Project 1: 37 days left

# Offense-Based Cyber Security: Exploitation of Memory Vulnerabilities

CS 459/559: Science of Cyber Security
5th Lecture

**Instructor:**

Guanhua Yan

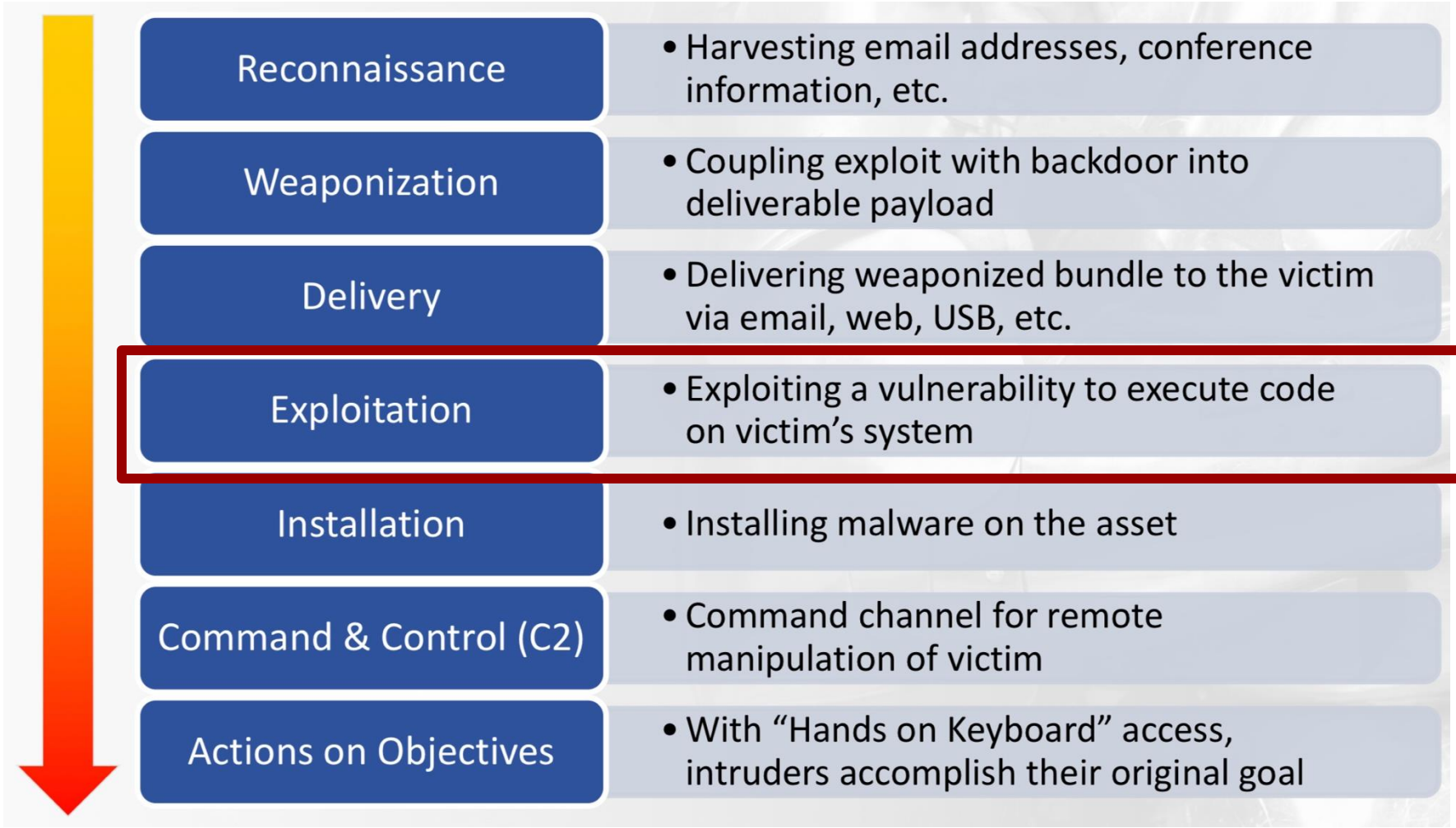
# Project 1

- Demonstrate a cyber attack
- In your first project report, please:
  - Explain what are the vulnerability, exploit, attack surface, and attack vector in your project
  - Explain how the attack (exploitation) occurs in your project
  - Explain why the attack (exploitation) works
- Each of your project reports should be **at least five pages, excluding bibliography**
- **Due time: October 10**
  - **Four days of grace period, with 2.5% penalty each half day late**
- **Grading criteria:**
  - **Results, novelty, difficulty, presentation**

# What we have learned from last lecture

- **Network vulnerability exploitation**
  - Cache poisoning attacks
  - Sniffing
  - Denial of service attacks

