ][ data buffer (0x08265270) ----> ... ] - Chunk 0x08265268 - In use
[ prev - 0x00000000 ][ size - 0x00000409 ][ data buffer (0x08265478) ----> ... ] - Chunk 0x08265470 - In use
[ prev - 0x00000000 ][ size - 0x00000809 ][ data buffer (0x08265880) ----> ... ] - Chunk 0x08265878 - In use
[ prev - 0x00000000 ][ size - 0x00001009 ][ data buffer (0x08266088) ----> ... ] - Chunk 0x08266080 - In use
[ prev - 0x00000000 ][ size - 0x00002009 ][ data buffer (0x08267090) ----> ... ] - Chunk 0x08267088 - In use
[ prev - 0x00000000 ][ size - 0x00004009 ][ data buffer (0x08269098) ----> ... ] - Chunk 0x08269090 - In use

freeing every other chunk...
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x08265020 ][ bk - 0x08265048 ] - Chunk 0x08265000 - Freed
[ prev - 0x00000010 ][ size - 0x00000010 ][ data buffer (0x08265018) ----> ... ] - Chunk 0x08265010 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x08266080 ][ bk - 0x08265000 ] - Chunk 0x08265020 - Freed
[ prev - 0x00000010 ][ size - 0x00000018 ][ data buffer (0x08265038) ----> ... ] - Chunk 0x08265030 - In use
[ prev - 0x00000000 ][ size - 0x00000021 ][ fd - 0x08265000 ][ bk - 0xb7730450 ] - Chunk 0x08265048 - Freed
[ prev - 0x00000020 ][ size - 0x00000028 ][ data buffer (0x08265070) ----> ... ] - Chunk 0x08265068 - In use
[ prev - 0x00000000 ][ size - 0x00000049 ][ fd - 0xb7730450 ][ bk - 0x08265160 ] - Chunk 0x08265090 - Freed
[ prev - 0x00000048 ][ size - 0x00000088 ][ data buffer (0x082650e0) ----> ... ] - Chunk 0x082650d8 - In use
[ prev - 0x00000000 ][ size - 0x00000109 ][ fd - 0x08265090 ][ bk - 0x08265470 ] - Chunk 0x08265160 - Freed
[ prev - 0x00000108 ][ size - 0x00000208 ][ data buffer (0x08265270) ----> ... ] - Chunk 0x08265268 - In use
[ prev - 0x00000000 ][ size - 0x00000409 ][ fd - 0x08265160 ][ bk - 0x08266080 ] - Chunk 0x08265470 - Freed
[ prev - 0x00000408 ][ size - 0x00000808 ][ data buffer (0x08265880) ----> ... ] - Chunk 0x08265878 - In use
[ prev - 0x00000000 ][ size - 0x00001009 ][ fd - 0x08265470 ][ bk - 0x08265020 ] - Chunk 0x08266080 - Freed
[ prev - 0x00001008 ][ size - 0x00002008 ][ data buffer (0x08267090) ----> ... ] - Chunk 0x08267088 - In use
lecture@warzone:/levels/lecture/heap$
```

/levels/lecture/heap/print_frees

prints heap chunks in their different states

```
freeing every other chunk...
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x08265020 ][ bk - 0x08265048 ] - Chunk 0x08265000 - Freed
[ prev - 0x00000010 ][ size - 0x00000010 ][ data buffer (0x08265018) ----> ... ] - Chunk 0x08265010 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x08266080 ][ bk - 0x08265000 ] - Chunk 0x08265020 - Freed
[ prev - 0x00000010 ][ size - 0x00000018 ][ data buffer (0x08265038) ----> ... ] - Chunk 0x08265030 - In use
[ prev - 0x00000000 ][ size - 0x00000021 ][ fd - 0x08265000 ][ bk - 0xb7730450 ] - Chunk 0x08265048 - Freed
[ prev - 0x00000020 ][ size - 0x00000028 ][ data buffer (0x08265070) ----> ... ] - Chunk 0x08265068 - In use
[ prev - 0x00000000 ][ size - 0x00000049 ][ fd - 0xb7730450 ][ bk - 0x08265160 ] - Chunk 0x08265090 - Freed
[ prev - 0x00000048 ][ size - 0x00000088 ][ data buffer (0x082650e0) ----> ... ] - Chunk 0x082650d8 - In use
[ prev - 0x00000000 ][ size - 0x00000109 ][ fd - 0x08265090 ][ bk - 0x08265470 ] - Chunk 0x08265160 - Freed
[ prev - 0x00000108 ][ size - 0x00000208 ][ data buffer (0x08265270) ----> ... ] - Chunk 0x08265268 - In use
[ prev - 0x00000000 ][ size - 0x00000409 ][ fd - 0x08265160 ][ bk - 0x08266080 ] - Chunk 0x08265470 - Freed
[ prev - 0x00000408 ][ size - 0x00000808 ][ data buffer (0x08265880) ----> ... ] - Chunk 0x08265878 - In use
[ prev - 0x00000000 ][ size - 0x00001009 ][ fd - 0x08265470 ][ bk - 0x08265020 ] - Chunk 0x08266080 - Freed
[ prev - 0x00001008 ][ size - 0x00002008 ][ data buffer (0x08267090) ----> ... ] - Chunk 0x08267088 - In use
lecture@warzone:/levels/lecture/heap$
```

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;
};
```

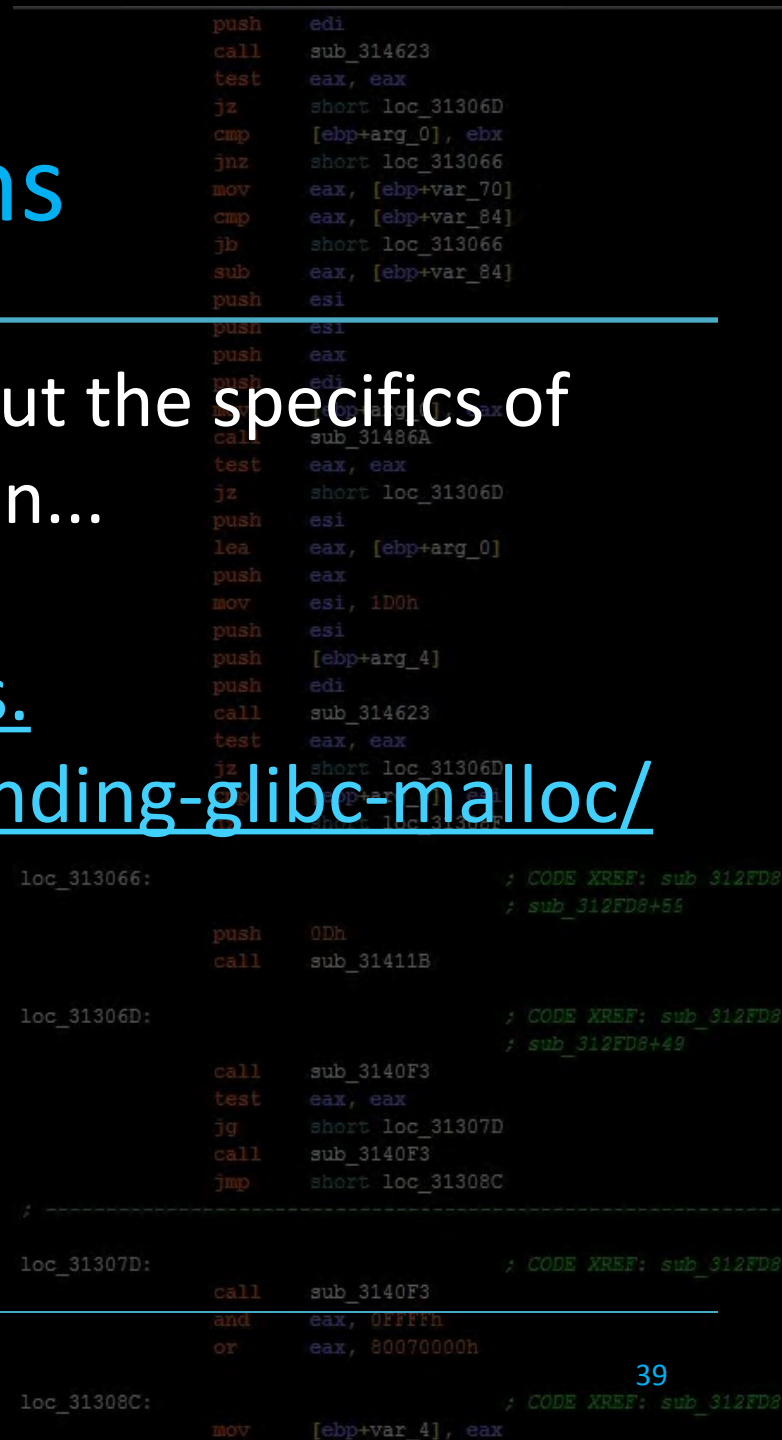
Heap Implementations

- 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

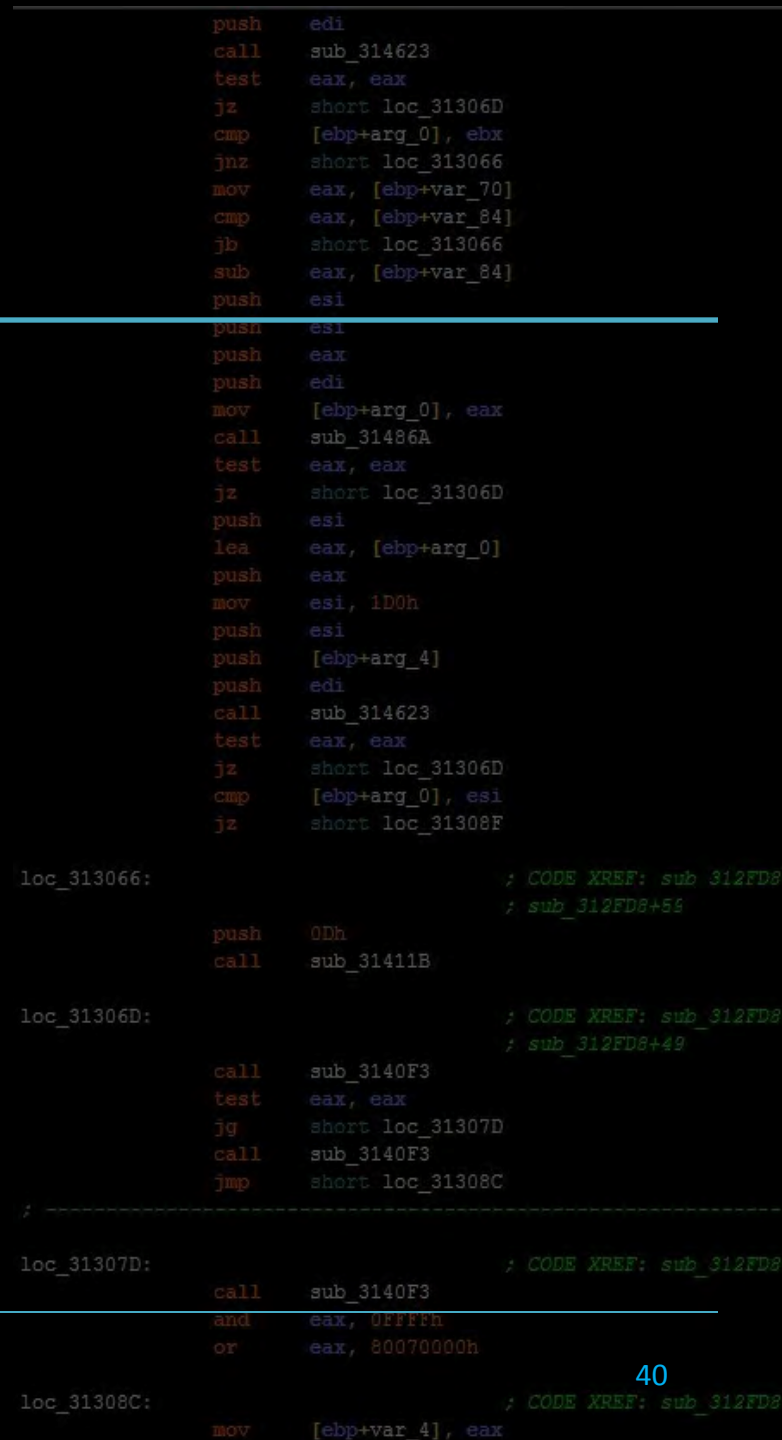
Heap Implementations

- 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!



Lecture Overview

- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

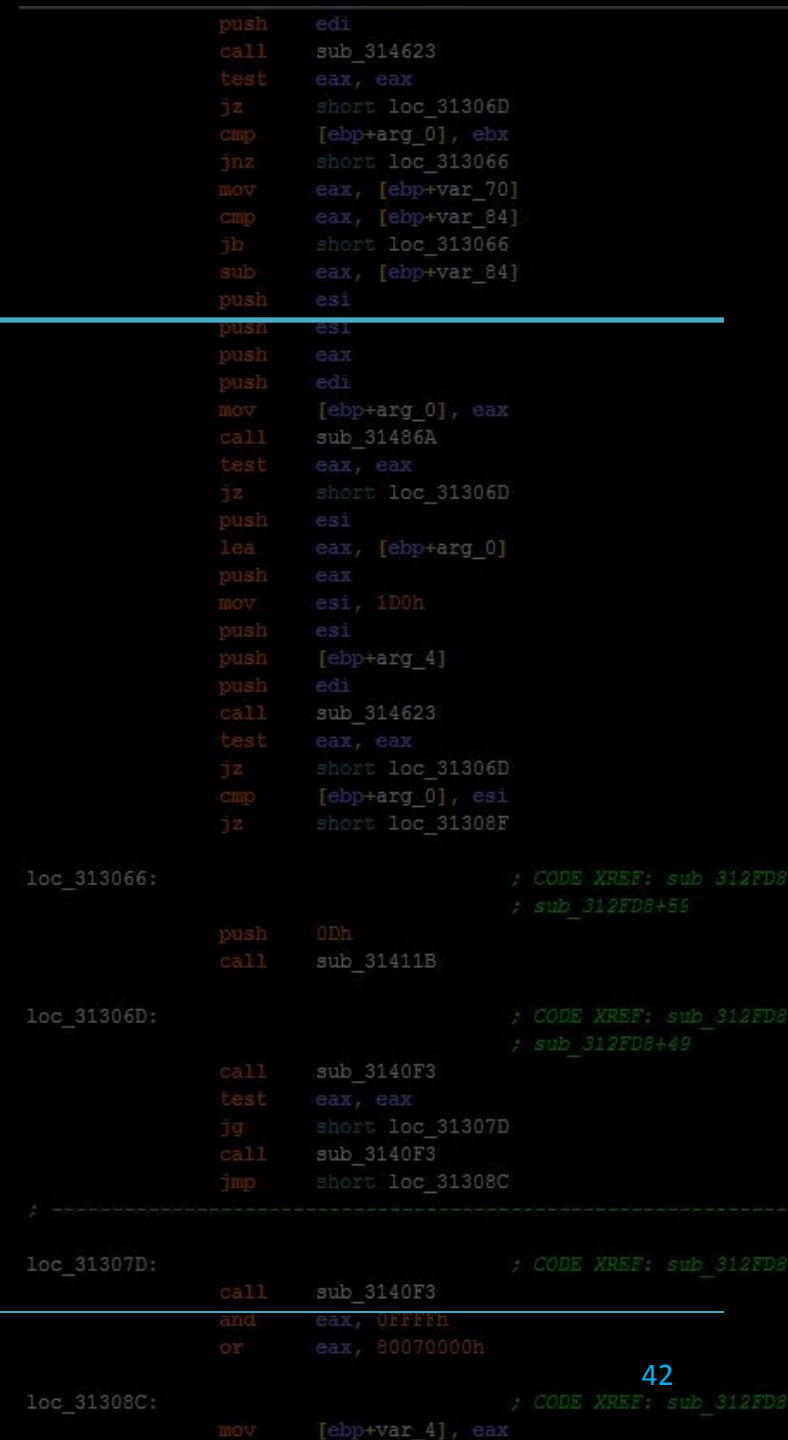


HEAP EXPLOITATION

Common heap related concepts as used in exploitation

Lecture Overview

- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

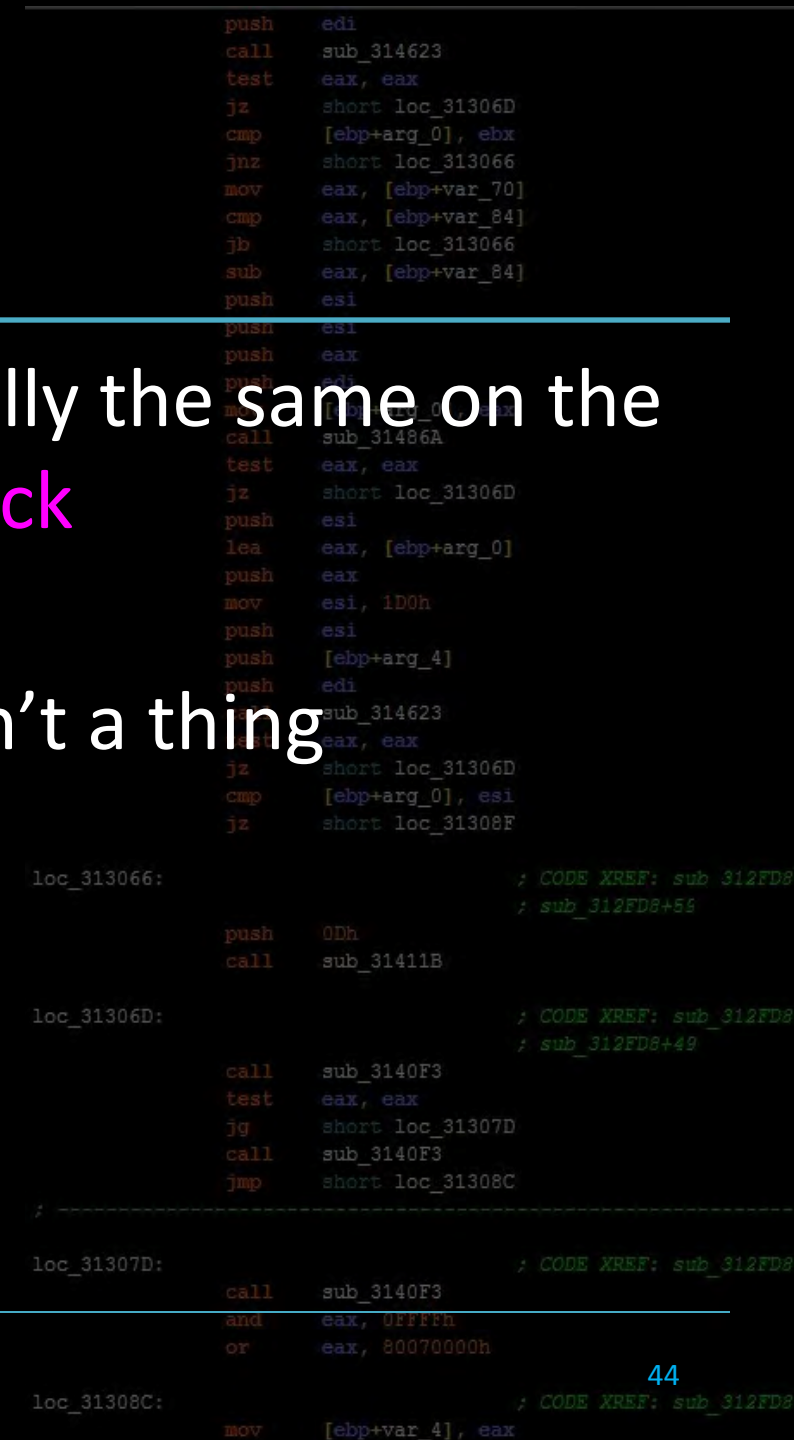


Heap Overflows

- Buffer overflows are basically the same on the heap as they are on the stack

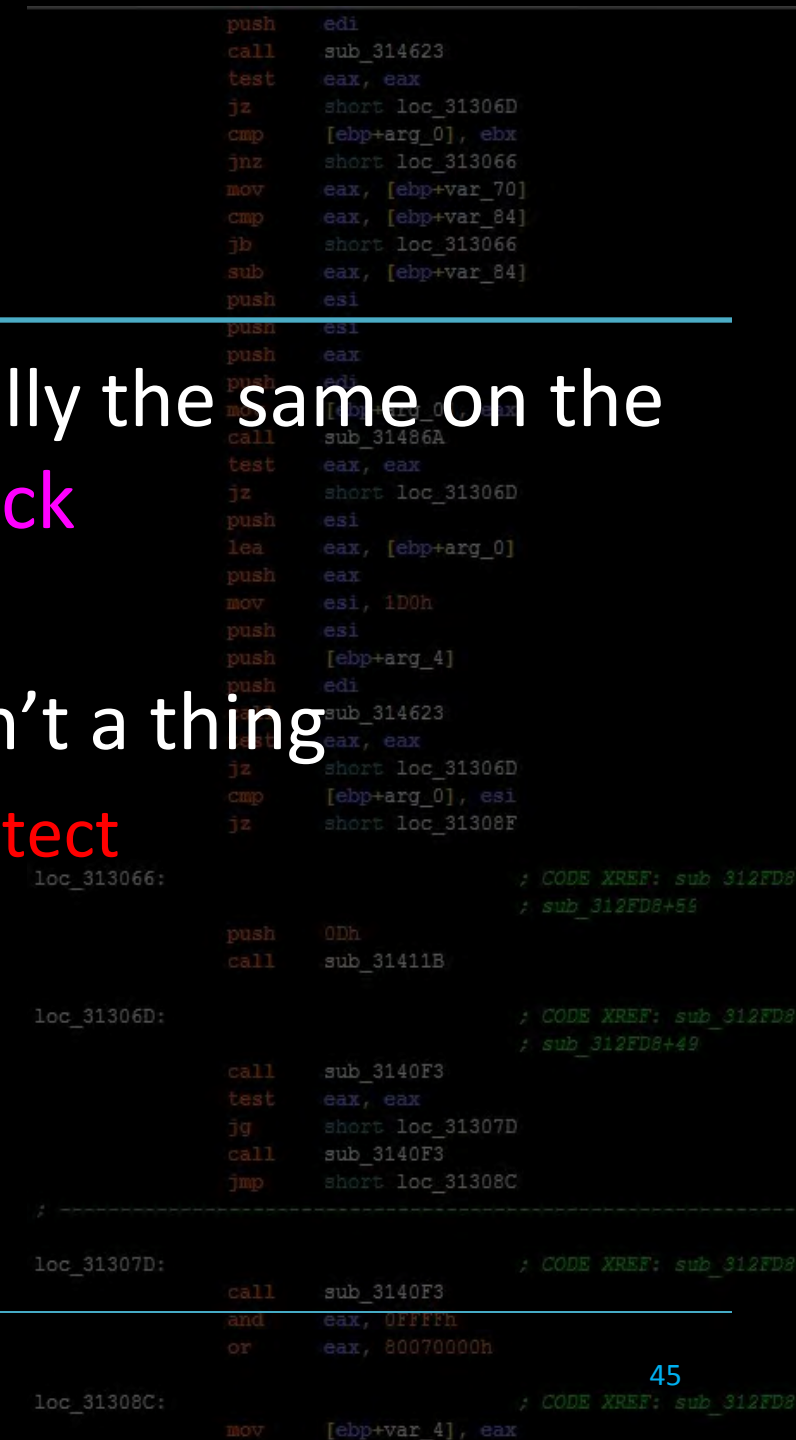
Heap Overflows

- Buffer overflows are basically the same on the heap as they are on the stack
- Heap cookies/canaries aren't a thing