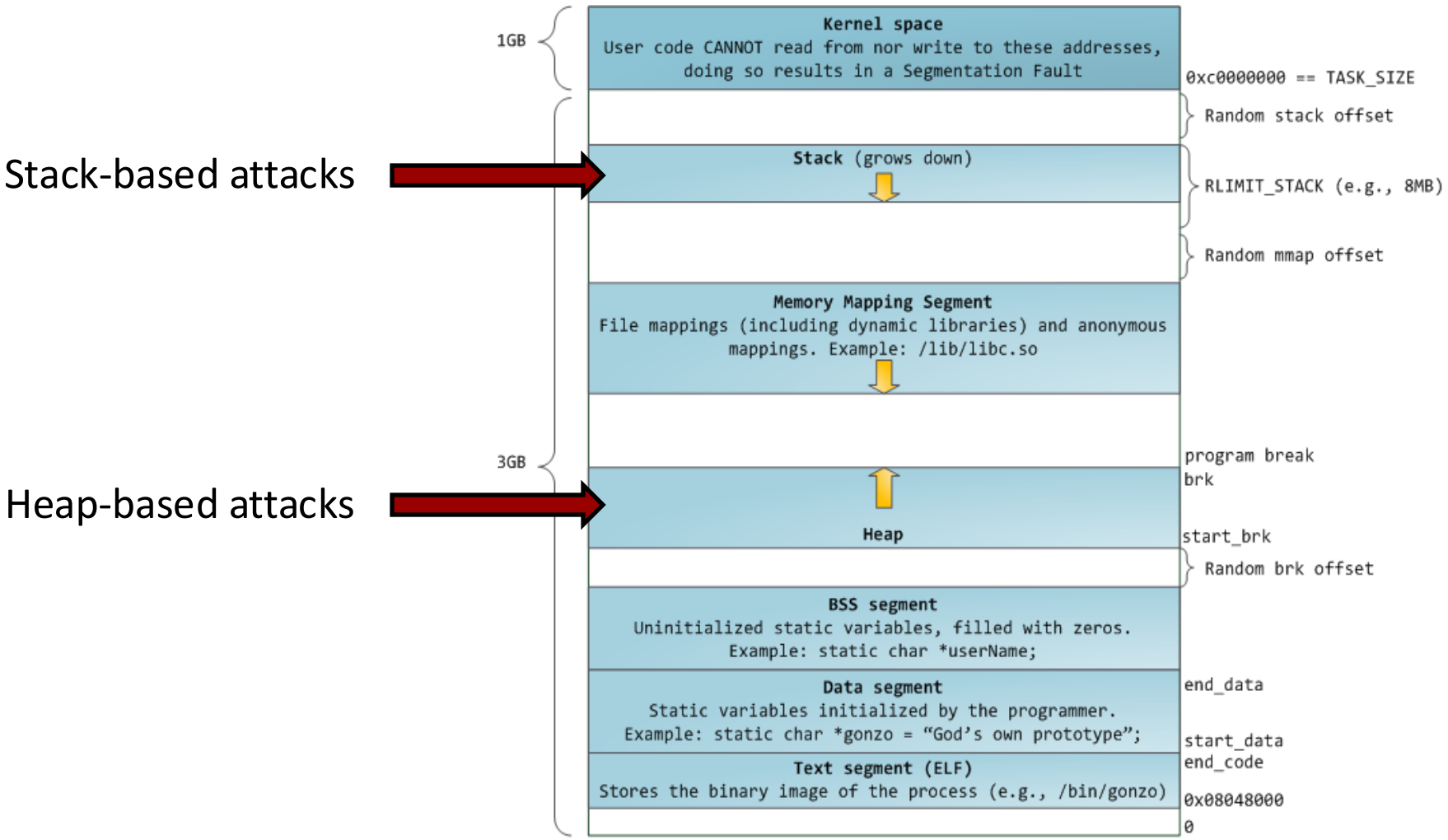
# Exploitation of memory vulnerabilities