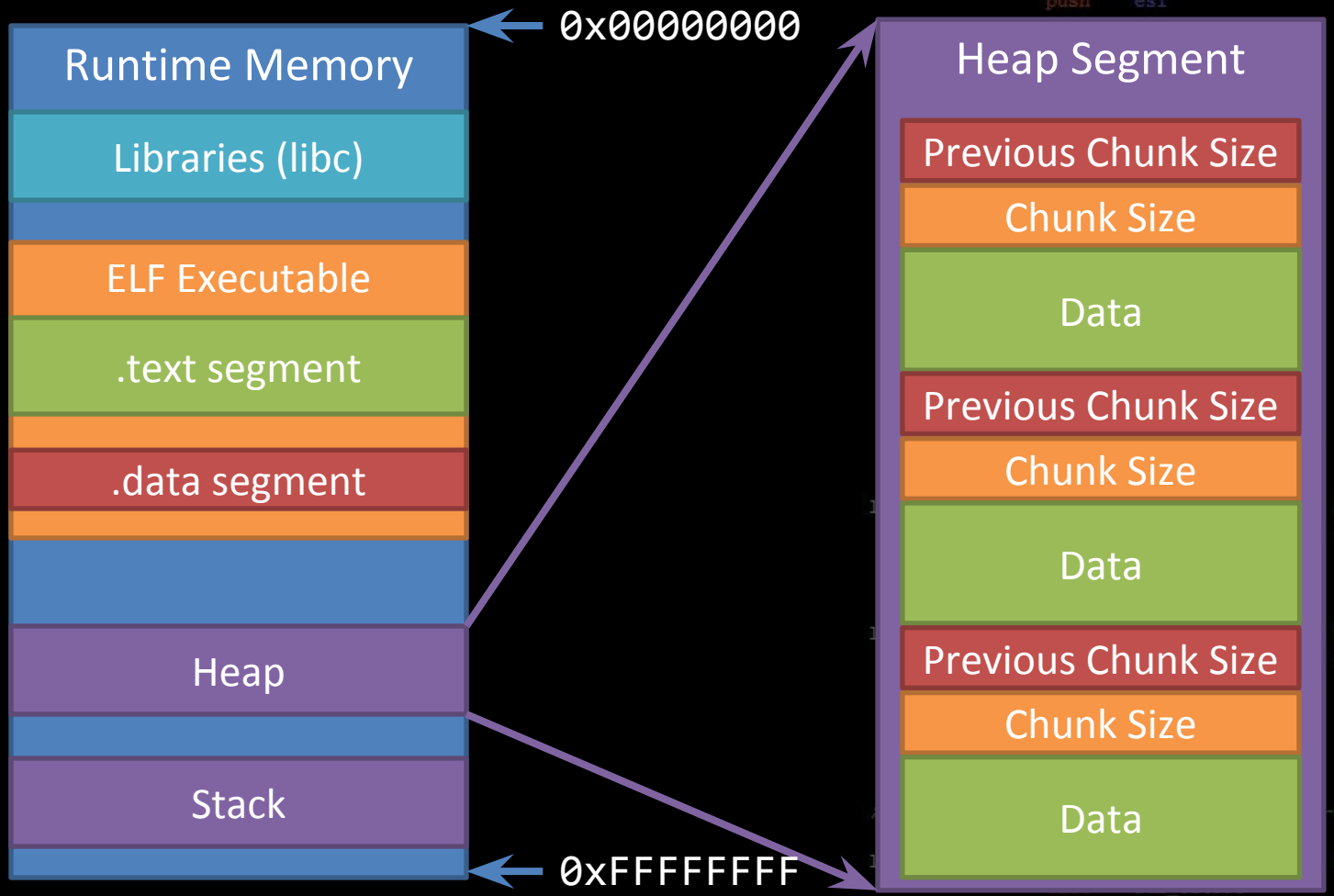


Heap Overflows

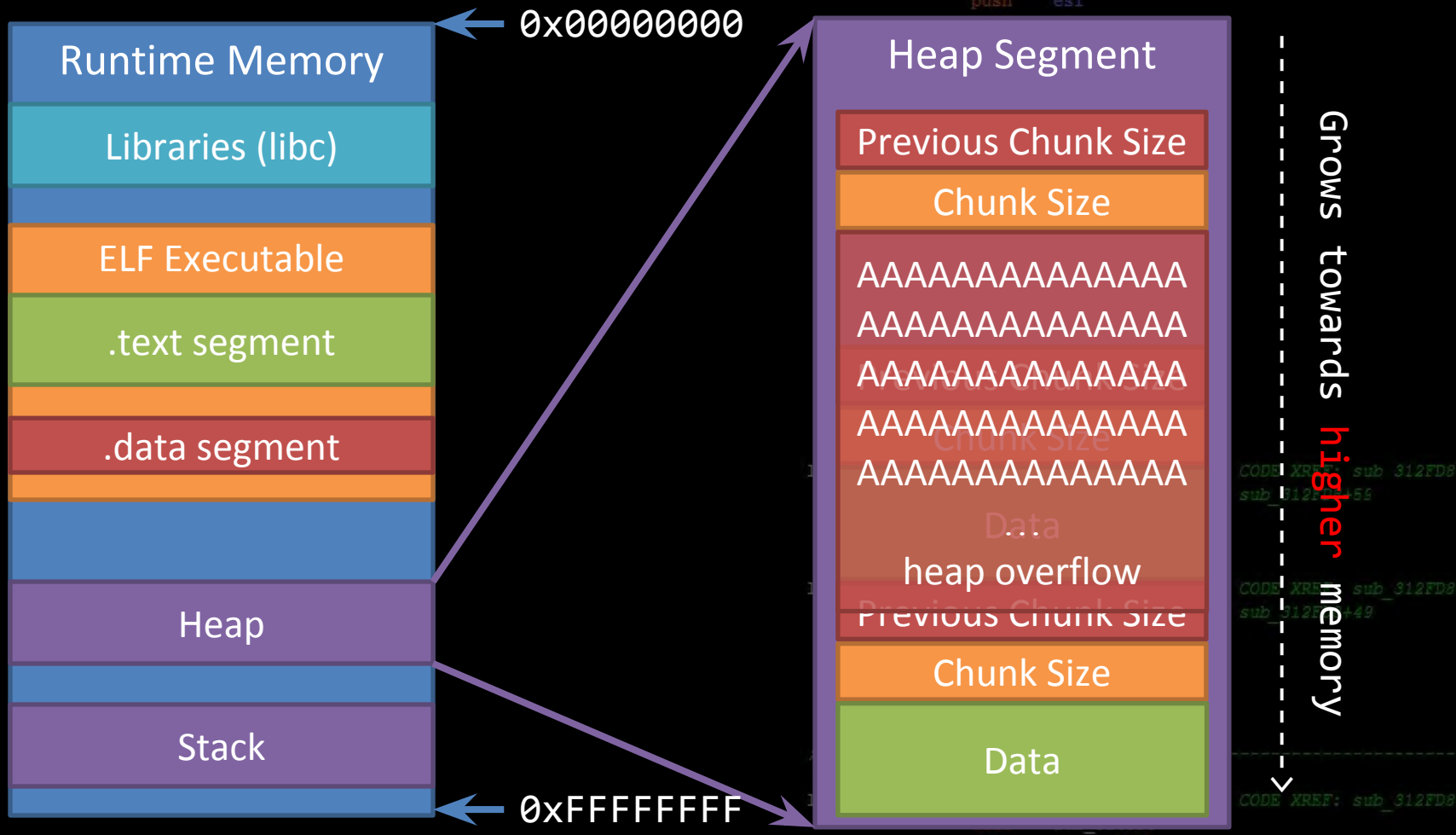
- Buffer overflows are basically the same on the heap as they are on the stack
- Heap cookies/canaries aren't a thing
  - No 'return' addresses to protect



Heap Overflows

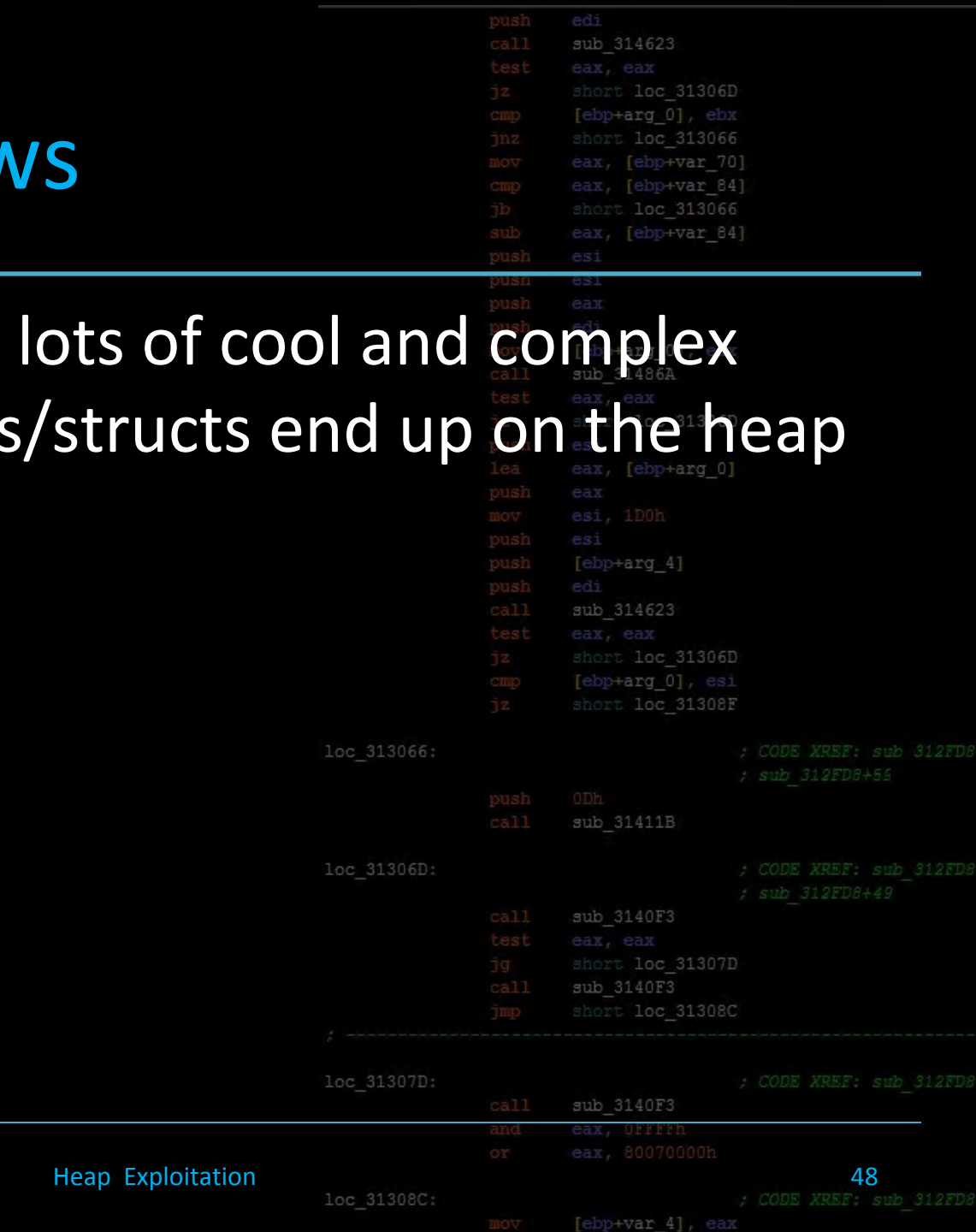


Heap Overflows



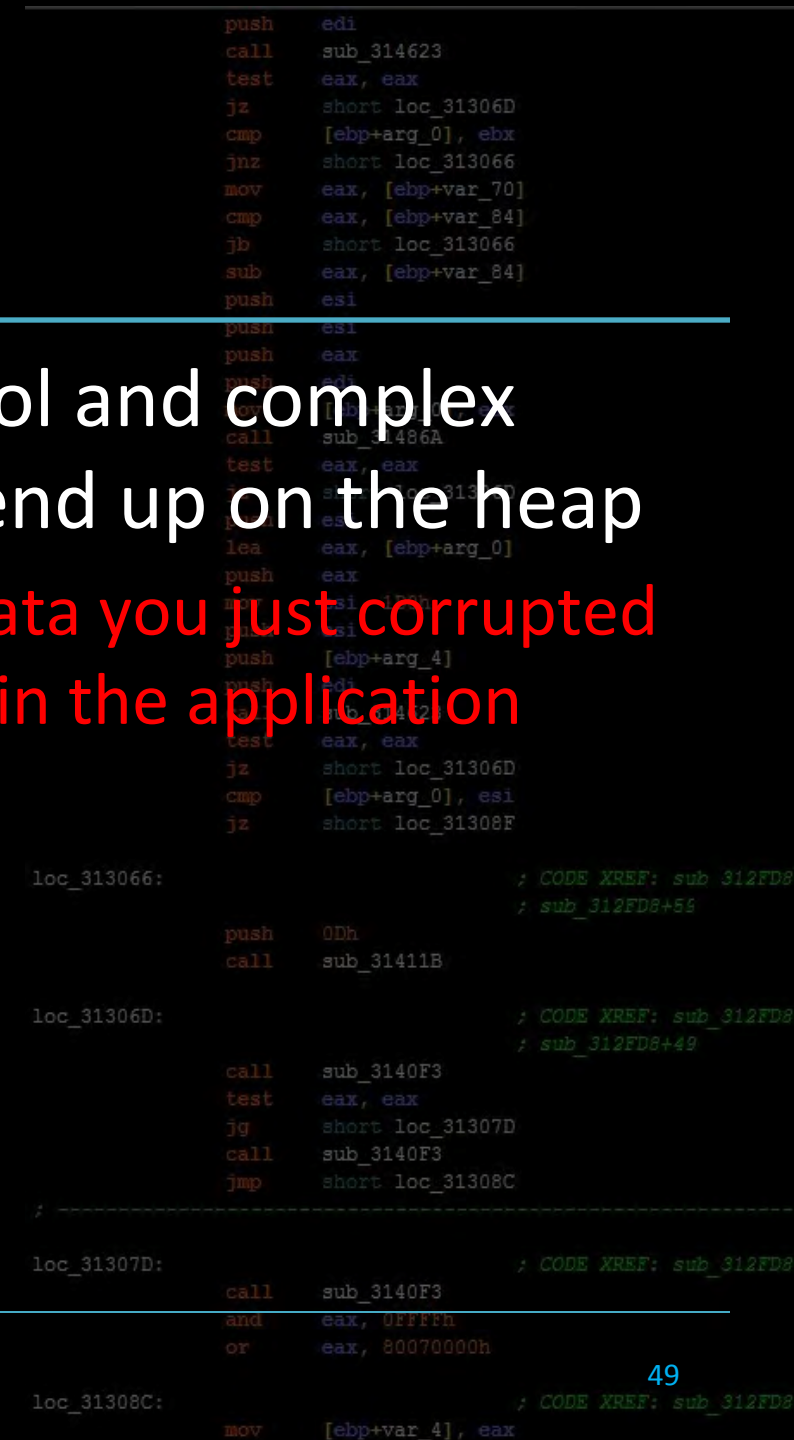
Heap Overflows

- In the real world, lots of cool and complex things like objects/structs end up on the heap



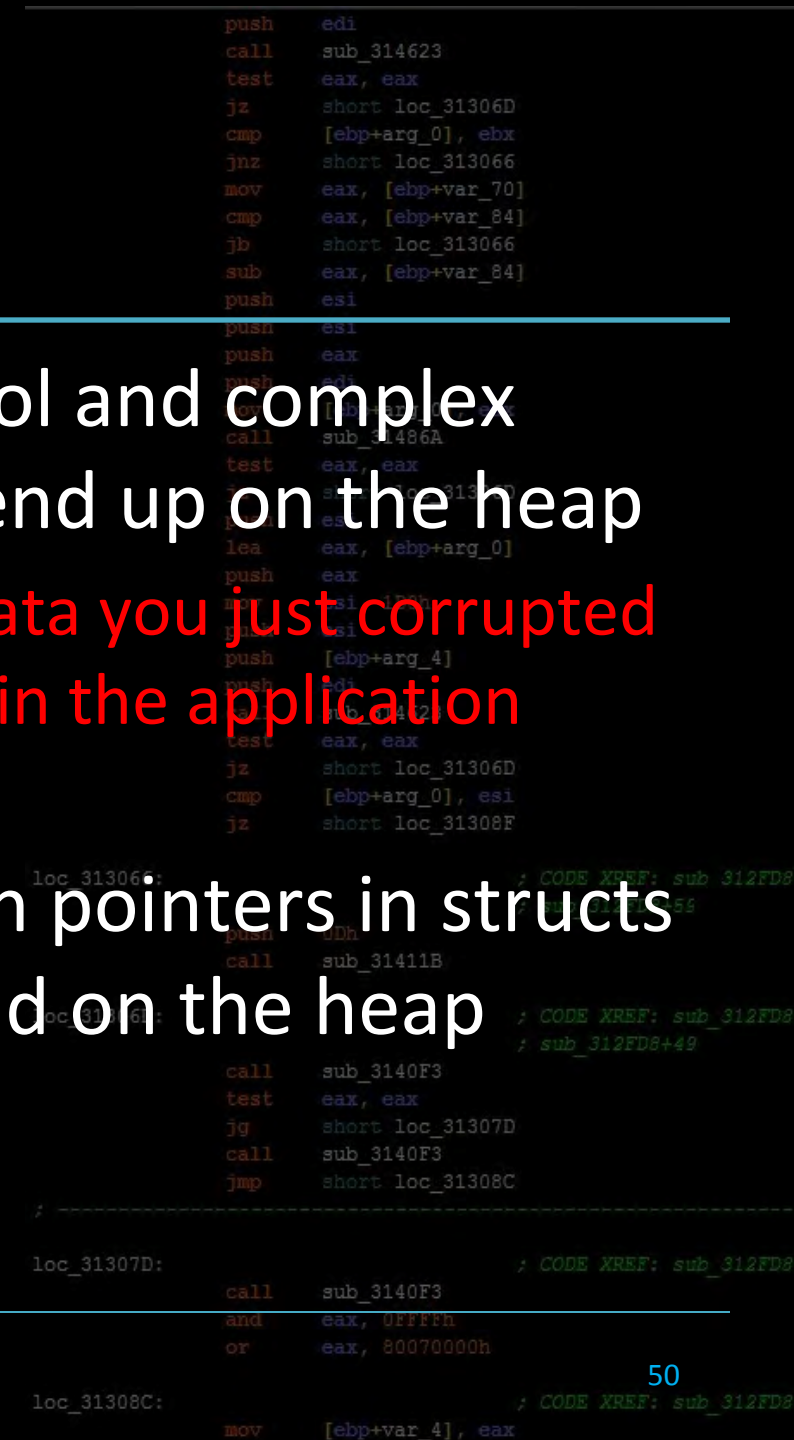
Heap Overflows

- 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



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



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

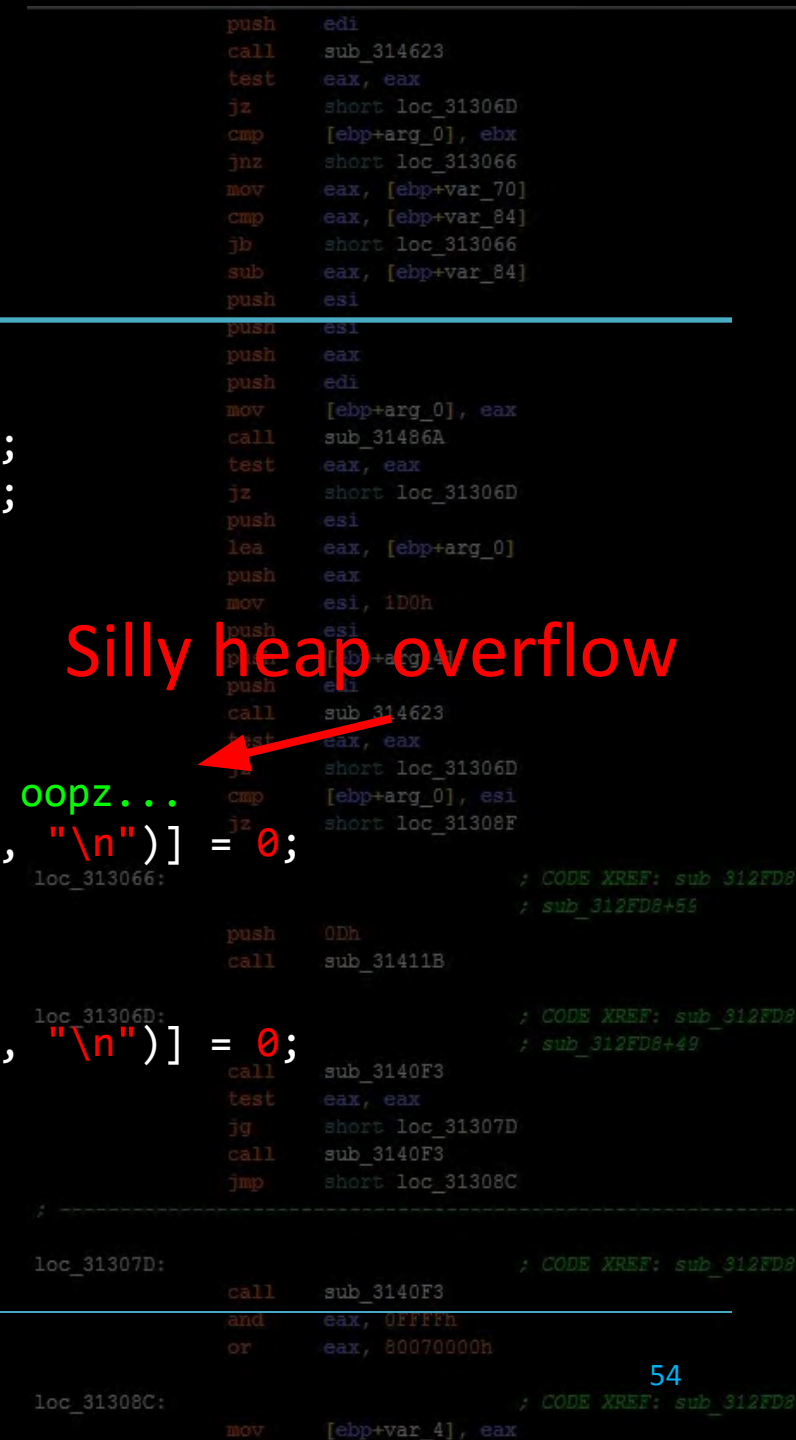
```
struct toystr {
    void (* message)(char *);
    char buffer[20];
};
```

Heap Overflows

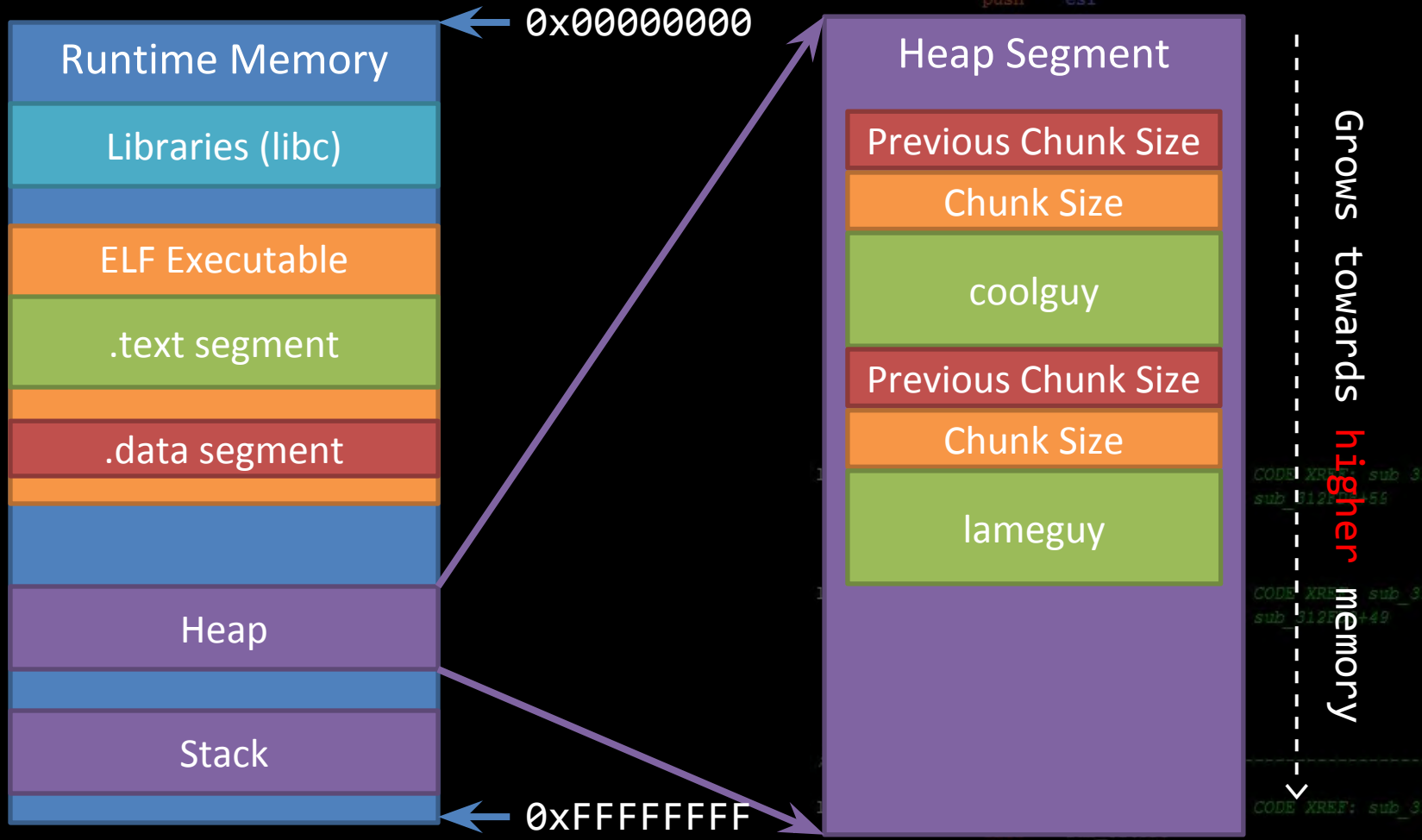
```
coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

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

printf("Input coolguy's name: ");
fgets(coolguy->buffer, 200, stdin);  // oopz...
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);
```

Heap Overflows

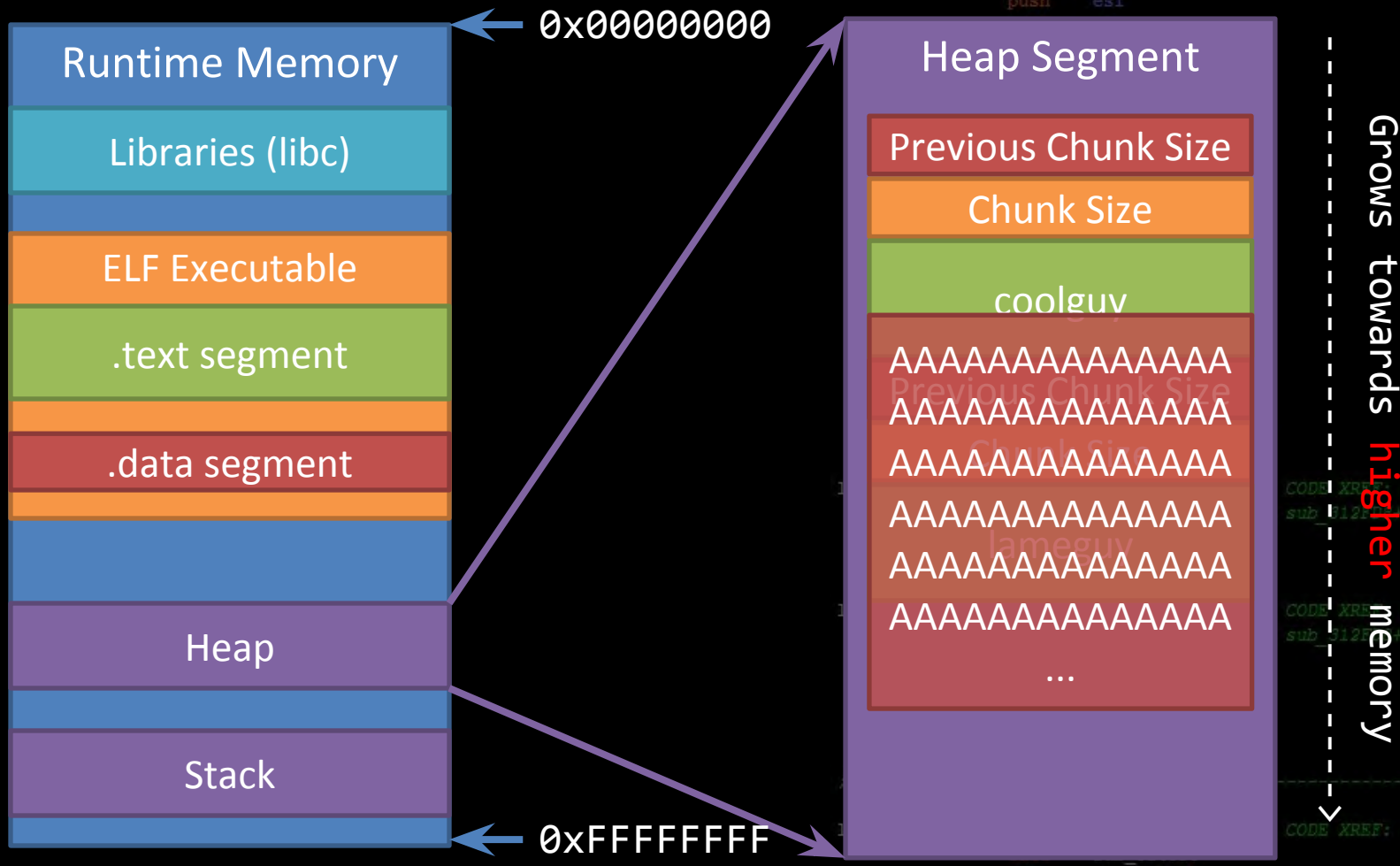
```
coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

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

printf("Input coolguy's name: ");
fgets(coolguy->buffer, 200, stdin);  // oopz...
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);
```



Heap Overflows



Heap Overflows



Heap Overflows

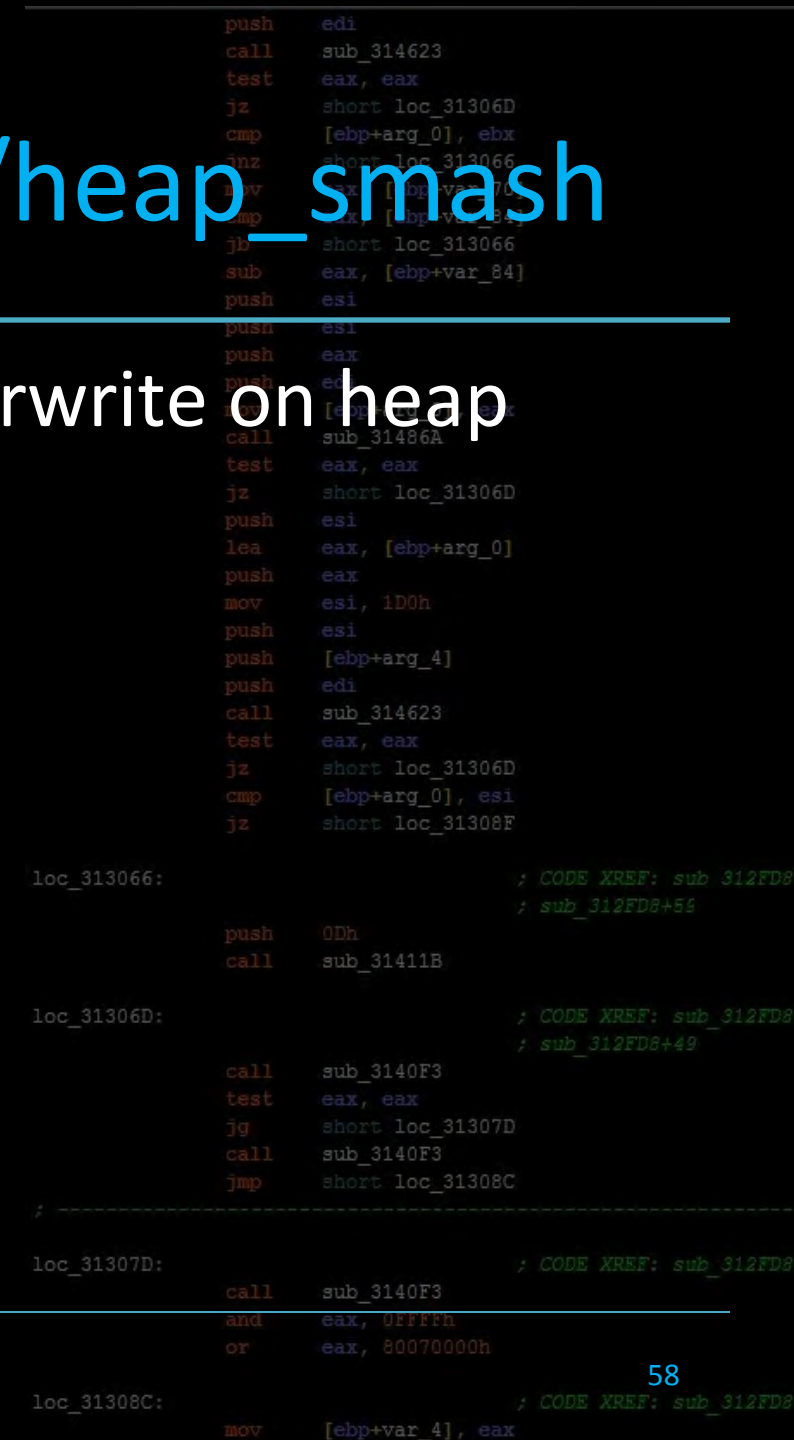
```
coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

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

printf("Input coolguy's name: ");
fgets(coolguy->buffer, 200, stdin);  // oopz...
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);
```

Silly heap overflow

Overwritten function pointer!