# What we are going to learn

- *Stack-based attacks*
- *Heap-based attacks*

# Run-time Linux process memory layout



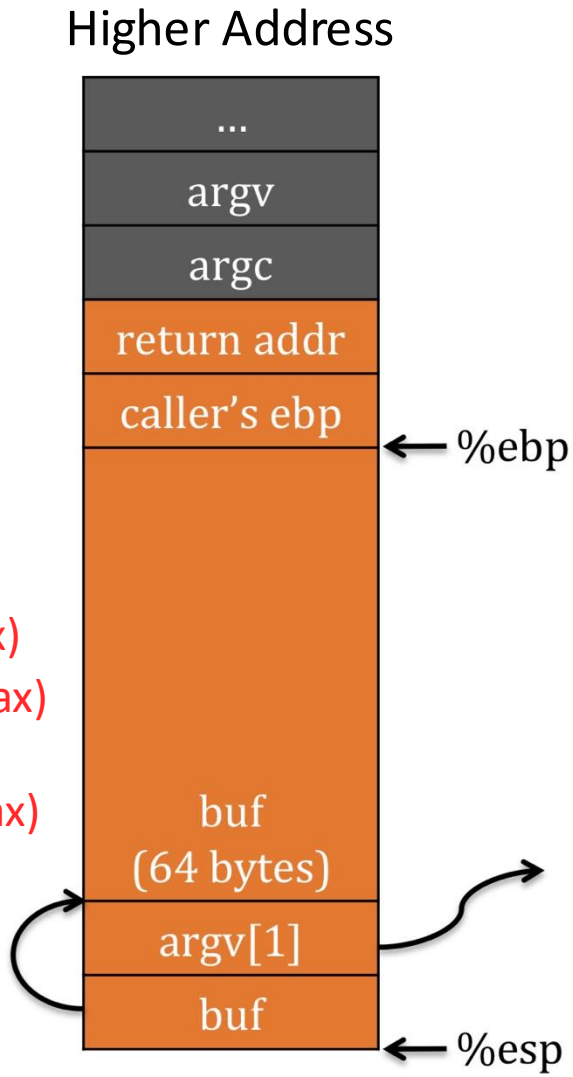
# *Stack-based attacks*

# Example vulnerable C code

leave:
mov %ebp, %esp
pop %ebp

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

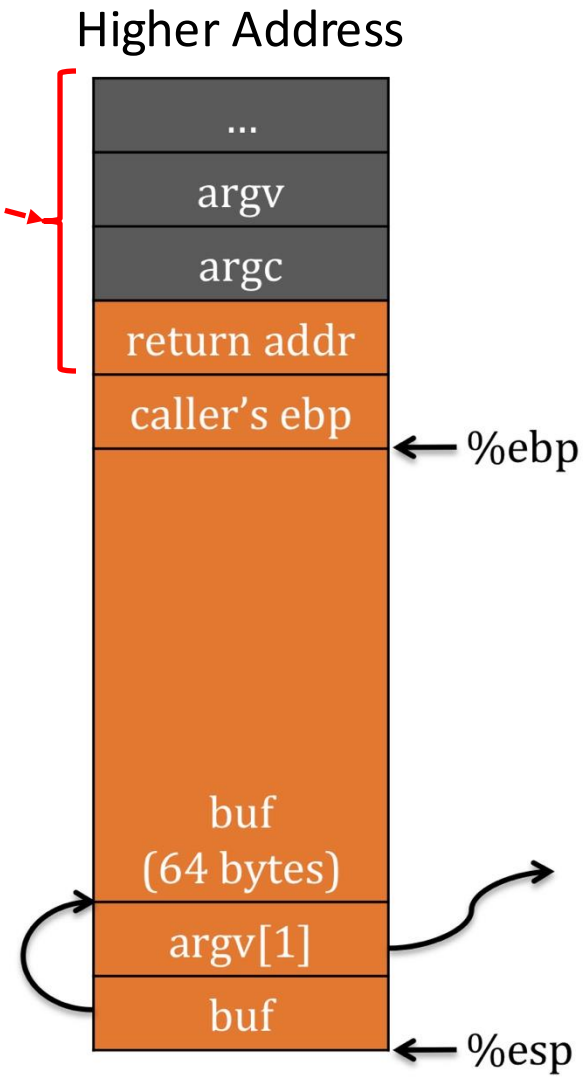
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax   (argv→eax)
   0x080483ed <+9>:  mov    4(%eax),%eax    (argv[1]→eax)
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax  (buf→eax)
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave  (mov ebp, esp; pop ebp)
   0x08048400 <+28>: ret
```



# Example vulnerable C code

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

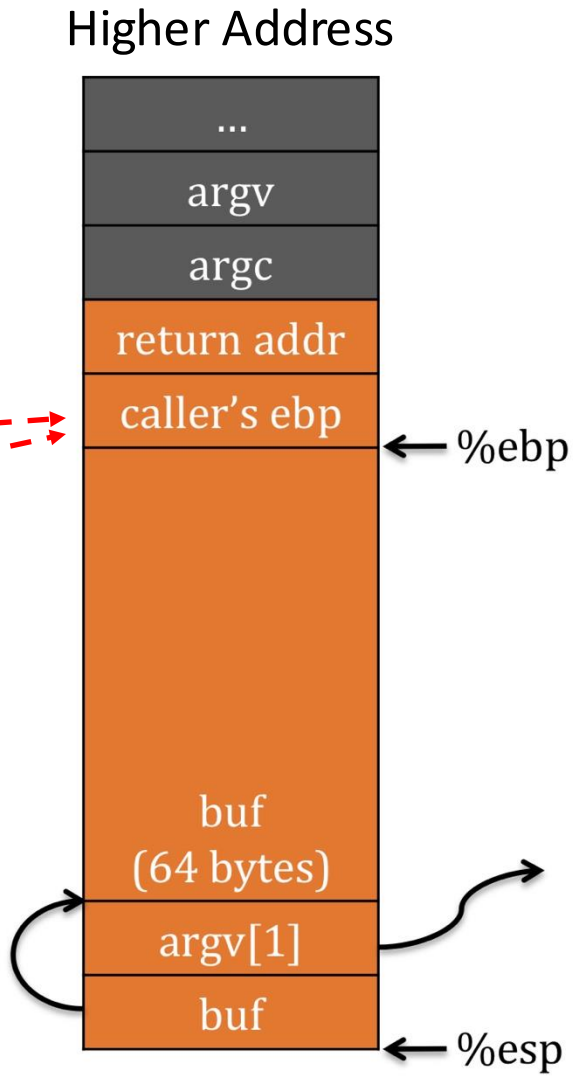
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



# Example vulnerable C code

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