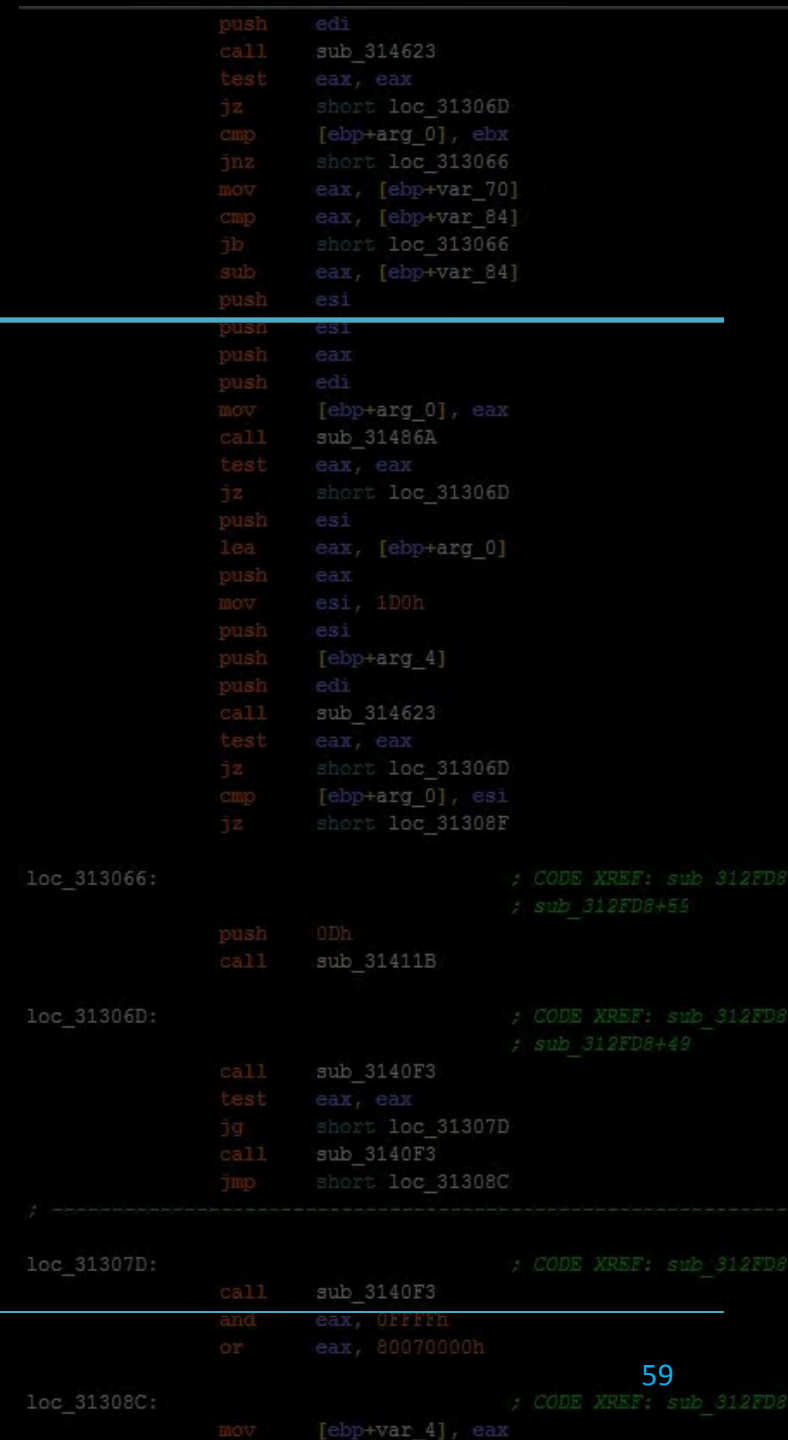
/levels/lecture/heap/heap_smash

toy function pointer overwrite on heap



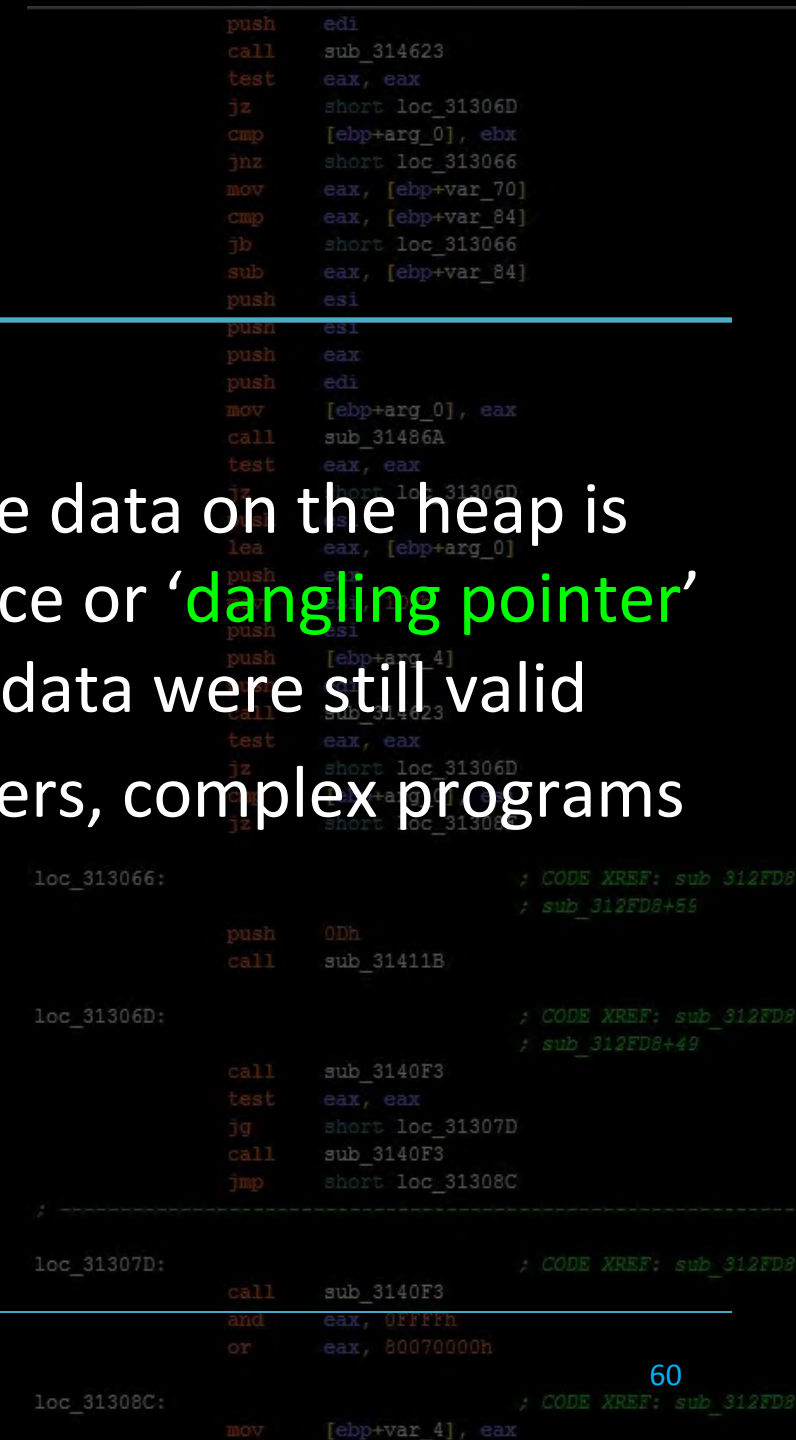
Lecture Overview

- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

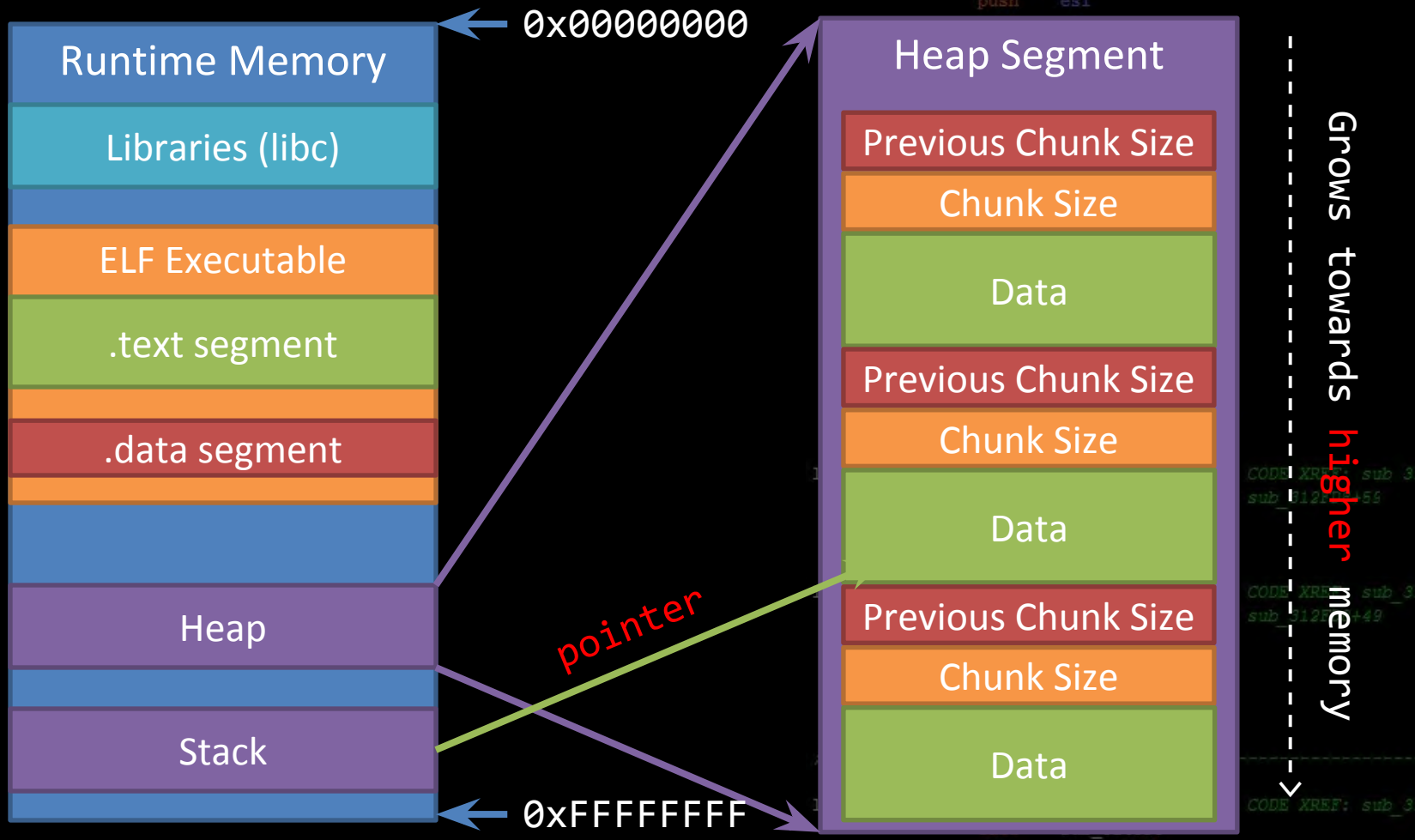


Course Terminology

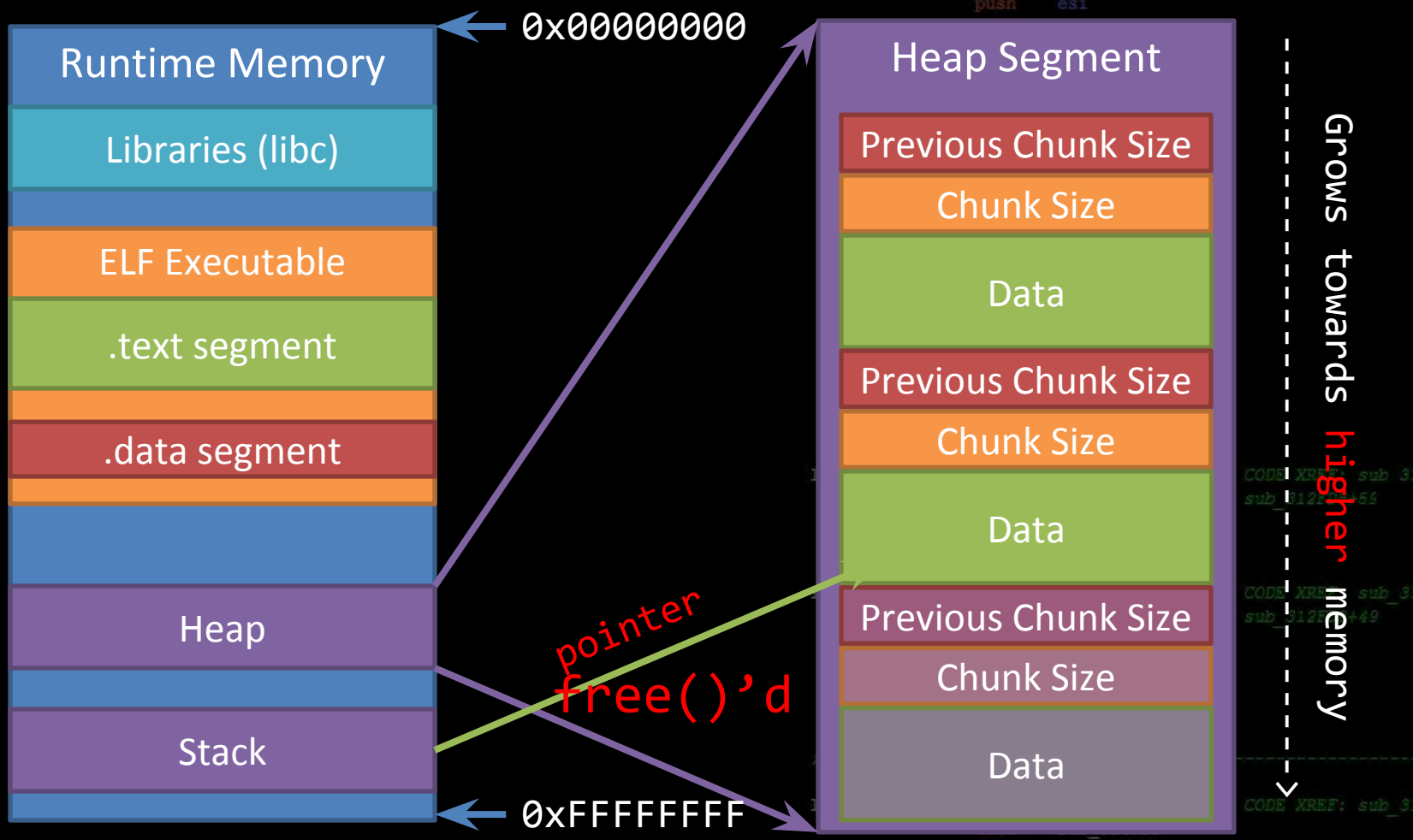
- Use After Free
  - A class of vulnerability where data on the heap is freed, but a leftover reference or 'dangling pointer' is used by the code as if the data were still valid
  - Most popular in Web Browsers, complex programs
  - Also known as UAF



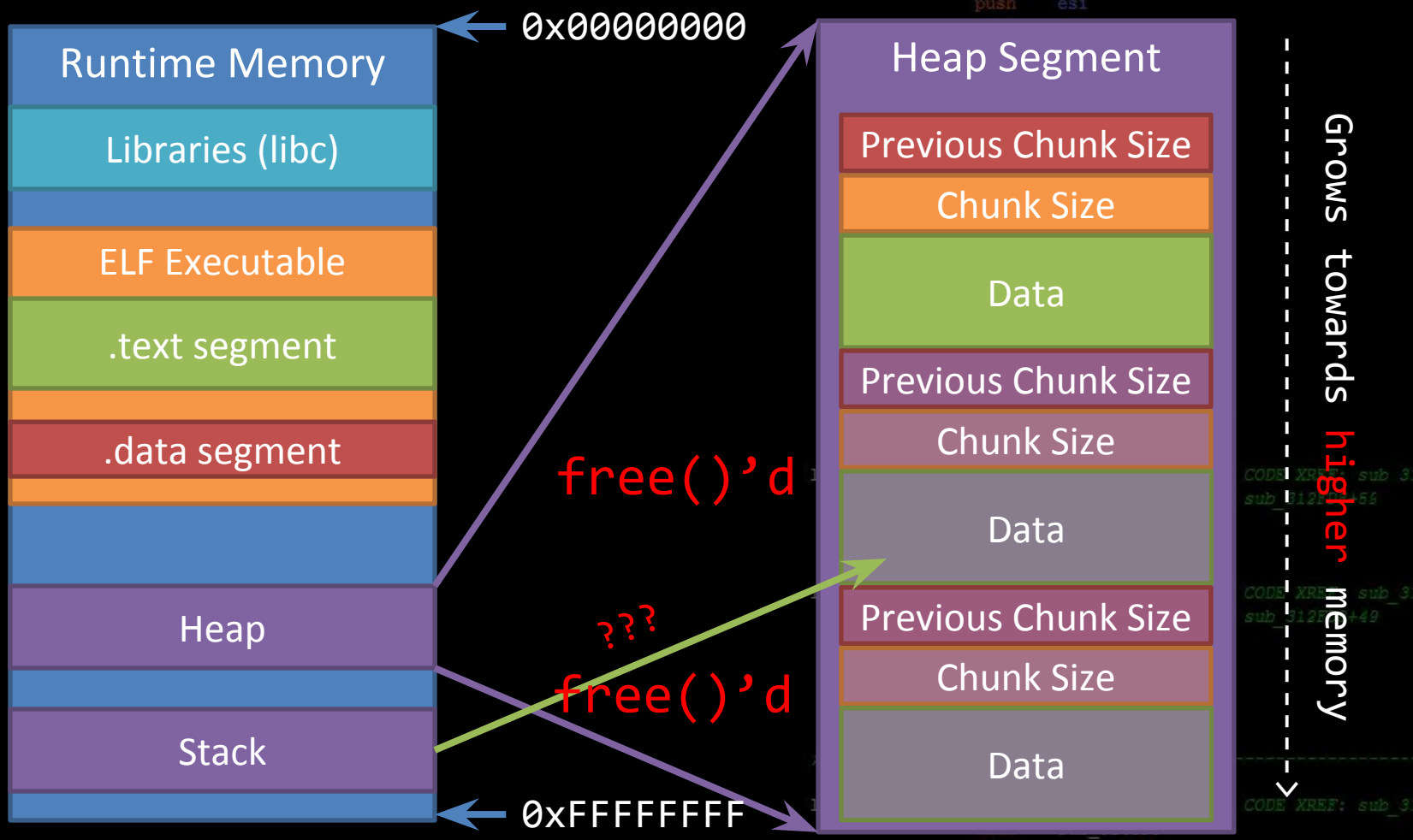
Use After Free



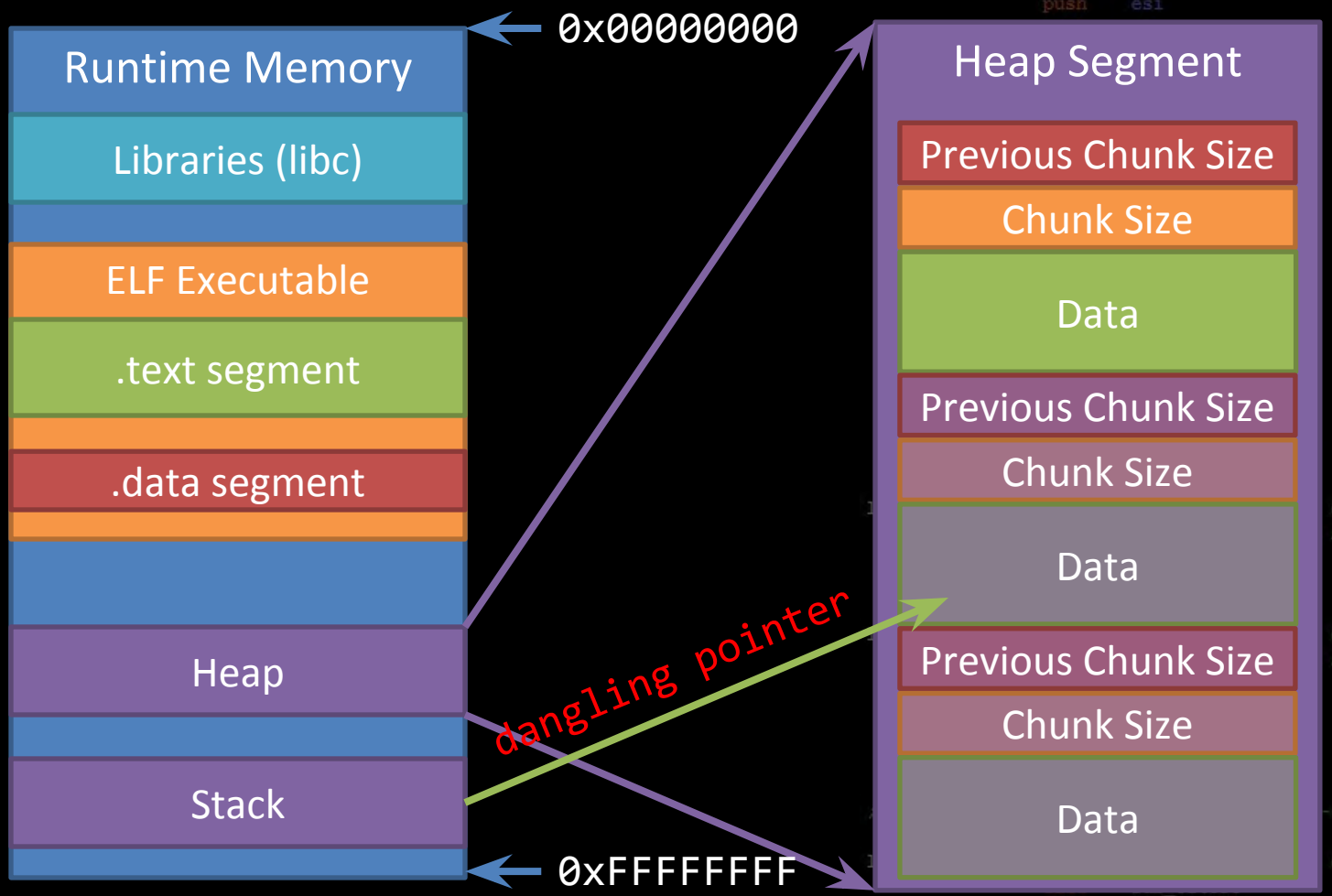
Use After Free



Use After Free

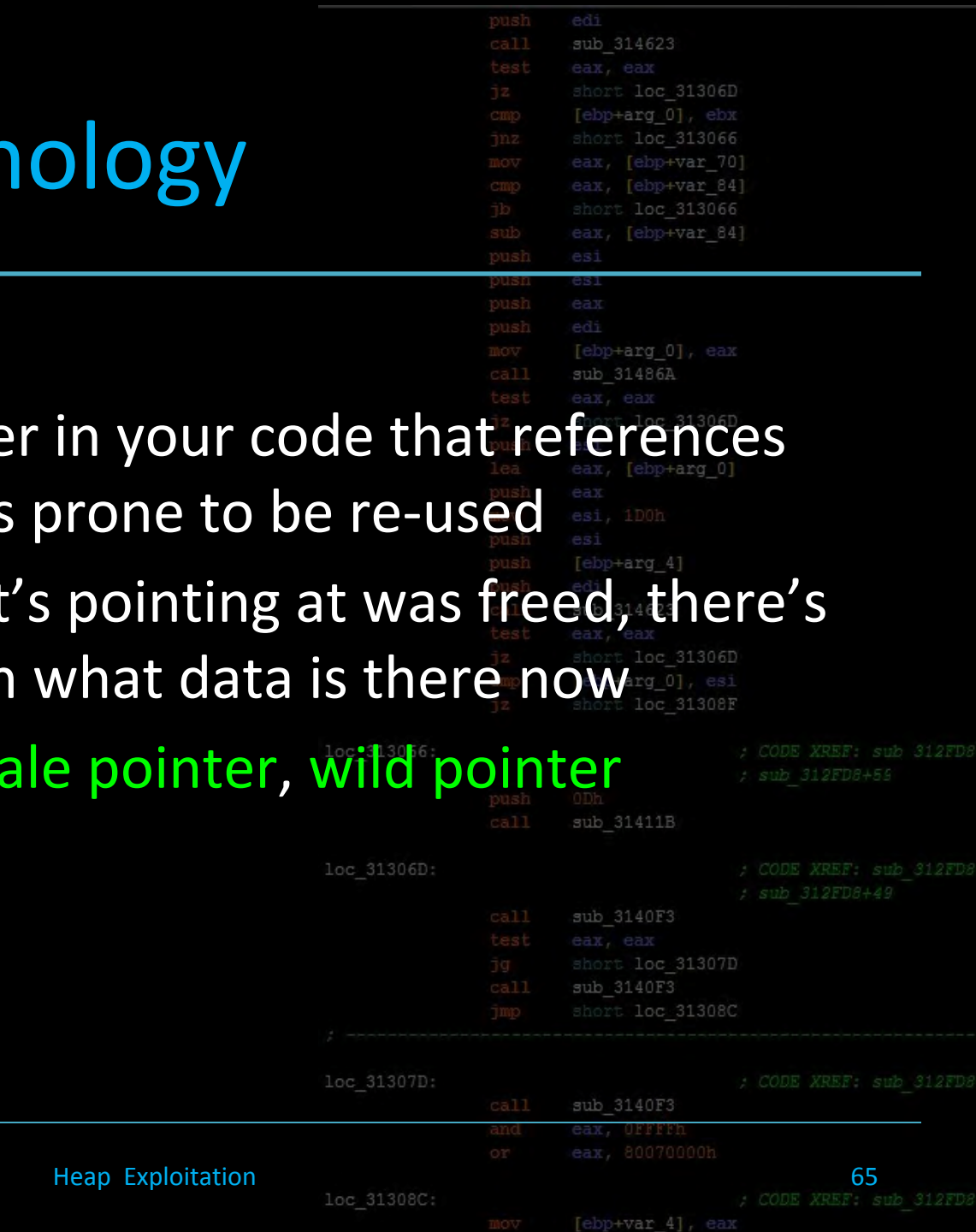


Use After Free

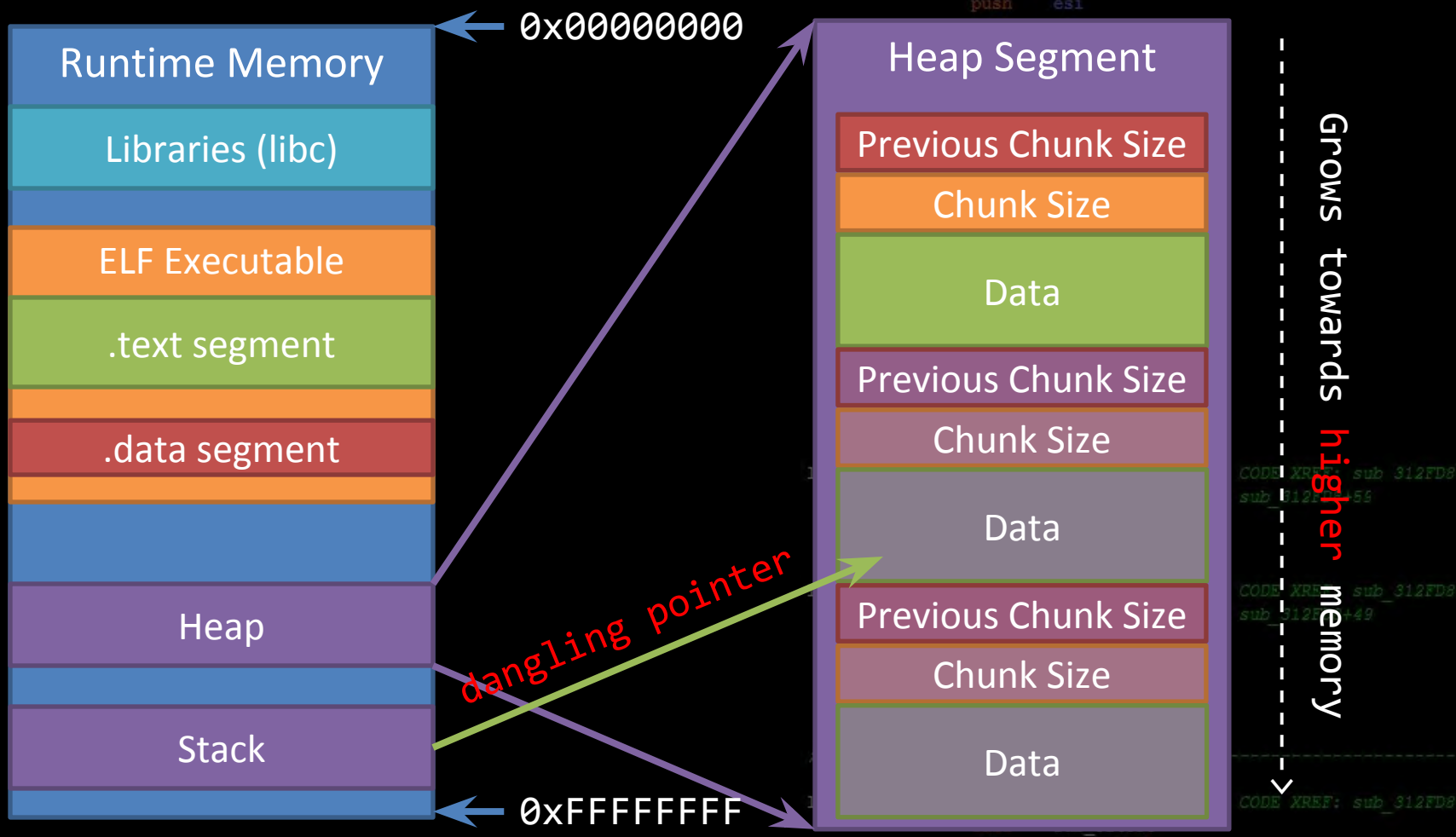


Course Terminology

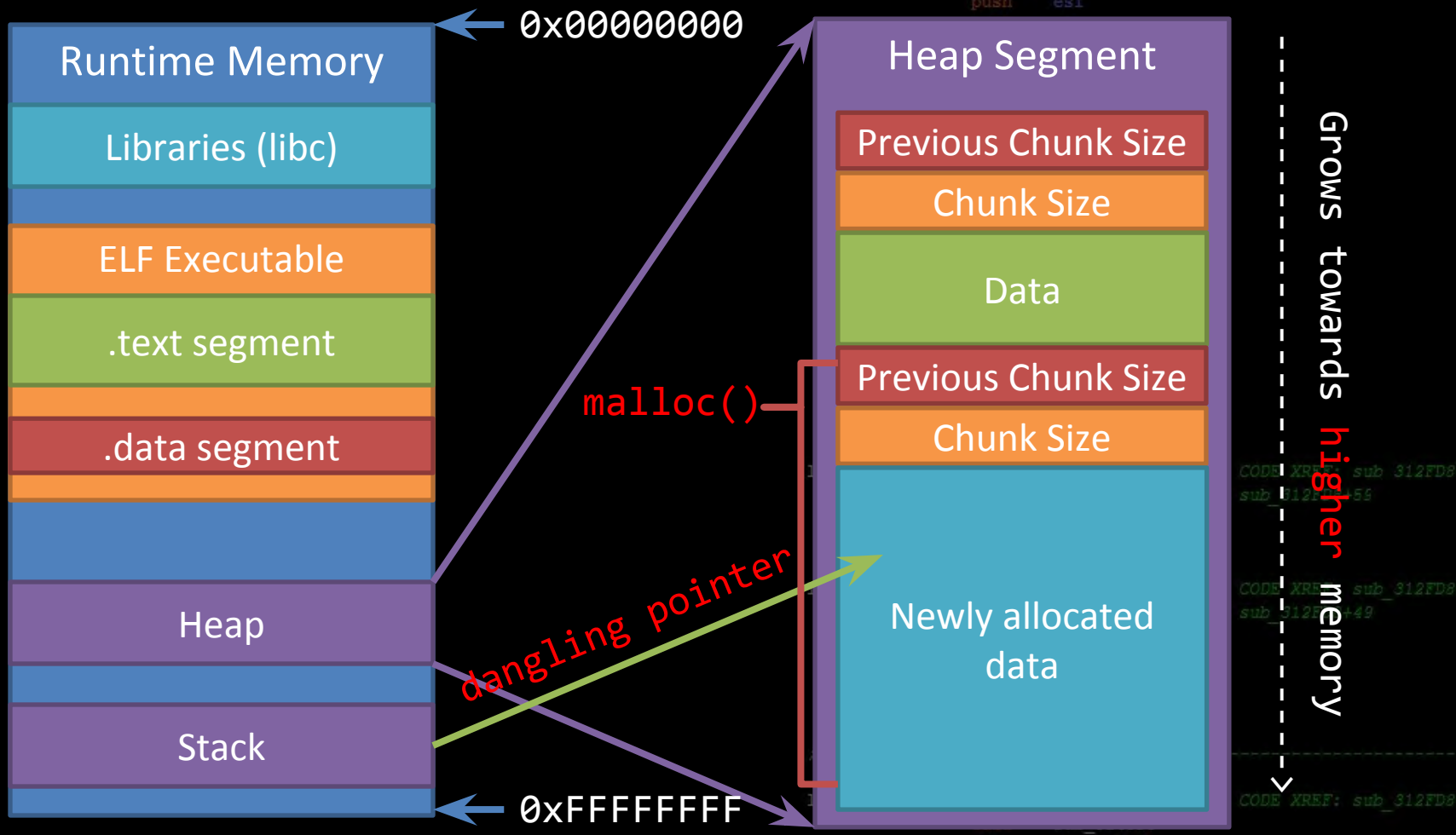
- Dangling Pointer
  - A left over pointer in your code that references free'd data and is prone to be re-used
  - As the memory it's pointing at was freed, there's no guarantees on what data is there now
  - Also known as stale pointer, wild pointer



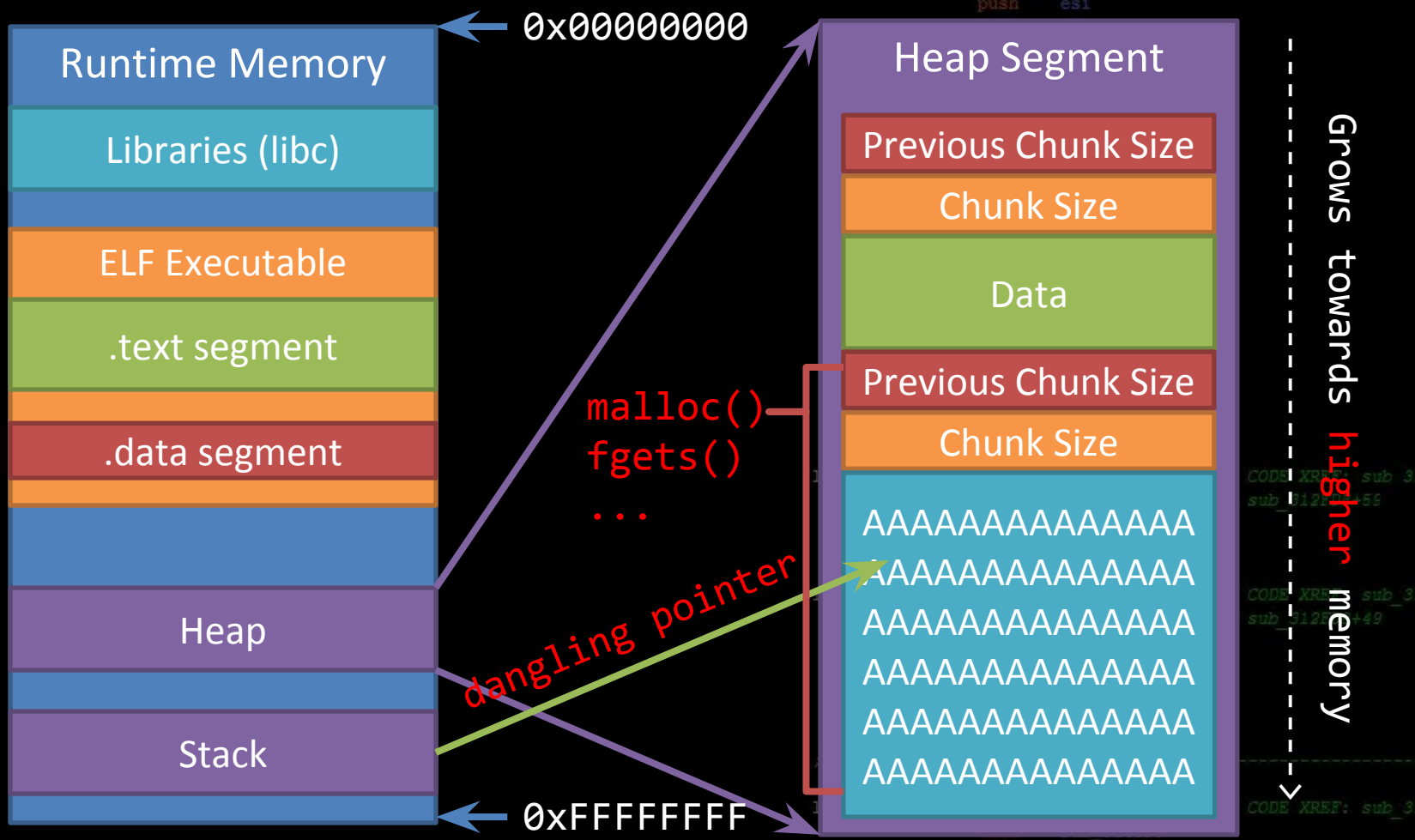
Use After Free



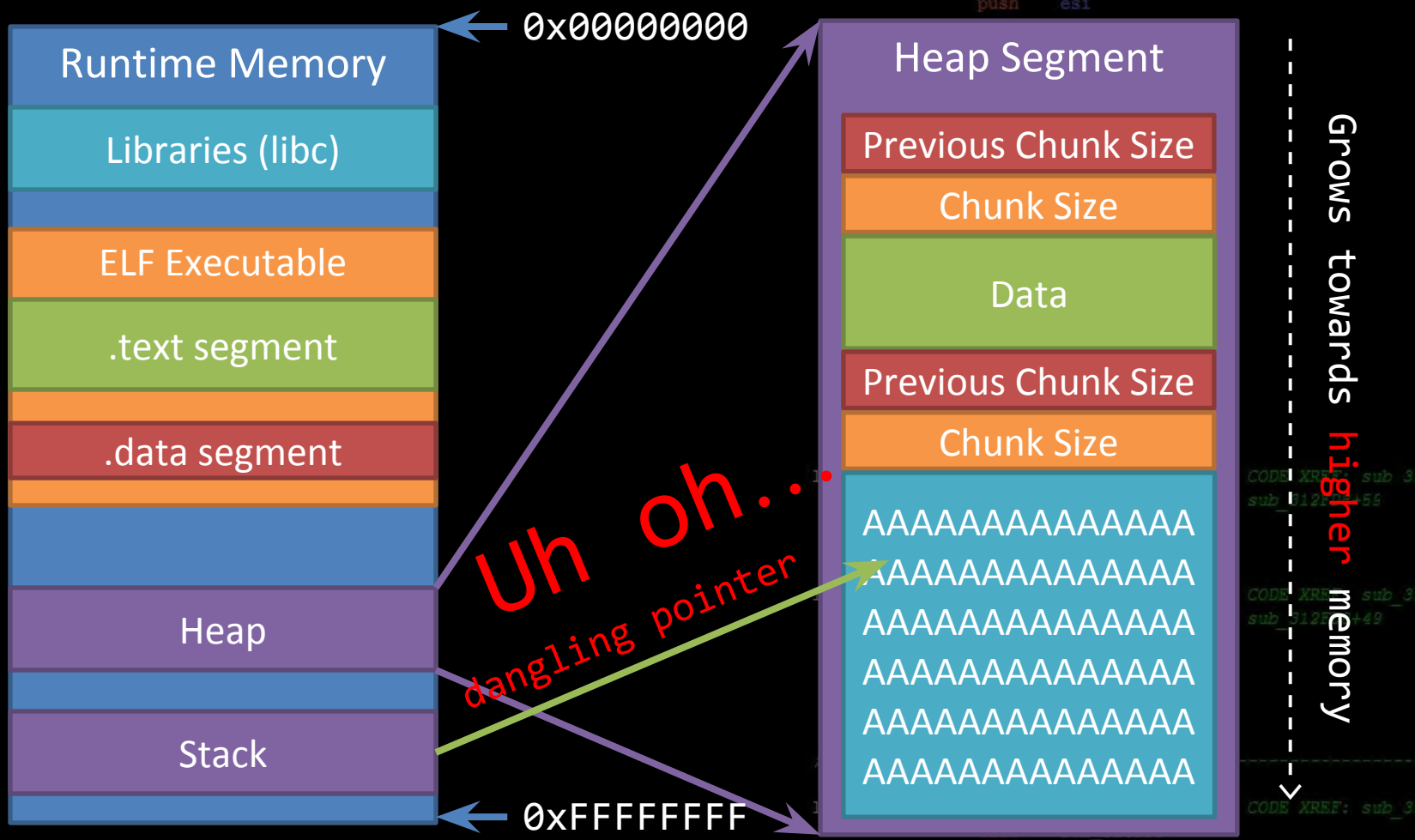
Use After Free



Use After Free



Use After Free



Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

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

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

assume dangling pointer exists

1. free()

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

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

assume dangling pointer exists

1. free()

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

2. malloc()

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

assume dangling pointer exists

1. free()

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

2. malloc()

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

3. Set favorite_num = 0x41414141

Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

assume dangling pointer exists

1. free()

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

2. malloc()

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

3. Set favorite_num = 0x41414141

4. Force dangling pointer to call 'message()'

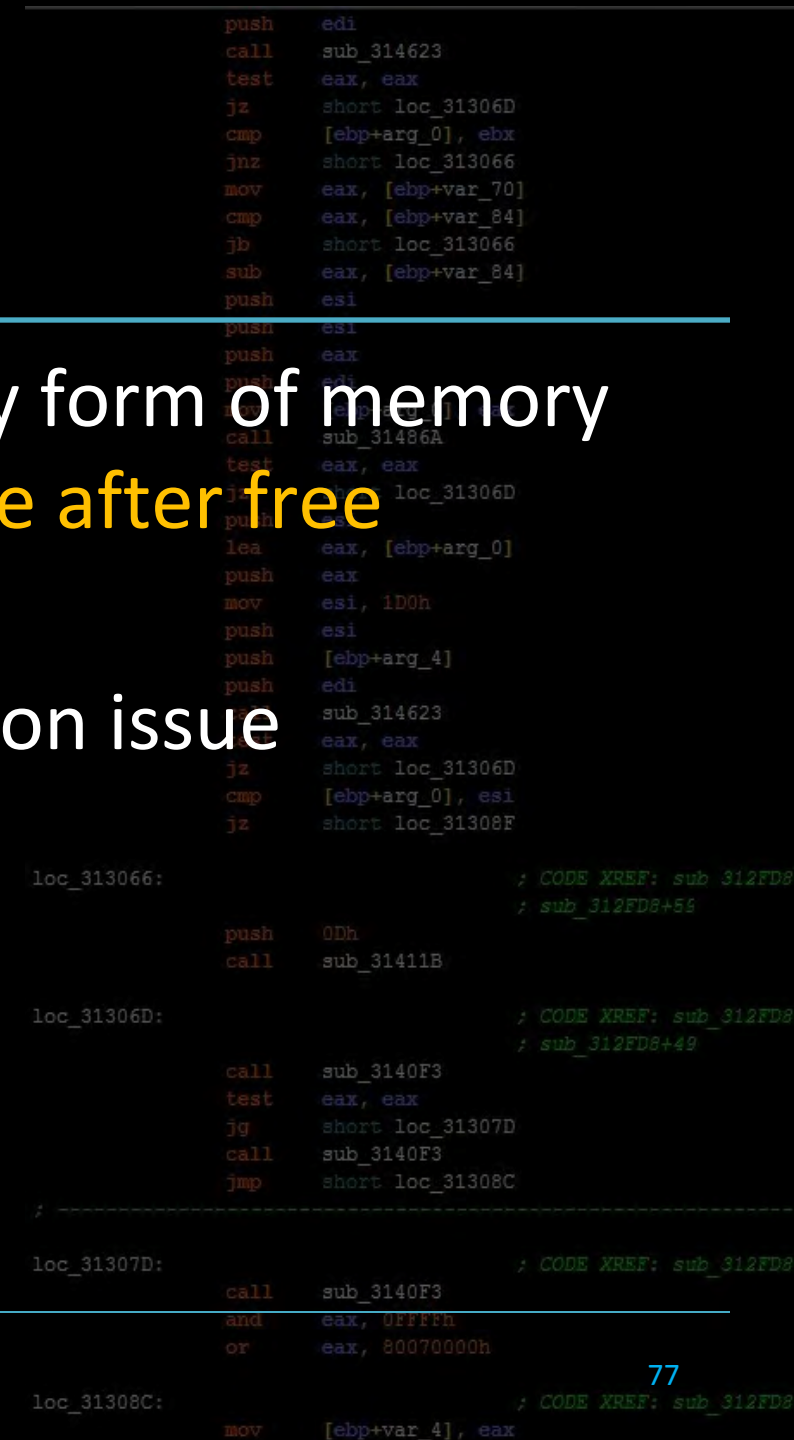
/levels/lecture/heap/heap_uaf

your very first use after free!

```
                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
                lea     eax, [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+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
```

Use After Free

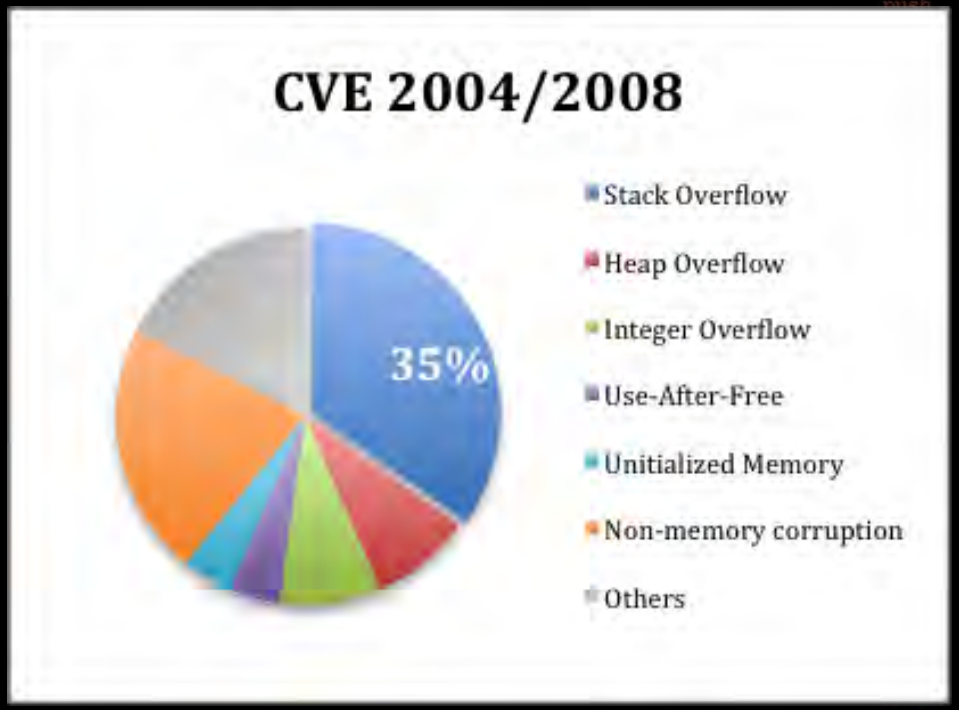
- You actually don't need any form of memory corruption to leverage a use after free
- It's simply an implementation issue
  - pointer mismanagement



UAF in the Wild

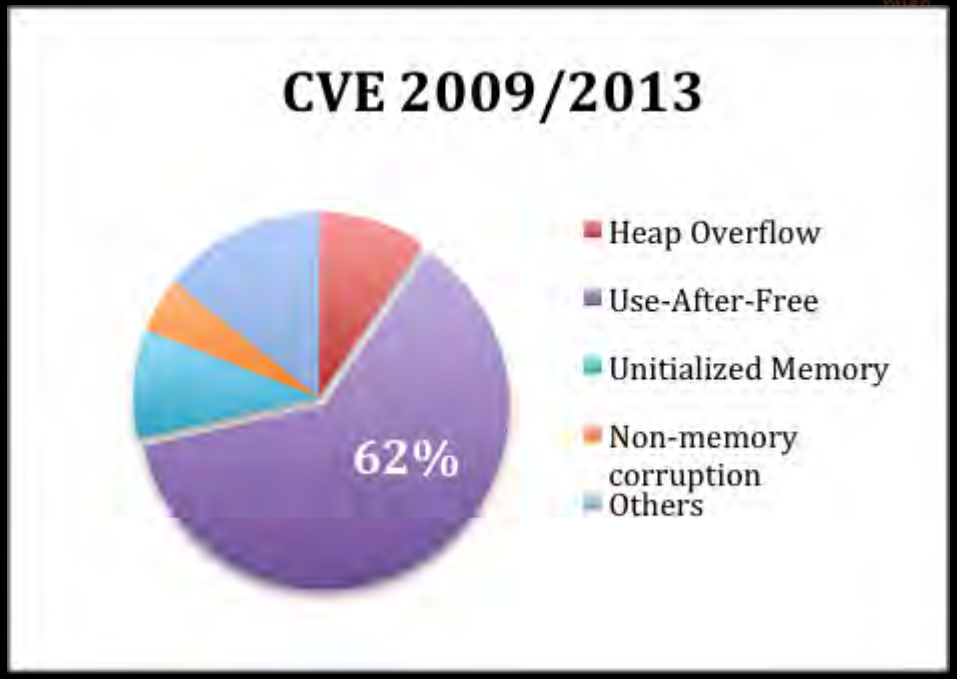
- The 'hot' vulnerability nowadays, almost every modern browser exploit leverages a UAF

IE CVE Statistics



http://blog.tempest.com.br/breno-cunha/perspectives-on-exploit-development-and-cyber-attacks.html

IE CVE Statistics



http://blog.tempest.com.br/breno-cunha/perspectives-on-exploit-development-and-cyber-attacks.html

UAF in the Wild

- The 'hot' vulnerability nowadays, almost every modern browser exploit leverages a UAF

- Why are they so well liked?

UAF in the Wild

- The 'hot' vulnerability nowadays, almost every modern browser exploit leverages a UAF

- Why are they so well liked?
  - Doesn't require any memory corruption to use

UAF in the Wild

- The 'hot' vulnerability nowadays, almost every modern browser exploit leverages a UAF

- Why are they so well liked?
  - Doesn't require any memory corruption to use
  - Can be used for info leaks

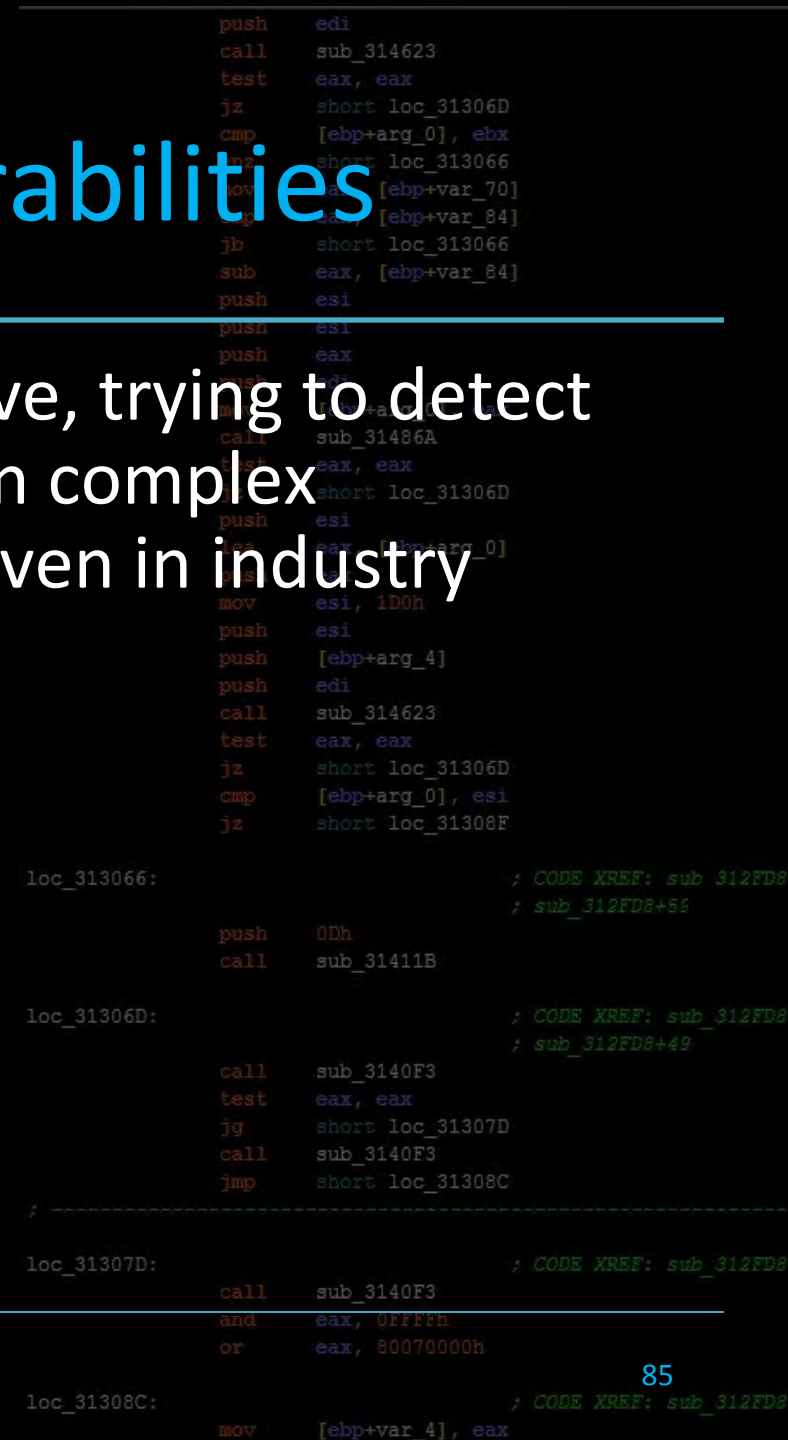
UAF in the Wild

- The 'hot' vulnerability nowadays, almost every modern browser exploit leverages a UAF

- Why are they so well liked?
  - Doesn't require any memory corruption to use
  - Can be used for info leaks
  - Can be used to trigger memory corruption or get control of EIP

Detecting UAF Vulnerabilities

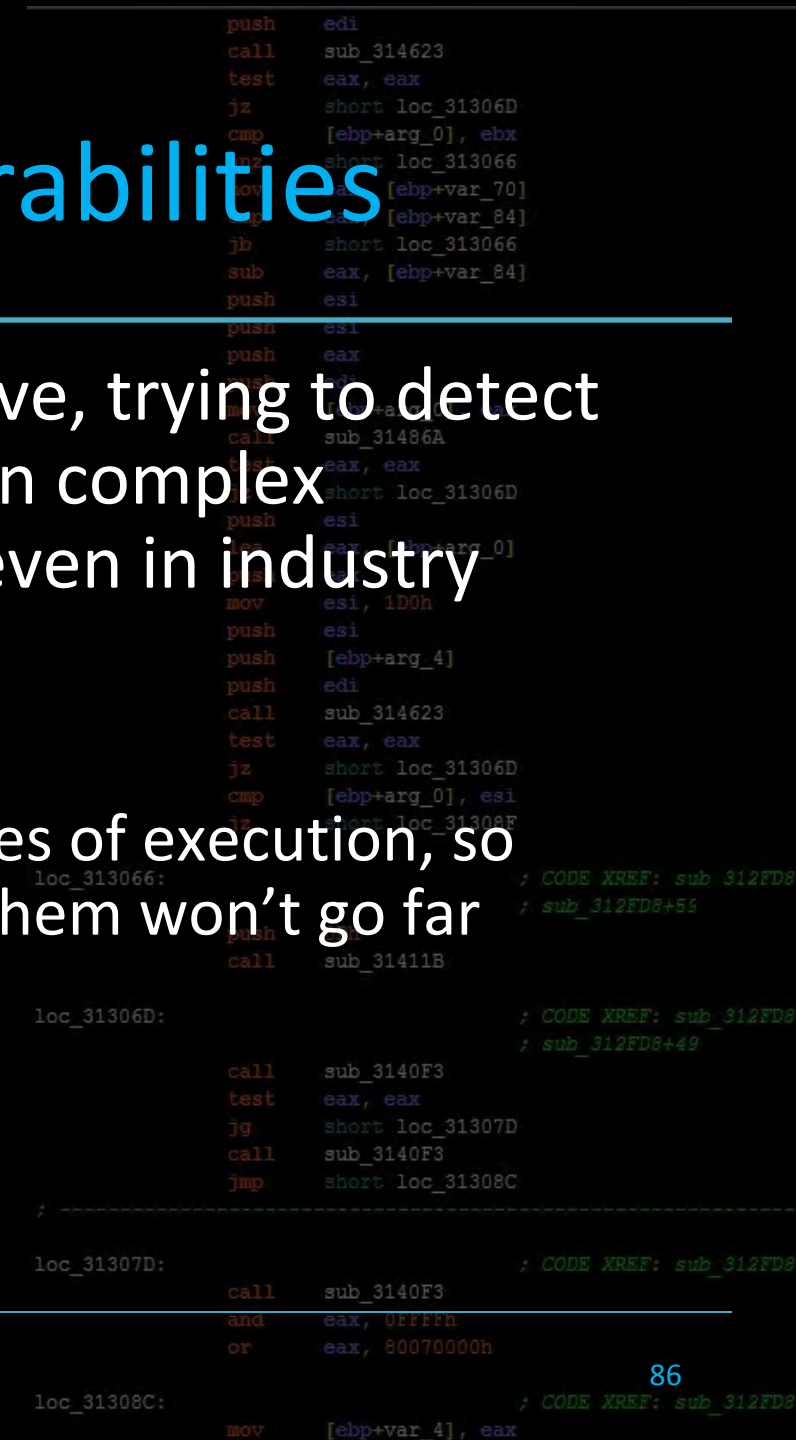
- From the defensive perspective, trying to detect use after free vulnerabilities in complex applications is very difficult, even in industry
- Why?