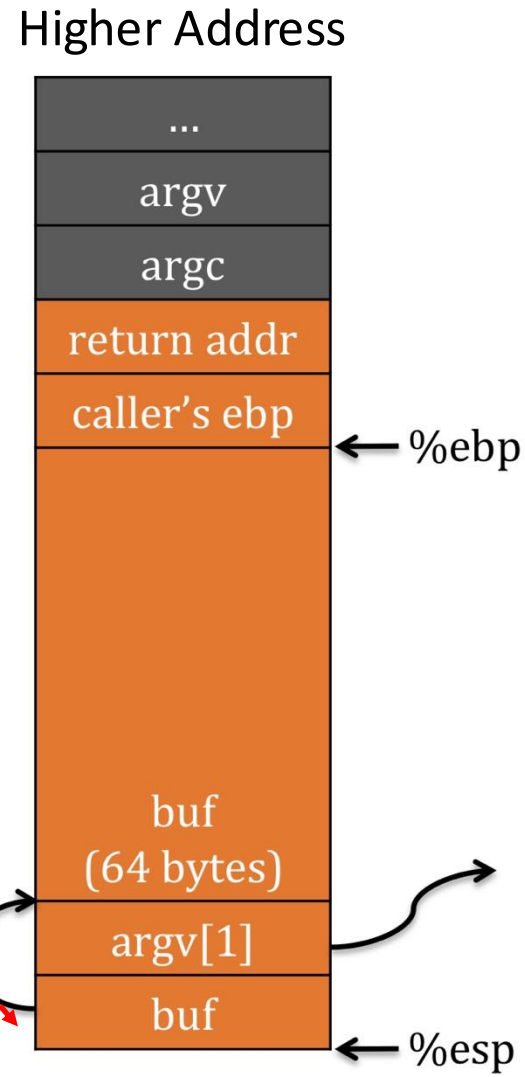
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



# Example vulnerable C code

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

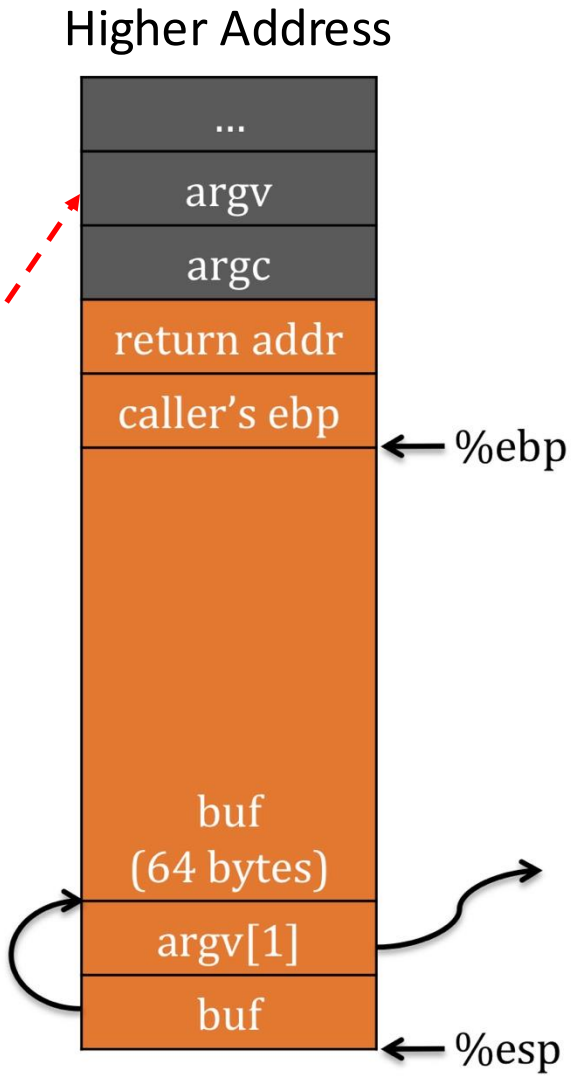
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



# Example vulnerable C code

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

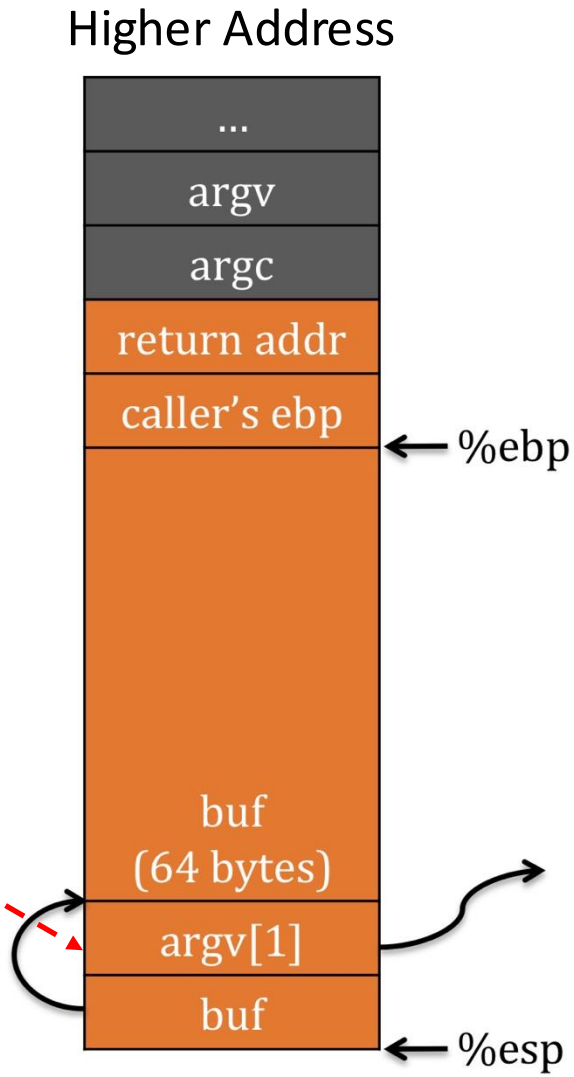
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



# Example vulnerable C code

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

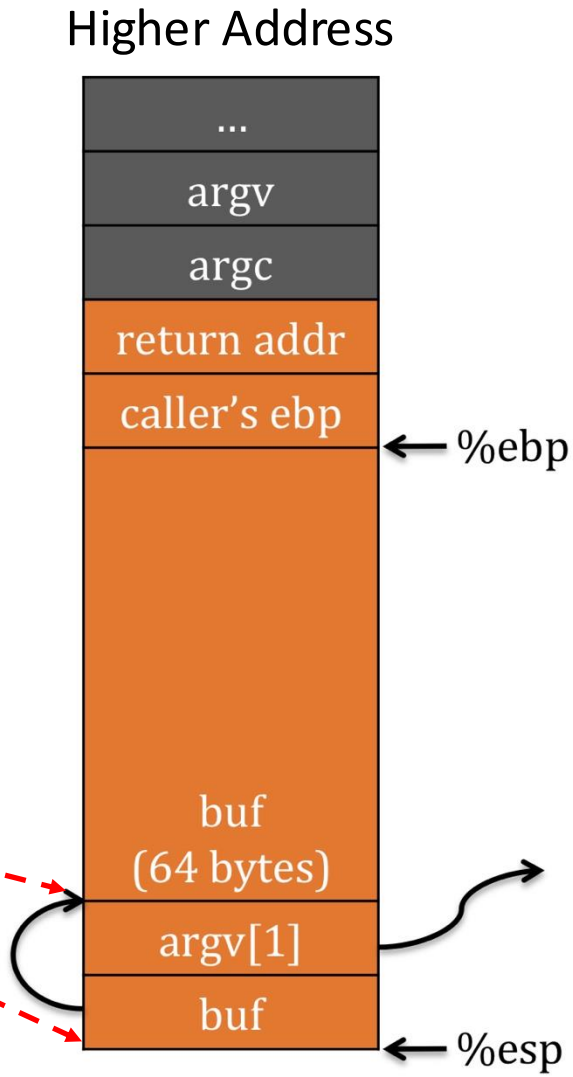
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



# Example vulnerable C code

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

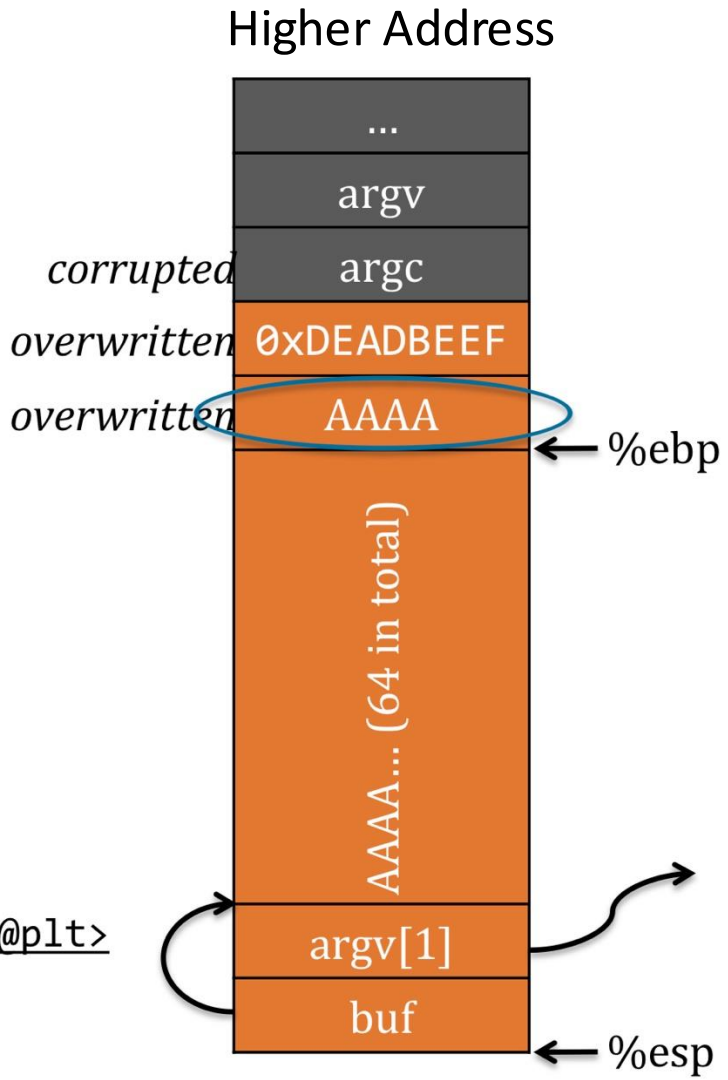
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



# Buffer overflow attack

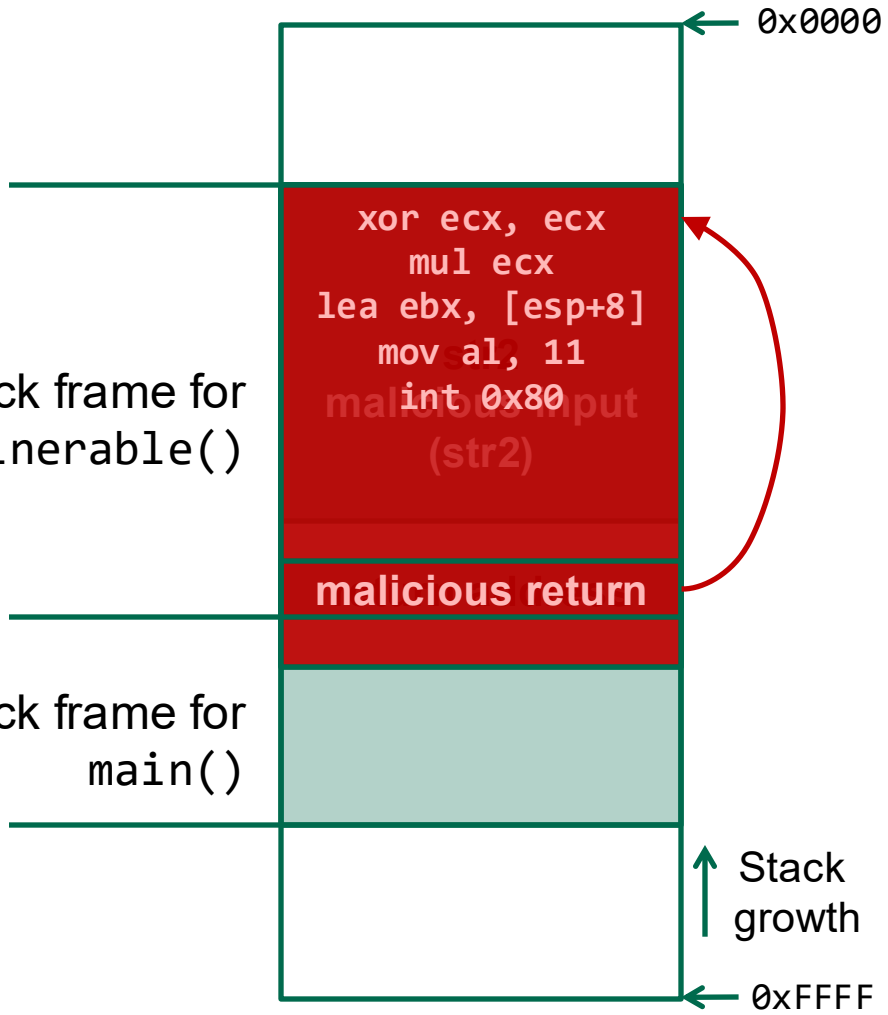
```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