Detecting UAF Vulnerabilities

- From the defensive perspective, trying to detect use after free vulnerabilities in complex applications is very difficult, even in industry

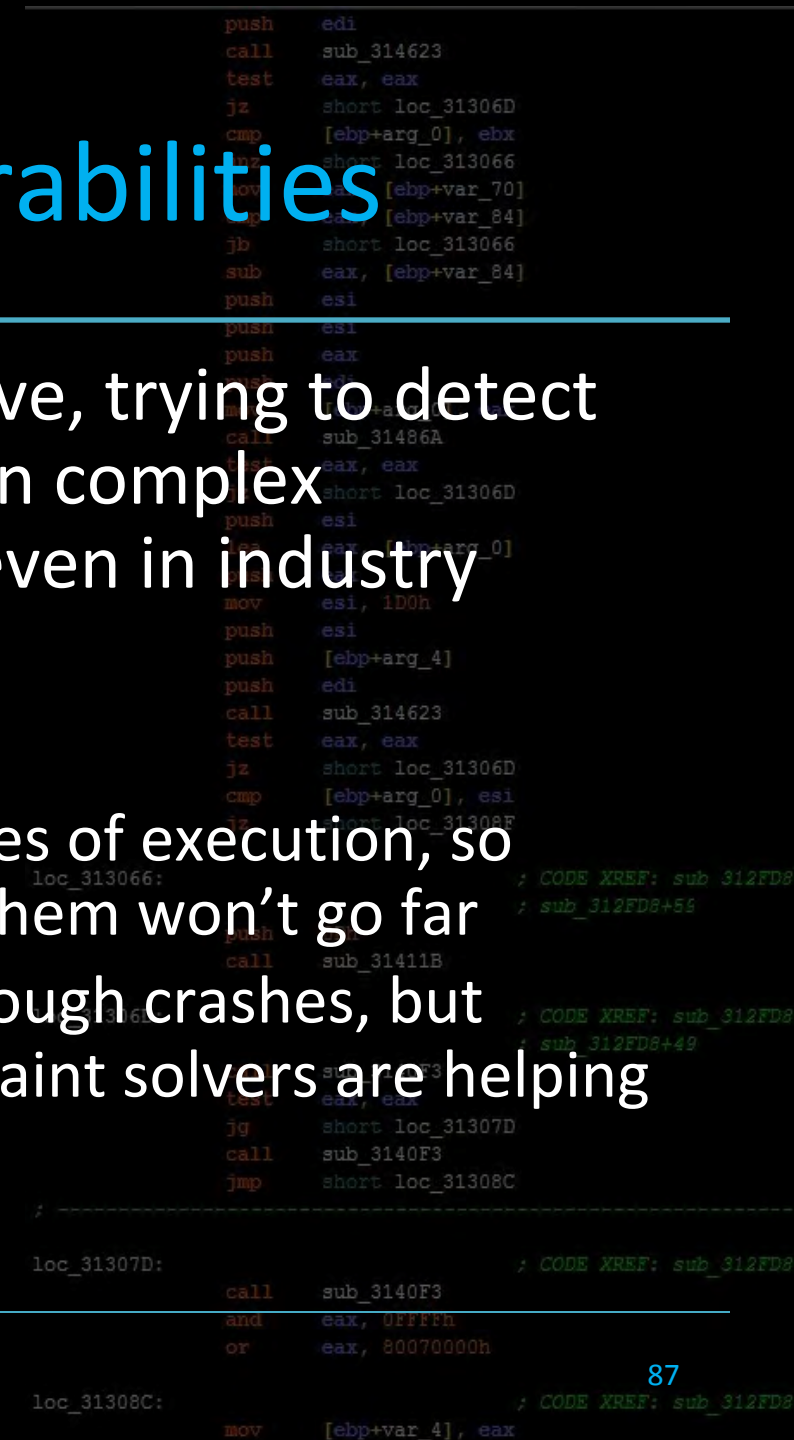
- Why?
  - UAF's only exist in certain states of execution, so statically scanning source for them won't go far



Detecting UAF Vulnerabilities

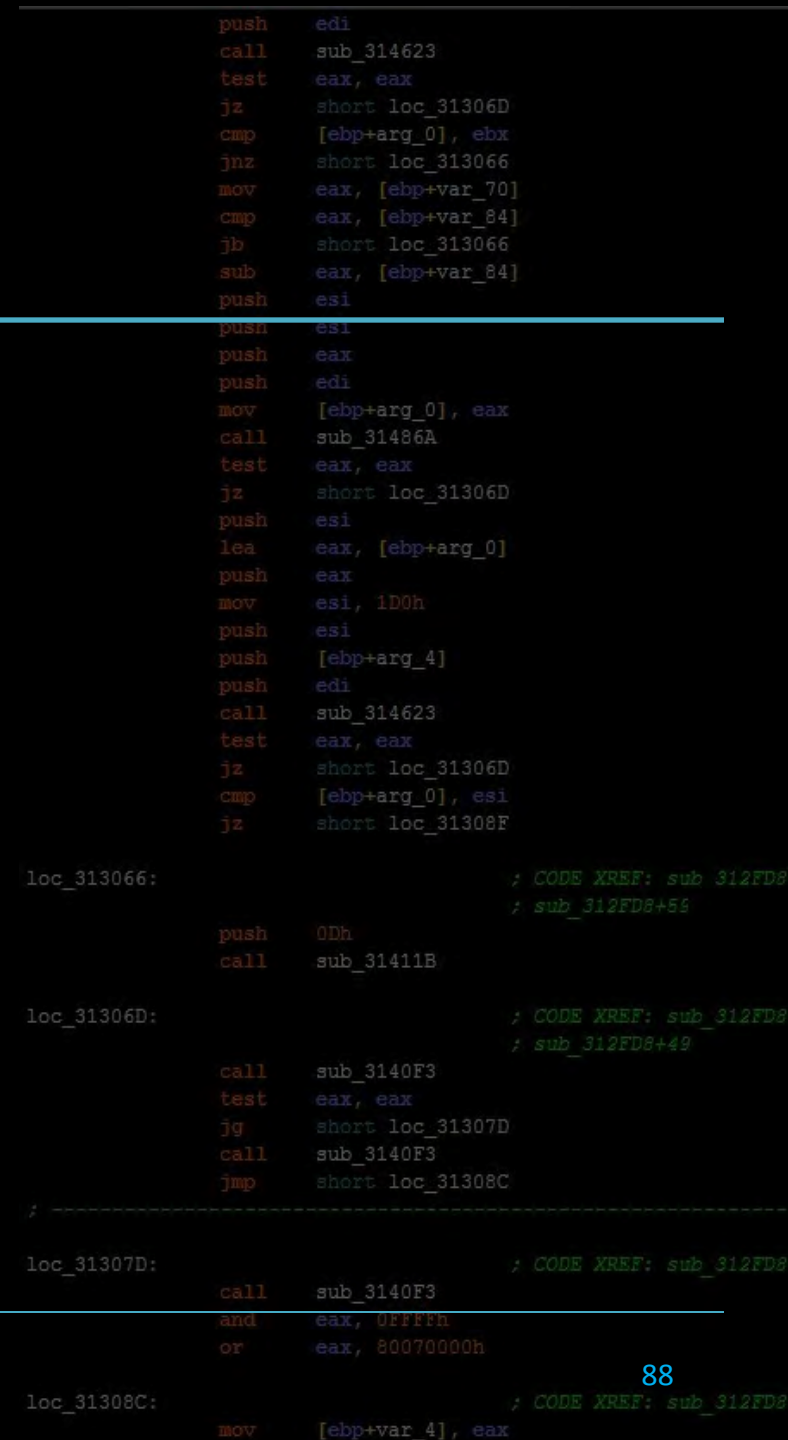
- From the defensive perspective, trying to detect use after free vulnerabilities in complex applications is very difficult, even in industry

- Why?
  - UAF's only exist in certain states of execution, so statically scanning source for them won't go far
  - They're usually only found through crashes, but symbolic execution and constraint solvers are helping find these bugs faster



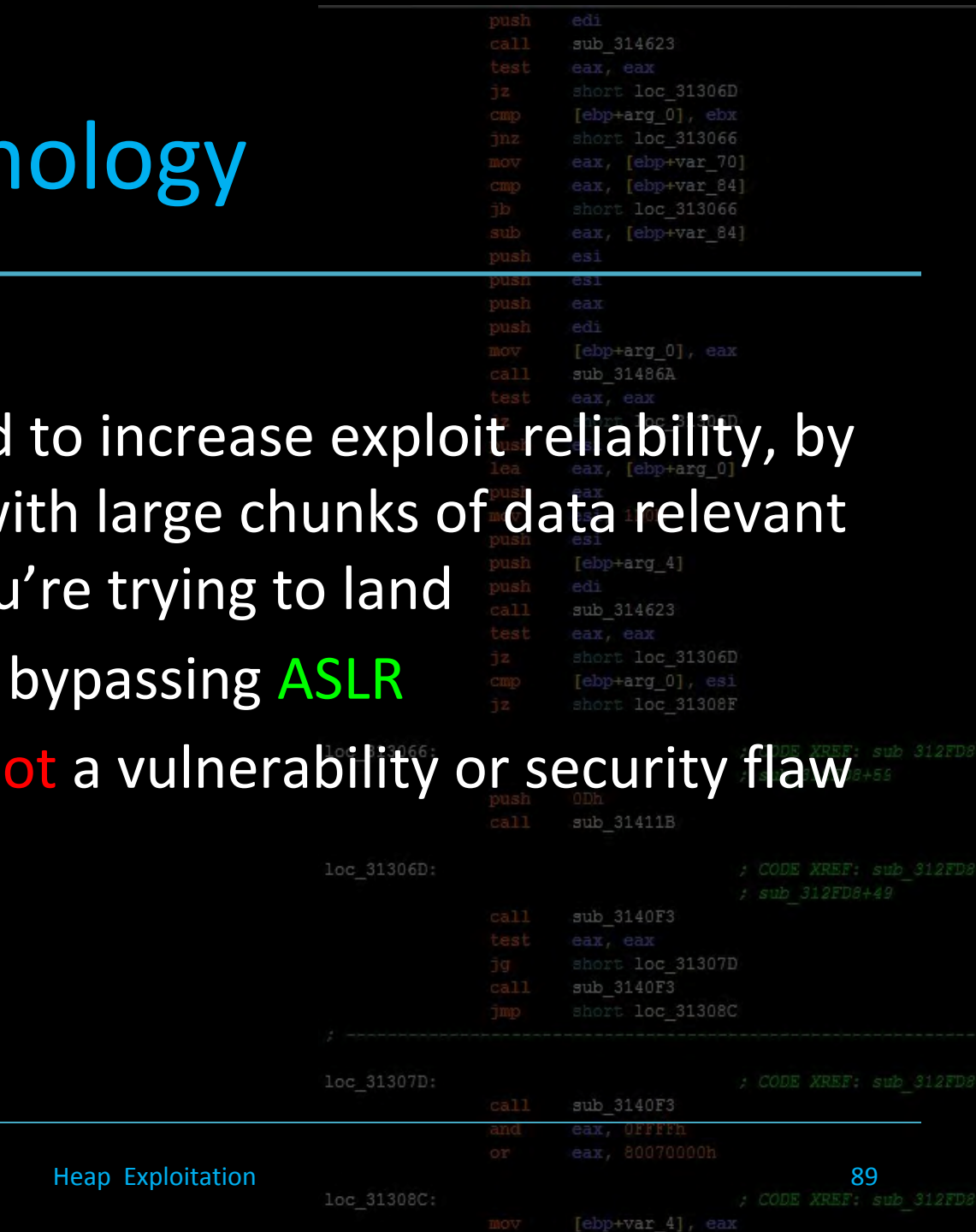
Lecture Overview

- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

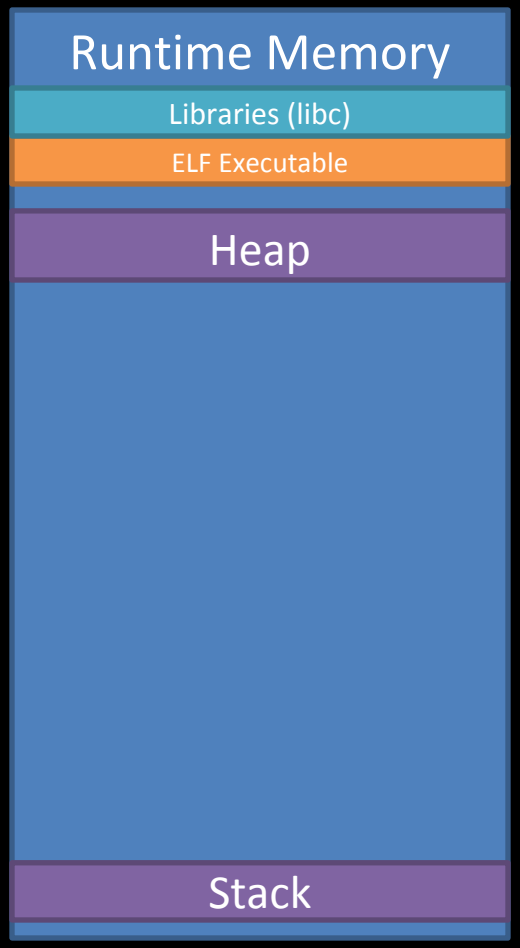


Course Terminology

- Heap Spraying
  - A technique used to increase exploit reliability, by filling the heap with large chunks of data relevant to the exploit you're trying to land
  - It can assist with bypassing ASLR
  - A heap spray is not a vulnerability or security flaw



Heap Spray in Action



0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

0x09104000 – Top of heap

```
filler = "AAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Heap Spray in Action

0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

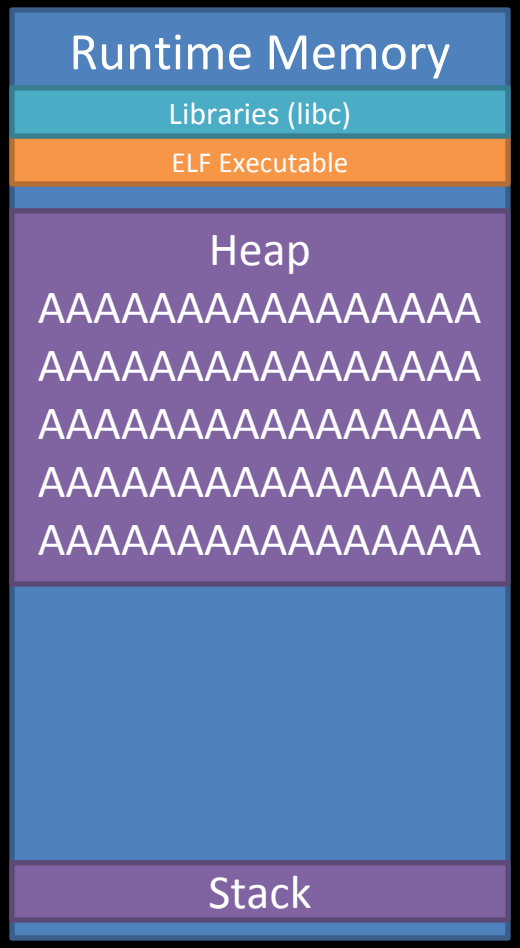
0x09104000 – Top of heap

```
filler = "AAAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Heap Spray in Action



0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

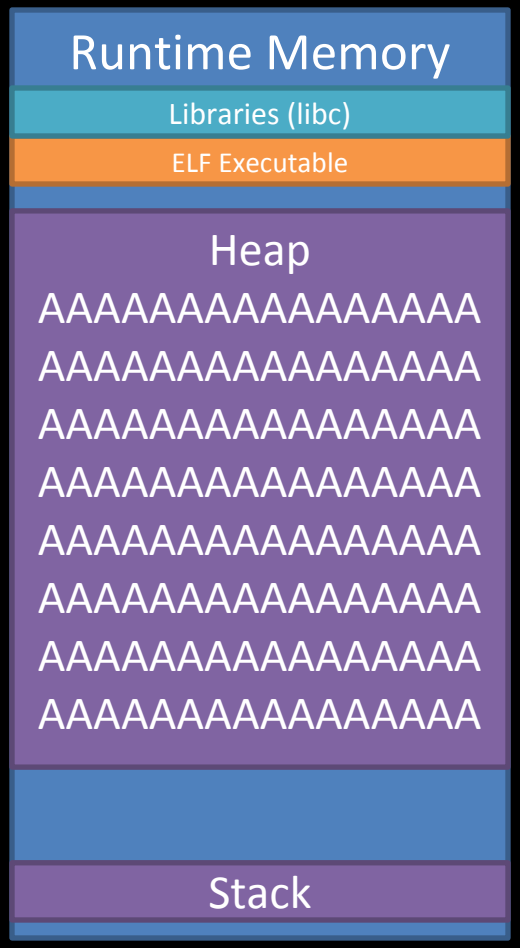
0x09104000 – Top of heap

```
filler = "AAAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Heap Spray in Action



0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

0x09104000 – Top of heap

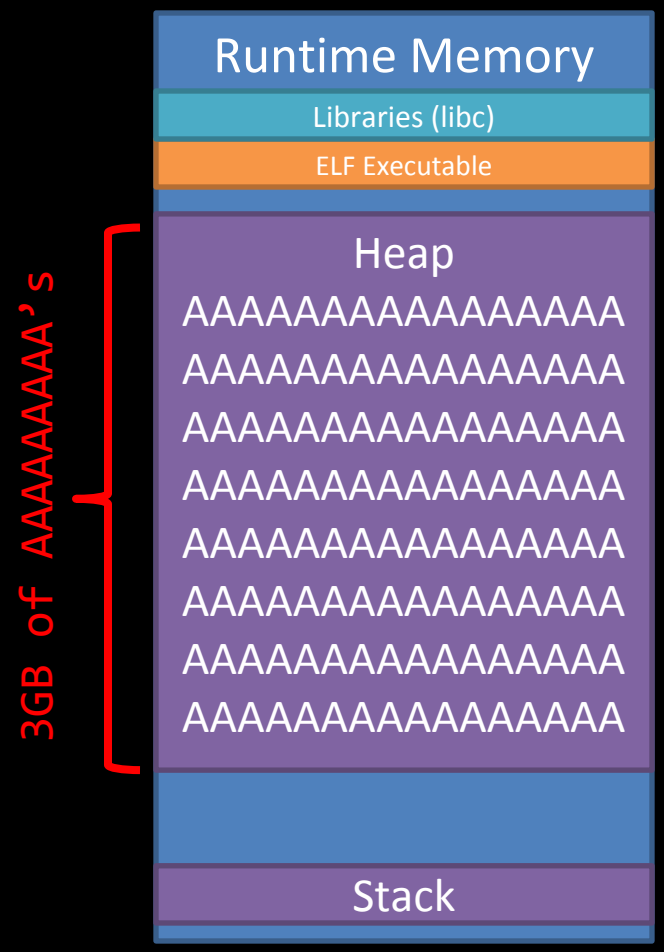
```
filler = "AAAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```