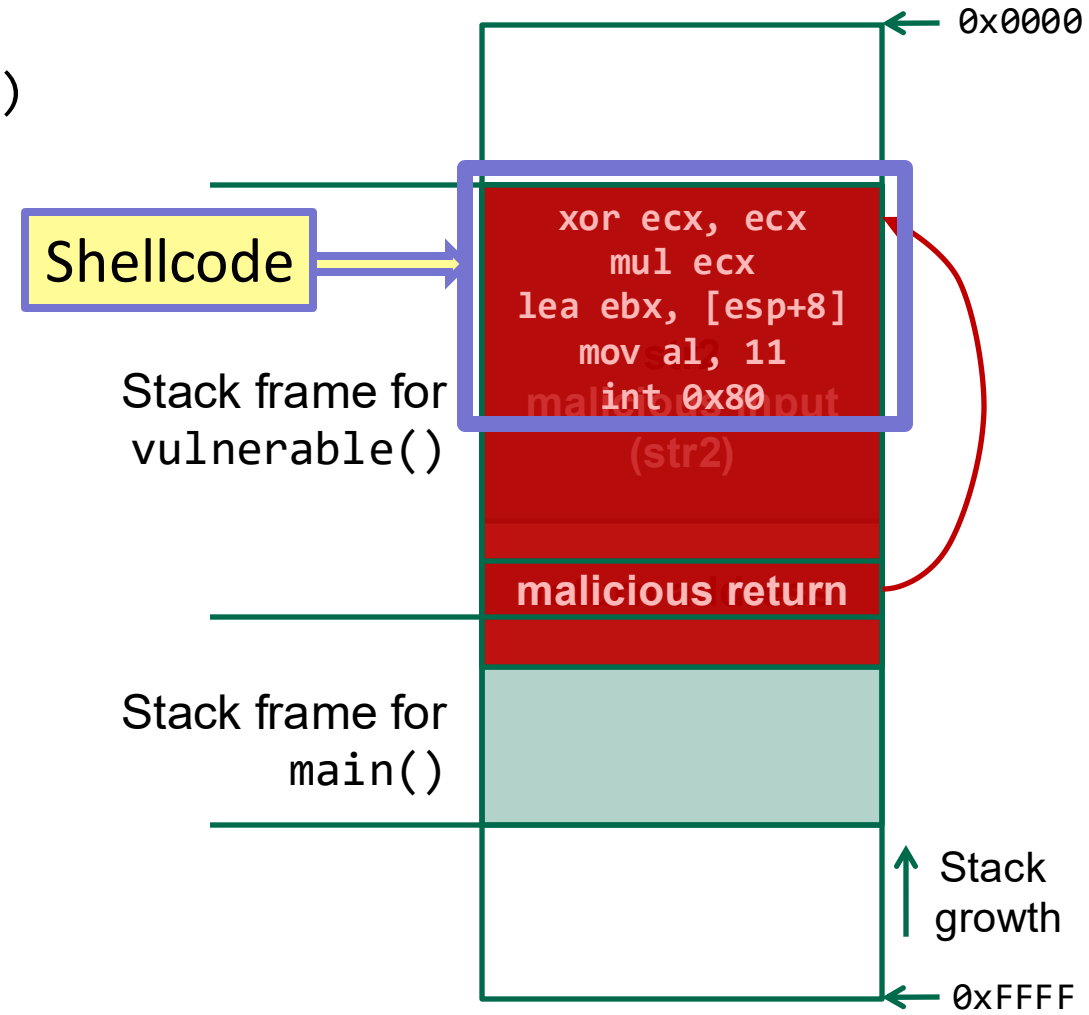
# Buffer Overflow and Code Injection Attack: Example

```
main (int argc, char **argv)
  {
    ...
    vulnerable(argv[1]);
    ...
  }

vulnerable(char *str1)
  {
    char str2[100];
    strcpy(str2,str1);
    return;
  }
```



# Shellcode: a piece of code used as payload when exploiting a software vulnerability

```
→main (int argc, char **argv)
  {
    ...
→   vulnerable(argv[1]);
    ...
  }

→vulnerable(char *str1)
  {
→   char str2[100];
→   strcpy(str2,str1);
→   return;
  }
```



# How to generate shellcode?

- Spawn a shell: a program that takes commands from the keyboard and gives them to the OS to perform.

```
#include <unistd.h>
int main() {
        char *args[2];
        args[0] = "/bin/sh";
        args[1] = NULL;
        execve(args[0], args, NULL);
}
```

A new shell!

ssh | bash

```
ghyan@bravo:~/teaching$ ls
a.out  get_shell  get_shell2.c  get_shell.asm  get_shell.c  get_shell.o  x2.c  x.c  y.c
ghyan@bravo:~/teaching$ gcc ./get_shell.c -o ./get_shell
ghyan@bravo:~/teaching$ ./get_shell
$
```

# Assembly to binary shellcode

**get_shell.asm**

```
jmp short    mycall              ; Immediately jump to the call instruction

shellcode:
    pop        esi               ; Store the address of "/bin/sh" in ESI
    xor        eax, eax          ; Zero out EAX
    mov byte   [esi + 7], al     ; Write the null byte at the end of the string

    mov dword  [esi + 8],  esi   ; [ESI+8], i.e. the memory immediately below the string
                                 ;   "/bin/sh", will contain the array pointed to by the
                                 ;   second argument of execve(2); therefore we store in
                                 ;   [ESI+8] the address of the string...
    mov dword  [esi + 12], eax   ; ...and in [ESI+12] the NULL pointer (EAX is 0)
    mov        al,  0xb          ; Store the number of the syscall (11) in EAX
    lea        ebx, [esi]        ; Copy the address of the string in EBX
    lea        ecx, [esi + 8]    ; Second argument to execve(2)
    lea        edx, [esi + 12]   ; Third argument to execve(2) (NULL pointer)
    int        0x80              ; Execute the system call

mycall:
    call       shellcode         ; Push the address of "/bin/sh" onto the stack
    db         "/bin/sh"
```