0xbbe09e00 – bottom of heap

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Heap Spray in Action



0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

0x09104000 – Top of heap

```
filler = "AAAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```

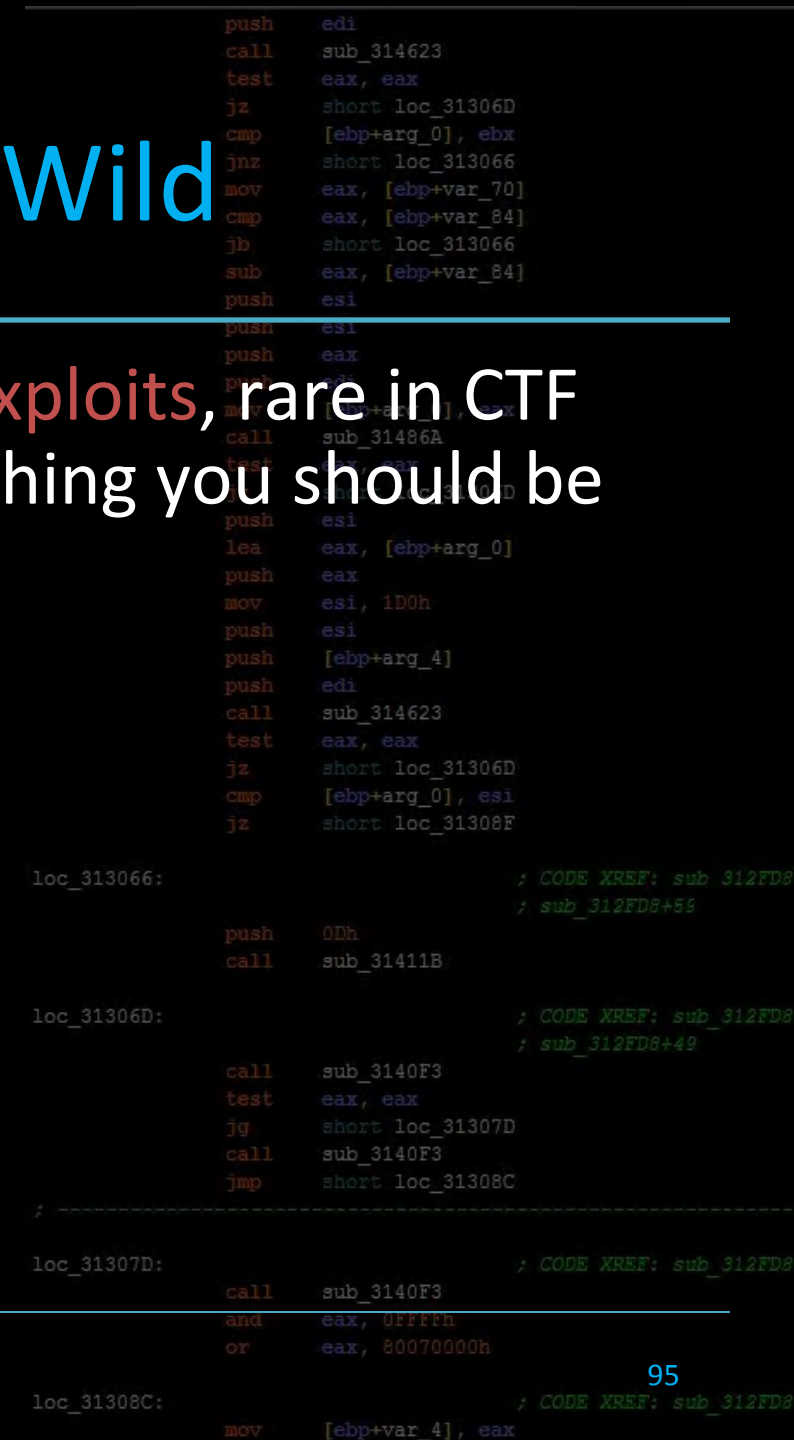
0xbbe09e00 – bottom of heap

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Heap Spraying in the Wild

- Generally found in browser exploits, rare in CTF and wargames but still something you should be aware of



Heap Spraying in the Wild

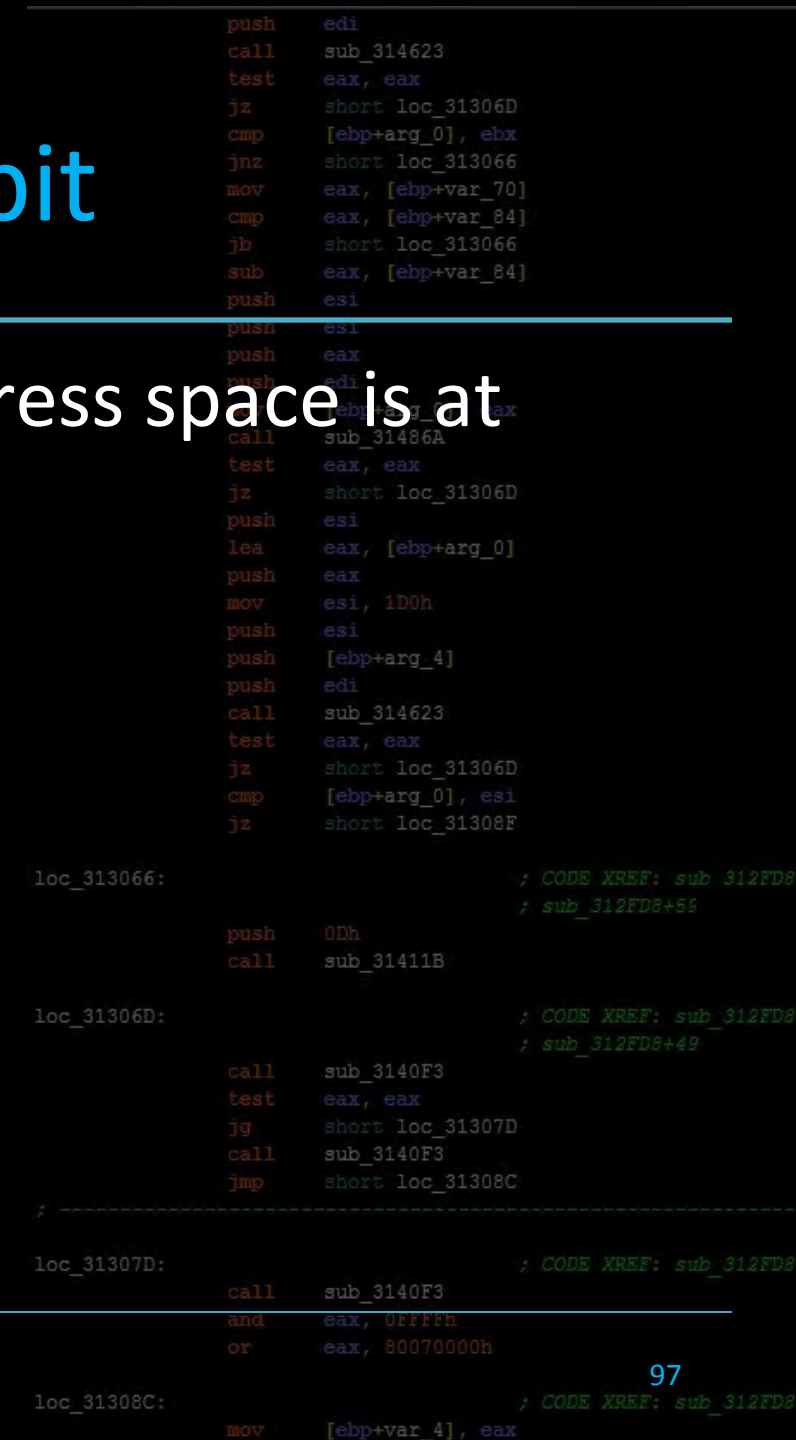
- Generally found in browser exploits, rare in CTF and wargames but still something you should be aware of

- Usually heap sprays are done in something like javascript placed on a malicious html page

```
memory = new Array();
for(i = 0; i < 0x100; i++)
     memory[i] = ROPNOP + ROP;
```

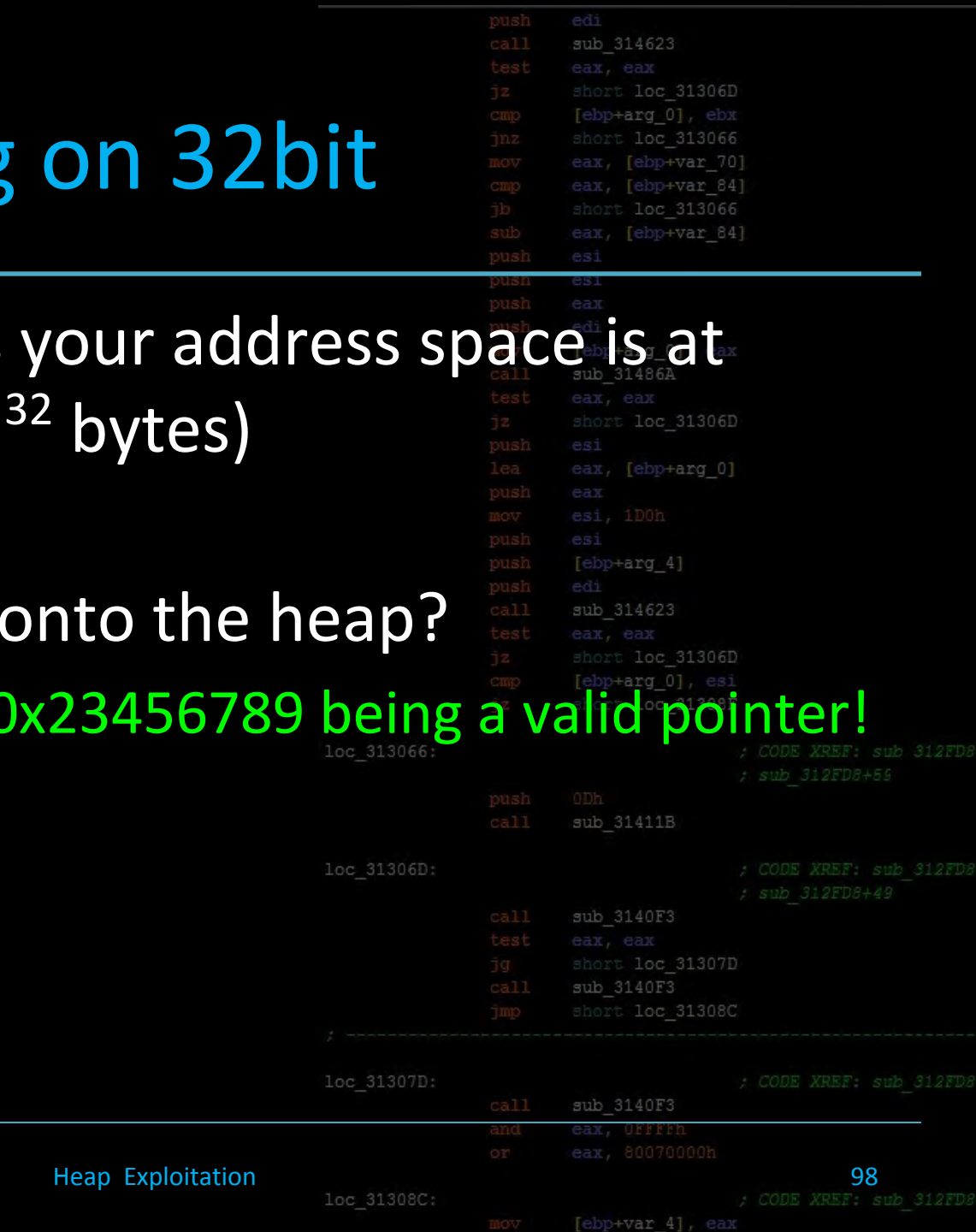
Heap Spraying on 32bit

- On 32bit systems your address space is at maximum 4GB (2^{32} bytes)



Heap Spraying on 32bit

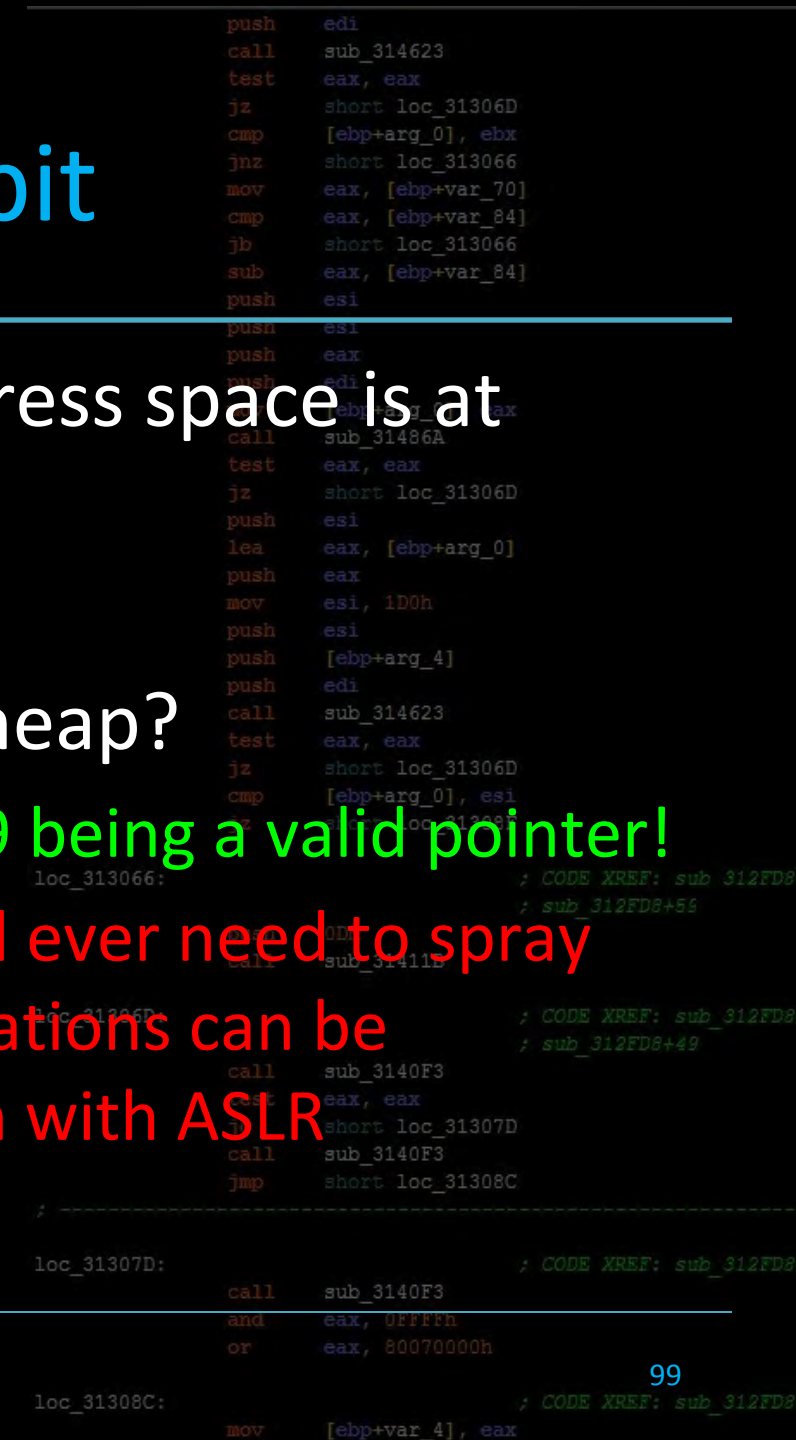
- On 32bit systems your address space is at maximum 4GB (2^{32} bytes)
- Spray 3GB of A's onto the heap?
  - +75% chance of 0x23456789 being a valid pointer!



Heap Spraying on 32bit

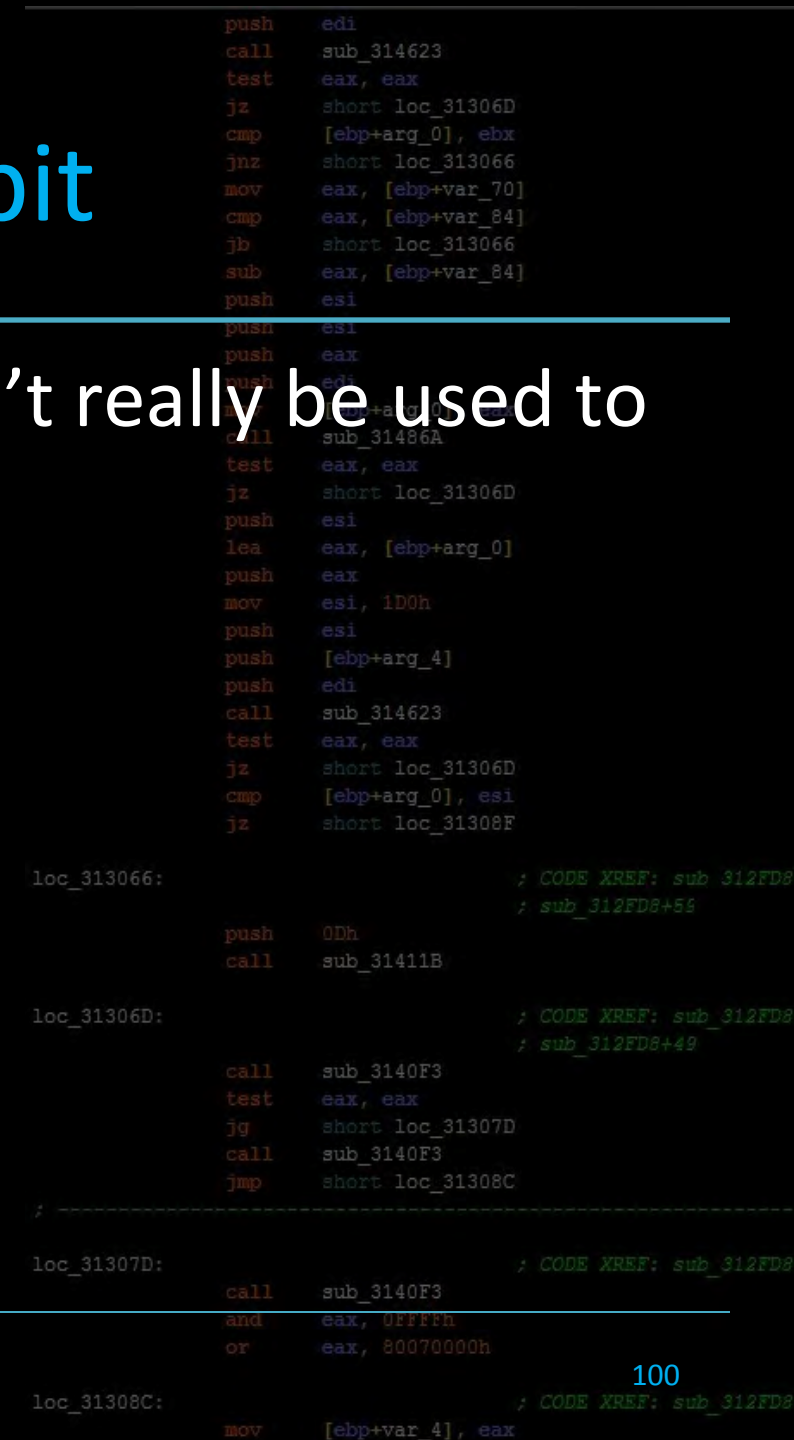
- On 32bit systems your address space is at maximum 4GB (2^{32} bytes)

- Spray 3GB of A's onto the heap?
  - +75% chance of 0x23456789 being a valid pointer!
  - Note: It's unlikely you would ever need to spray 3GB of anything as heap locations can be somewhat predictable, even with ASLR



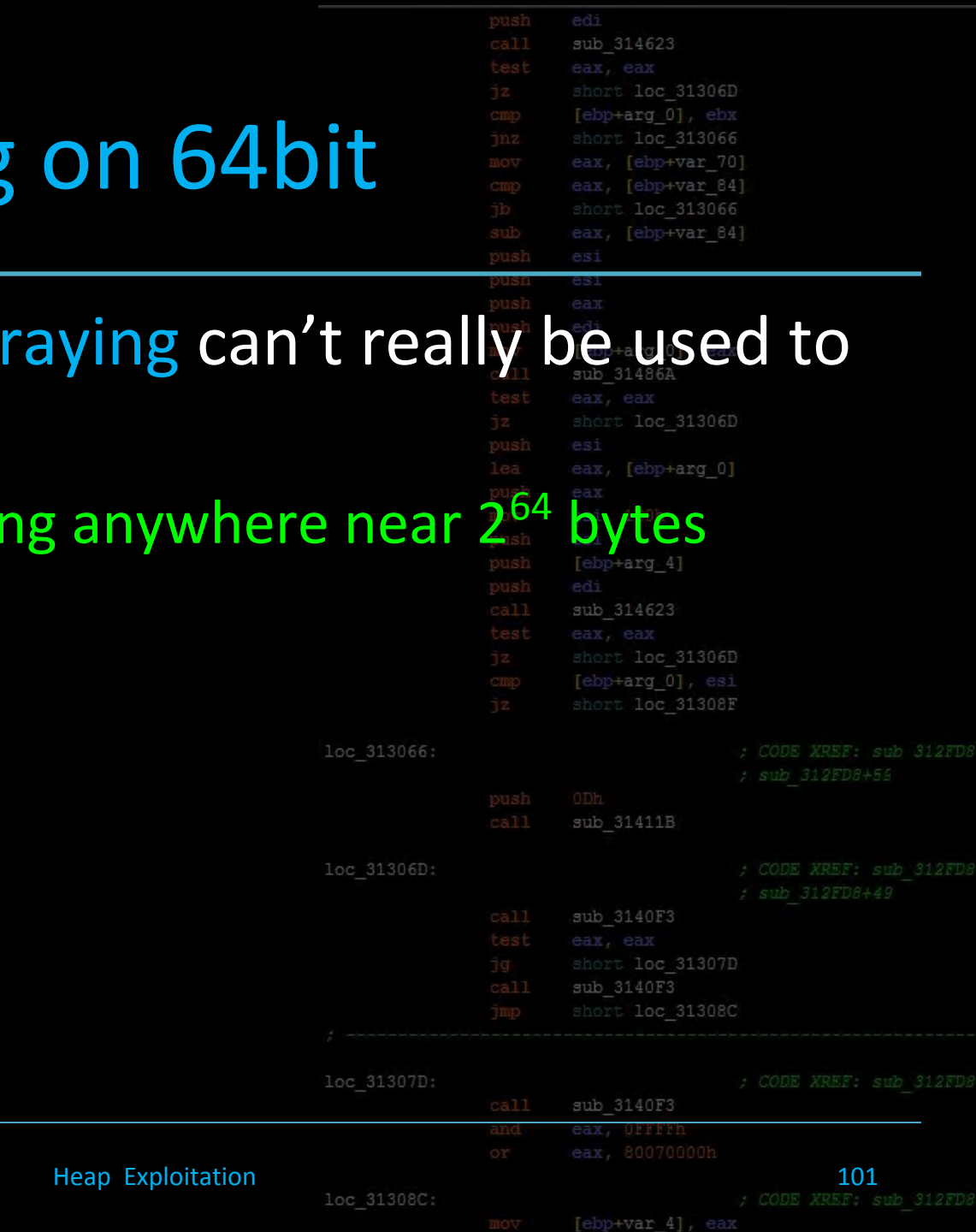
Heap Spraying on 64bit

- On 64bit heap spraying can't really be used to bypass ASLR



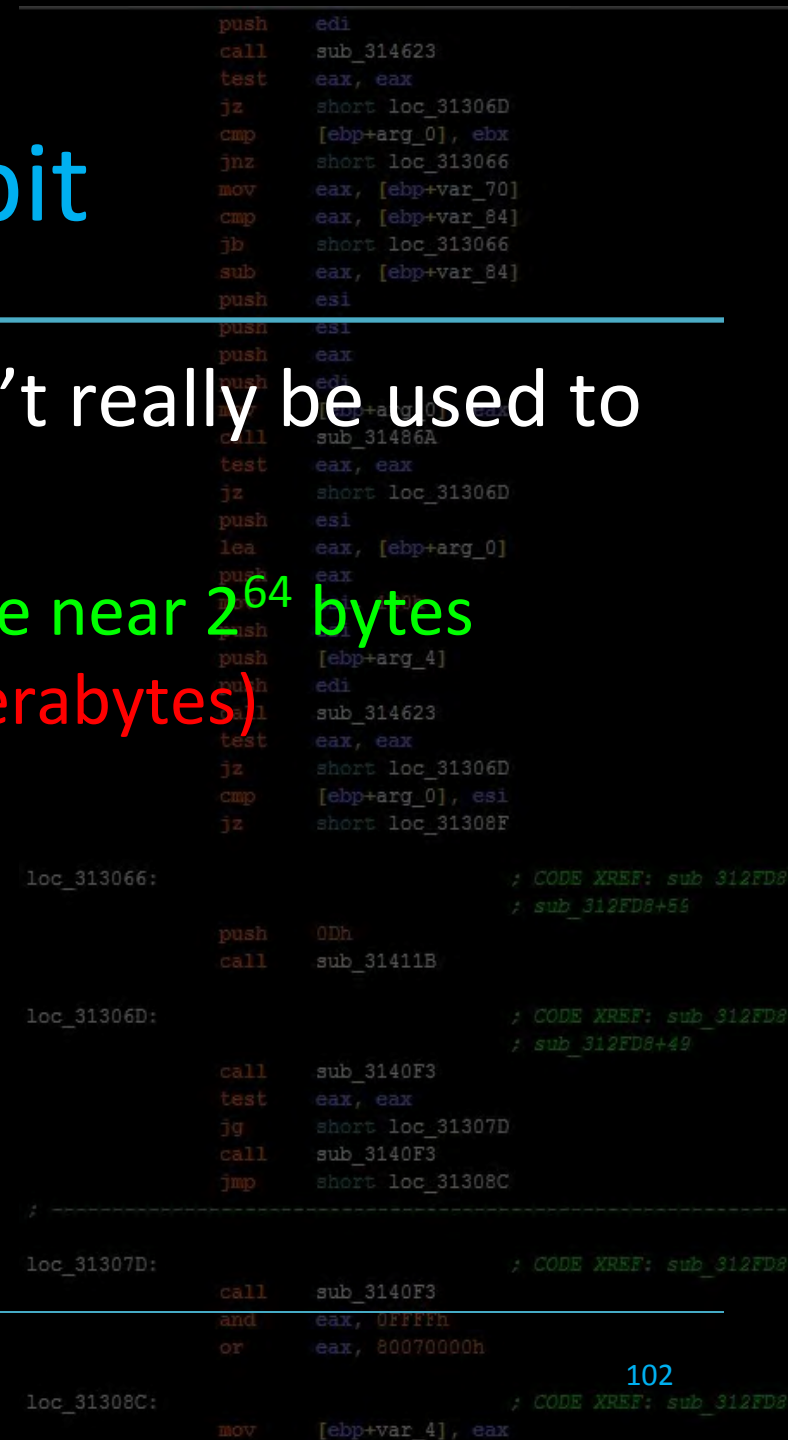
Heap Spraying on 64bit

- On 64bit heap spraying can't really be used to bypass ASLR
  - Good luck spraying anywhere near 2^{64} bytes



Heap Spraying on 64bit

- On 64bit heap spraying can't really be used to bypass ASLR
  - Good luck spraying anywhere near 2^{64} bytes (spoiler: that's ~18446744 terabytes)

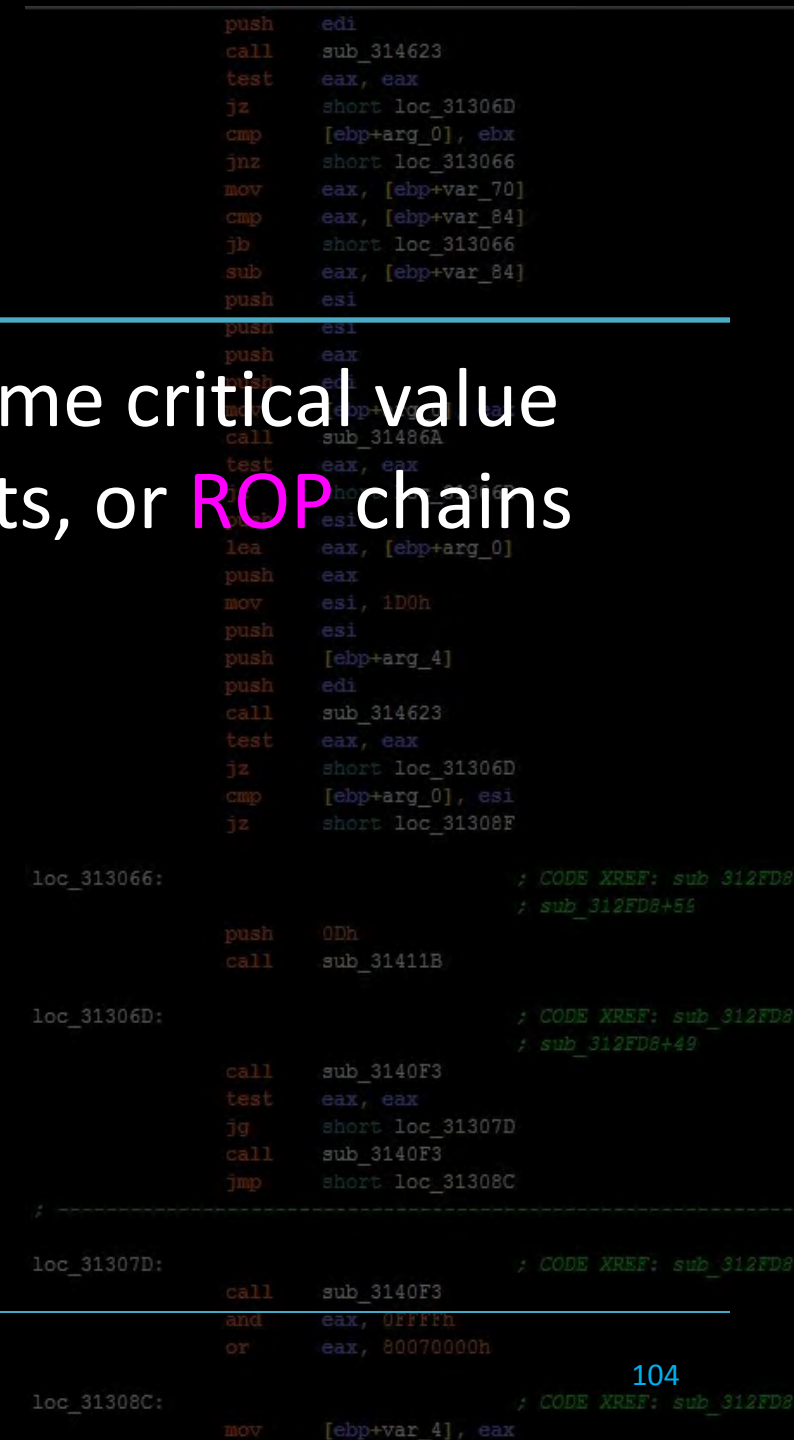


Heap Spraying on 64bit

- On 64bit heap spraying can't really be used to bypass ASLR
  - Good luck spraying anywhere near 2^{64} bytes (spoiler: that's ~18446744 terabytes)
- Targeted sprays are still useful in scenarios that you have a partial heap ptr overwrite or need to do some heap grooming

Heap Spray Payloads

- Pretty common to spray some critical value for your exploit, fake objects, or ROP chains

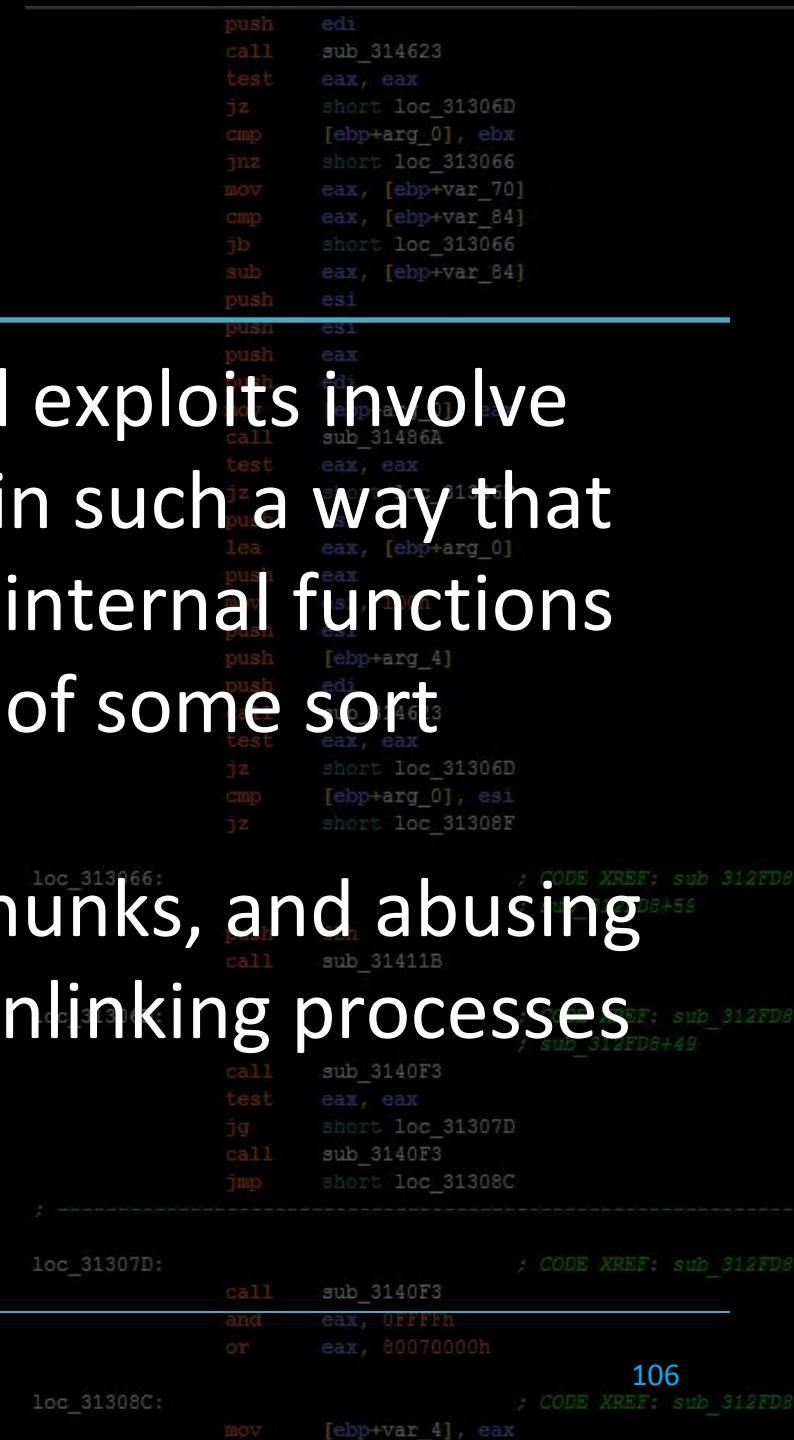


Lecture Overview

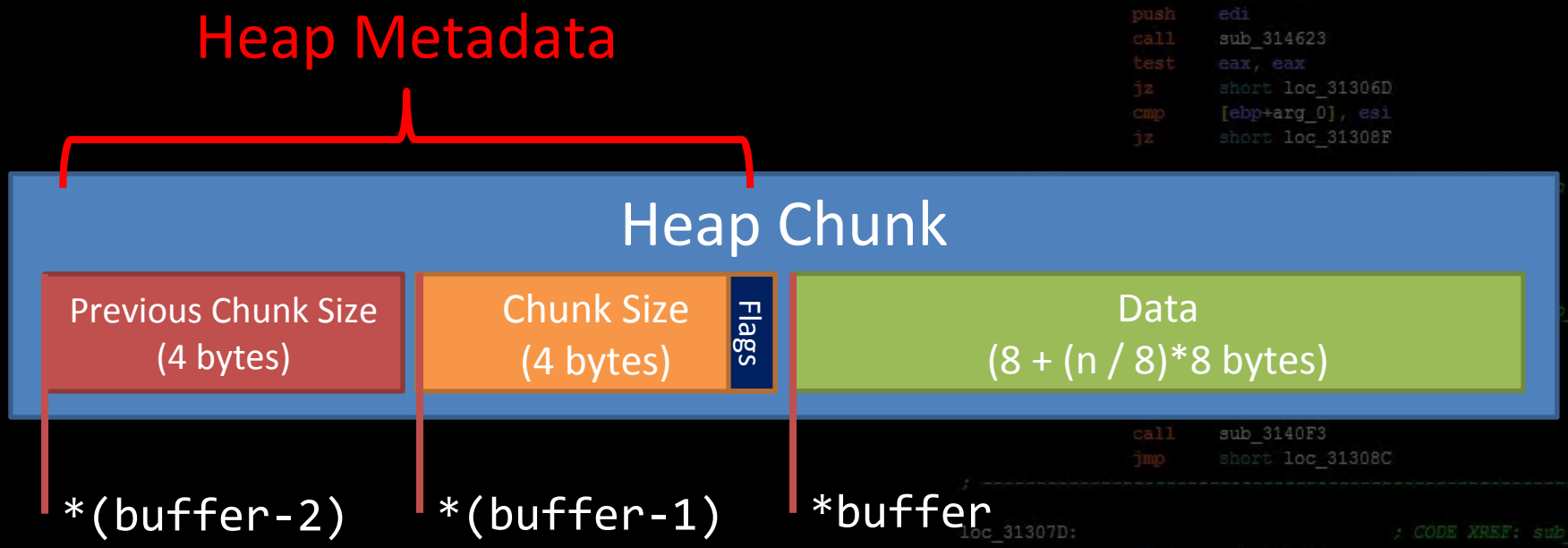
- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

Metadata Corruption

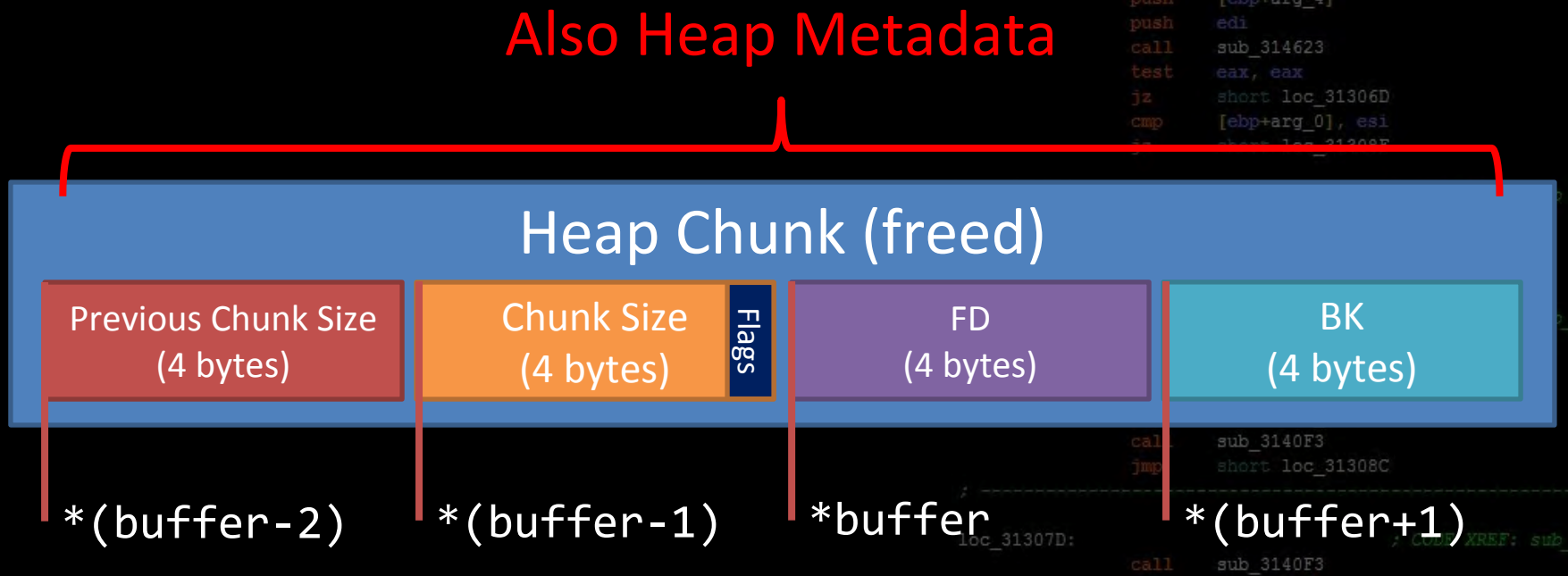
- Metadata corruption based exploits involve corrupting heap metadata in such a way that you can use the allocator's internal functions to cause a controlled write of some sort
- Generally involves faking chunks, and abusing its different coalescing or unlinking processes



Heap Chunks – In Use



Heap Chunks – Freed



Metadata Corruption

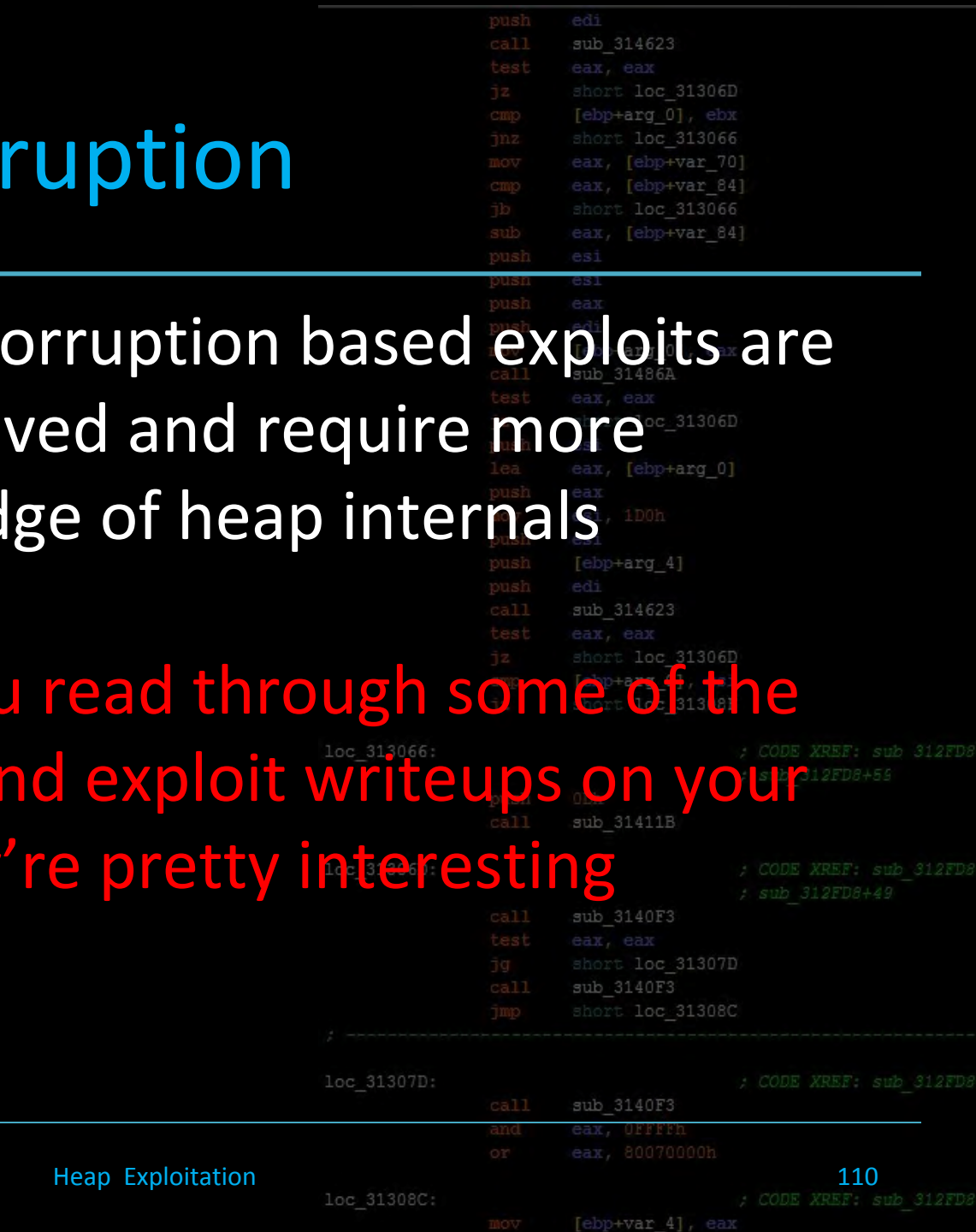
- The ‘hello world’ of heap metadata exploits is an example taught using the heap unlink() process when freeing a chunk
  - This is a dated and long since patched technique that is well documented

- https://sploitfun.wordpress.com/2015/02/26/heap-overflow-using-unlink/

Metadata Corruption

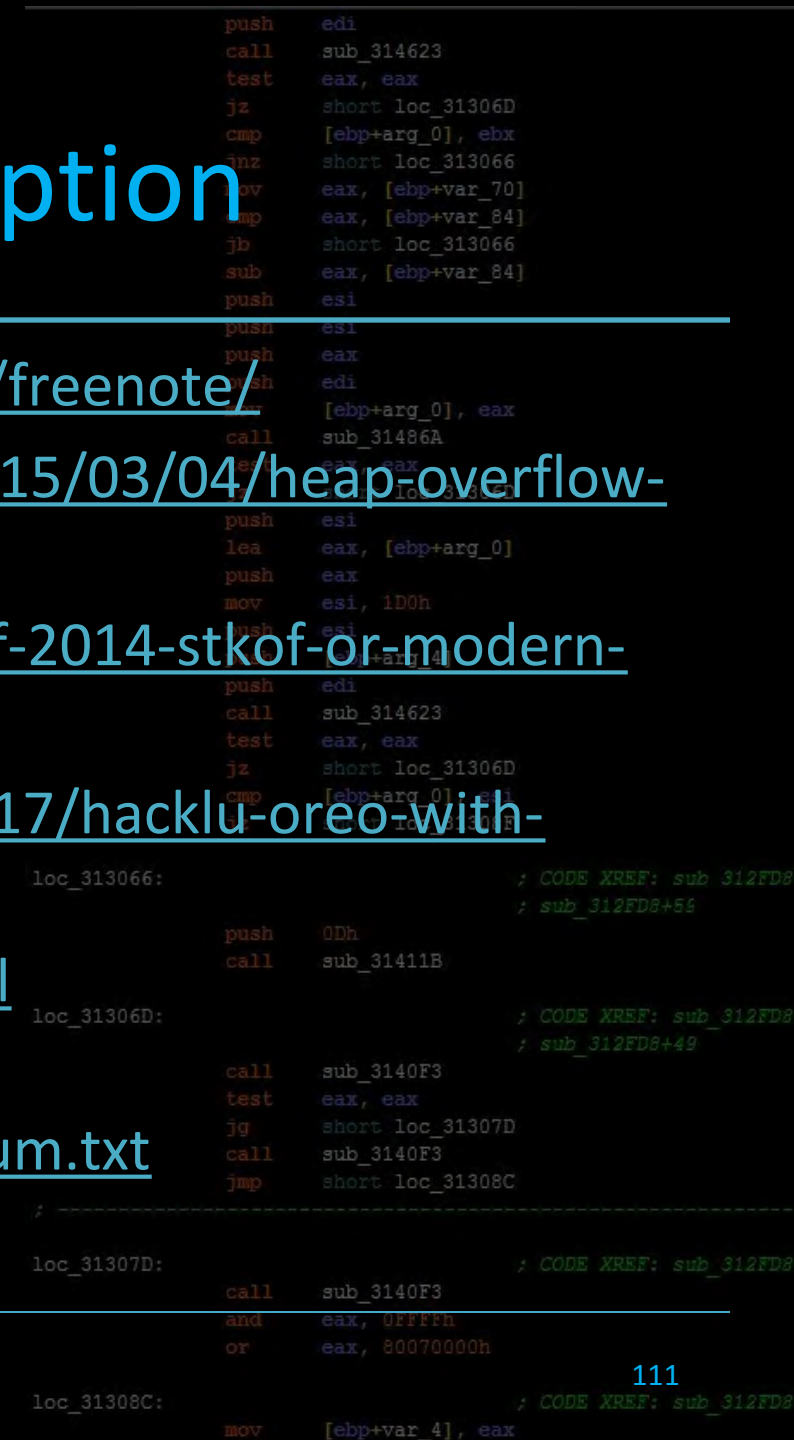
- Heap metadata corruption based exploits are usually very involved and require more intimate knowledge of heap internals

- It's suggested you read through some of the following blogs and exploit writeups on your own time as they're pretty interesting



glibc Metadata Corruption

- https://kitctf.de/writeups/0ctf2015/freenote/
- https://sploitfun.wordpress.com/2015/03/04/heap-overflow-using-malloc-maleficarum/
- http://acez.re/ctf-writeup-hitcon-ctf-2014-stkof-or-modern-heap-overflow/
- http://wapiflapi.github.io/2014/11/17/hacklu-oreo-with-ret2dl-resolve/
- http://phrack.org/issues/66/10.html
- http://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt



Metadata Corruption

- Metadata exploits are hard to pull of nowadays as heaps are fairly hardened (especially on modern Windows OS's)
- We won't really be testing on metadata corruption, but it's still something you try to familiarize yourself with