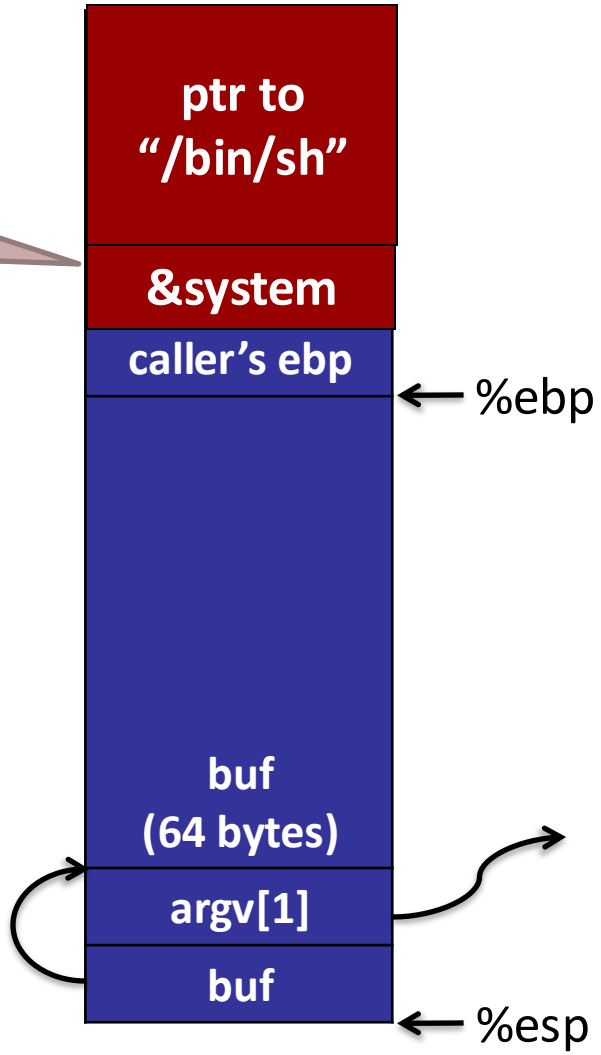
```
char shellcode[] = "\xeb\x18\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89\x46"
                   "\x0c\xb0\x0b\x8d\x1e\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
                   "\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

# Code Reuse Attacks

- Key Idea: Reuse existing library code instead of code injection
- Bypass NX
- Return-to-libc attacks
- Return Oriented Programming
- **Jump Oriented Programming**

# Return-to-libc attack

ret transfers control to `system`, which finds arguments on stack




**Overwrite return address with address of libc function**

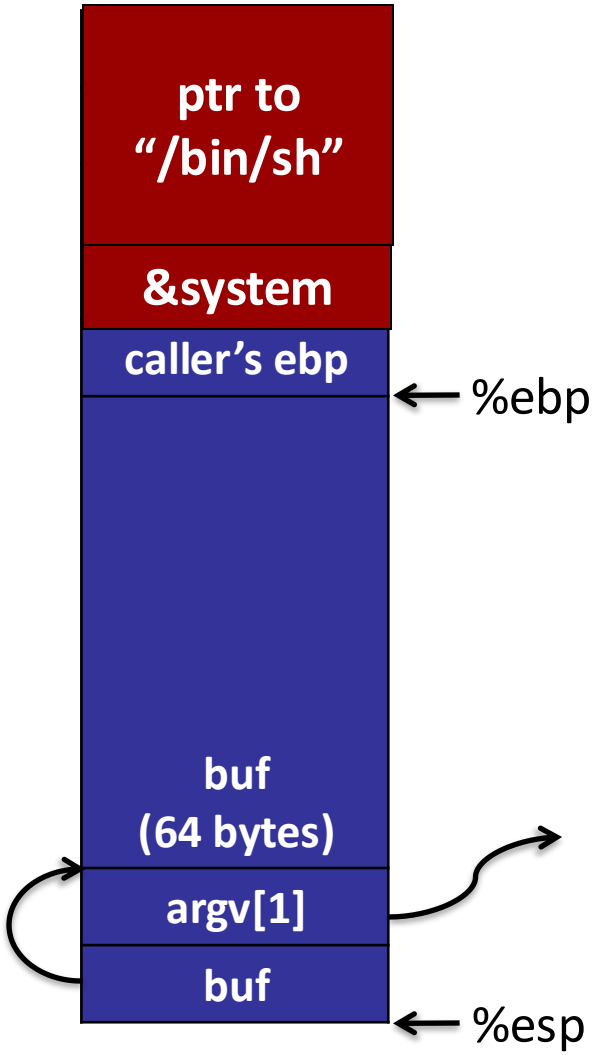
- **setup fake return address and argument(s)**
- **`ret` will "call" libc function**

**No injected code!**

# Challenge

What if we don't know the absolute address any pointers to "/bin/sh"

(objdump gives addresses, but we don't know ASLR constants)

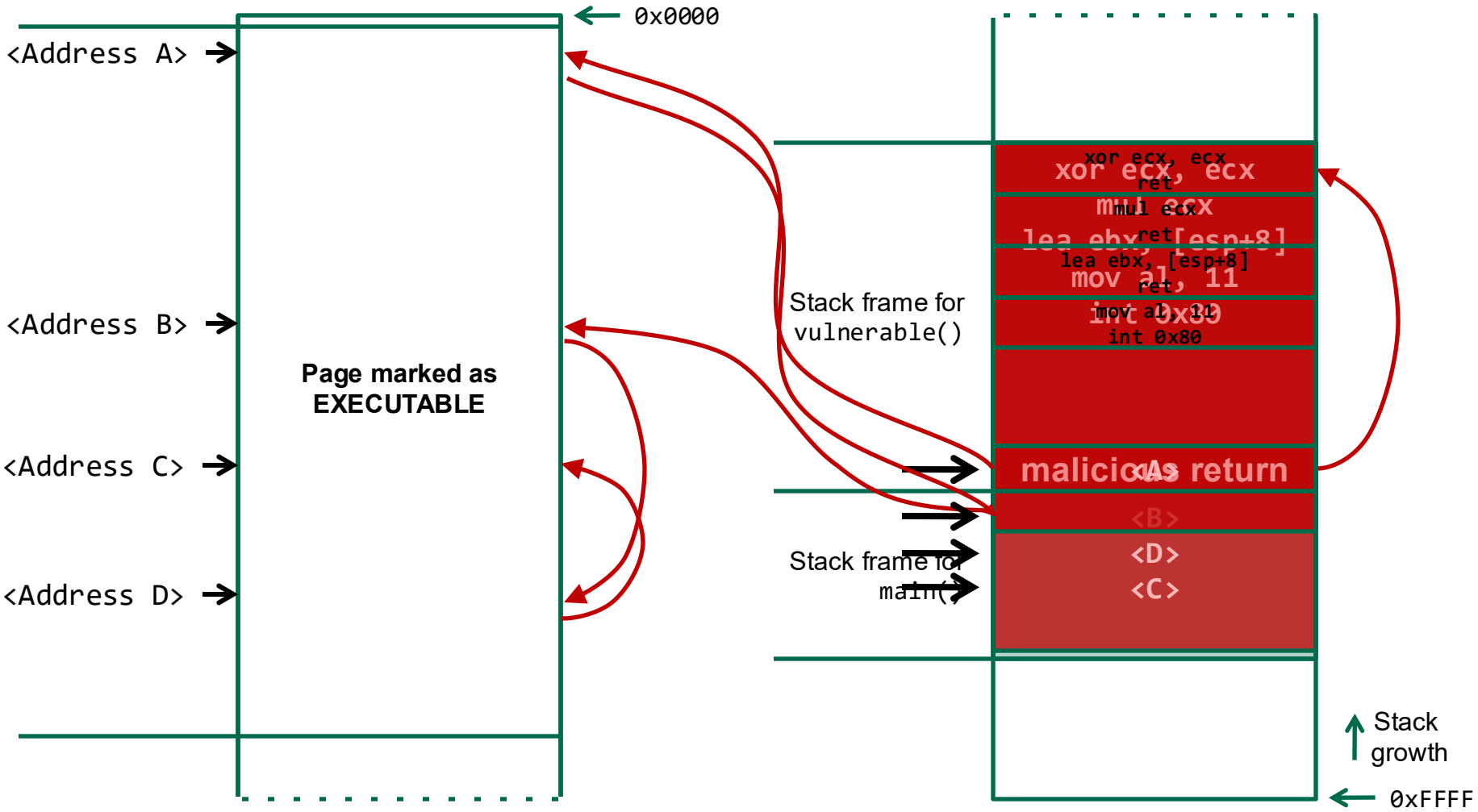


# Return Oriented Programming Attacks

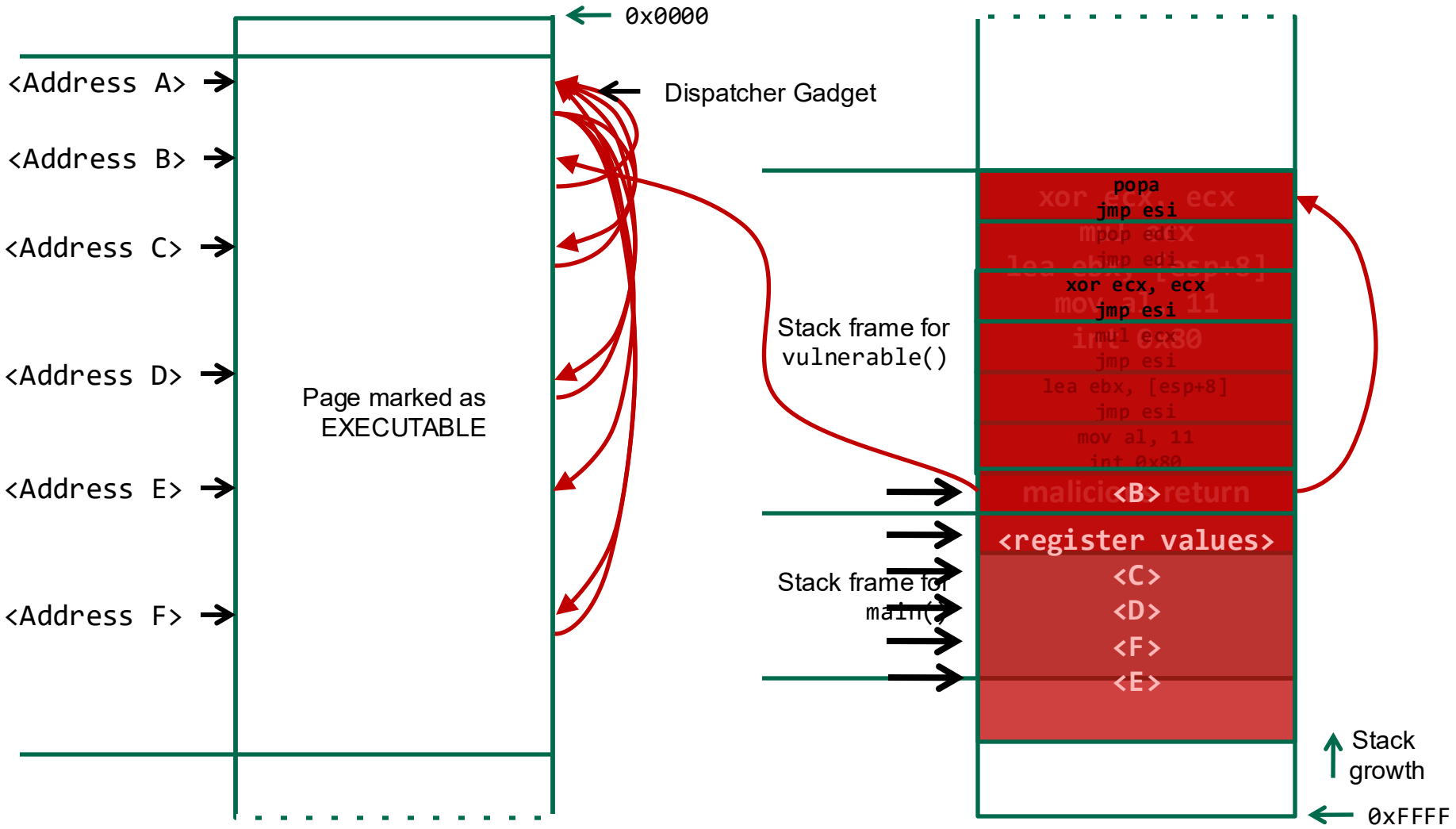
- **Turing-complete**
  - X86
  - SPARC
  - ARM
- **Exploits**
  - Voting machine
  - Atmel sensor
  - Cisco router
  - Xen hypervisor
  - Jailbreak
  - Pwn2Own
- **Automated tools**

- **Microsoft BlueHat Prize ($260K)**

# Return-Oriented Programming (ROP)



# Jump Oriented Programming



# pusha/popa

**pusha**

This instruction pushes all the general purpose registers onto the stack in the following order: AX, CX, DX, BX, SP, BP, SI, DI. The value of SP pushed is the value before the instruction is executed. It is useful for saving state before an operation that could potential change these registers.

**popa**

This instruction pops all the general purpose registers off the stack in the reverse order of PUSHA. That is, DI, SI, BP, SP, BX, DX, CX, AX. Used to restore state after a call to PUSHA.

# *Heap-based attacks*

# The heap

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

# 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
- Some applications even create their own heap implementations!

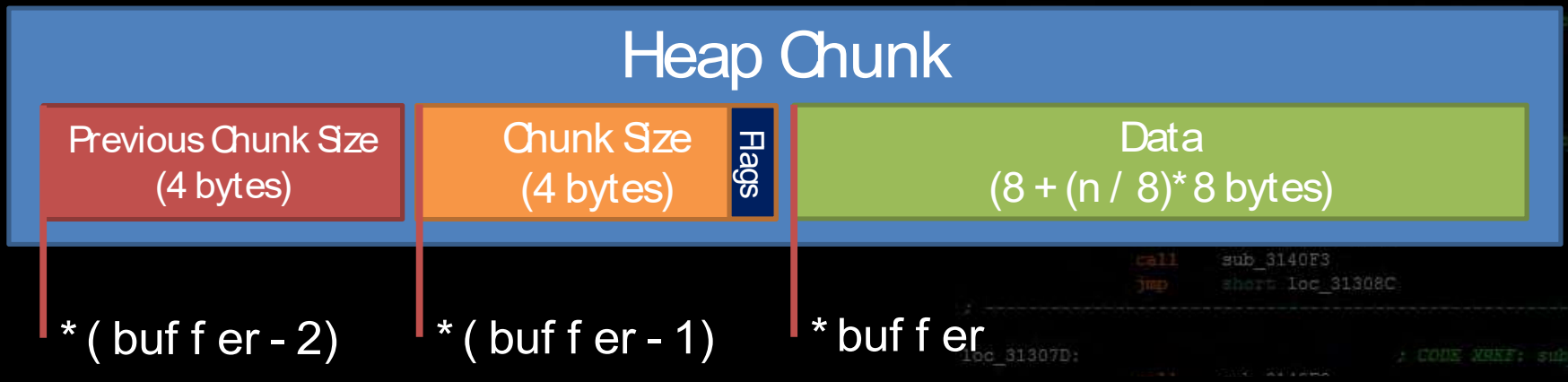
# Heap implementations

- 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

# Heap Chunks

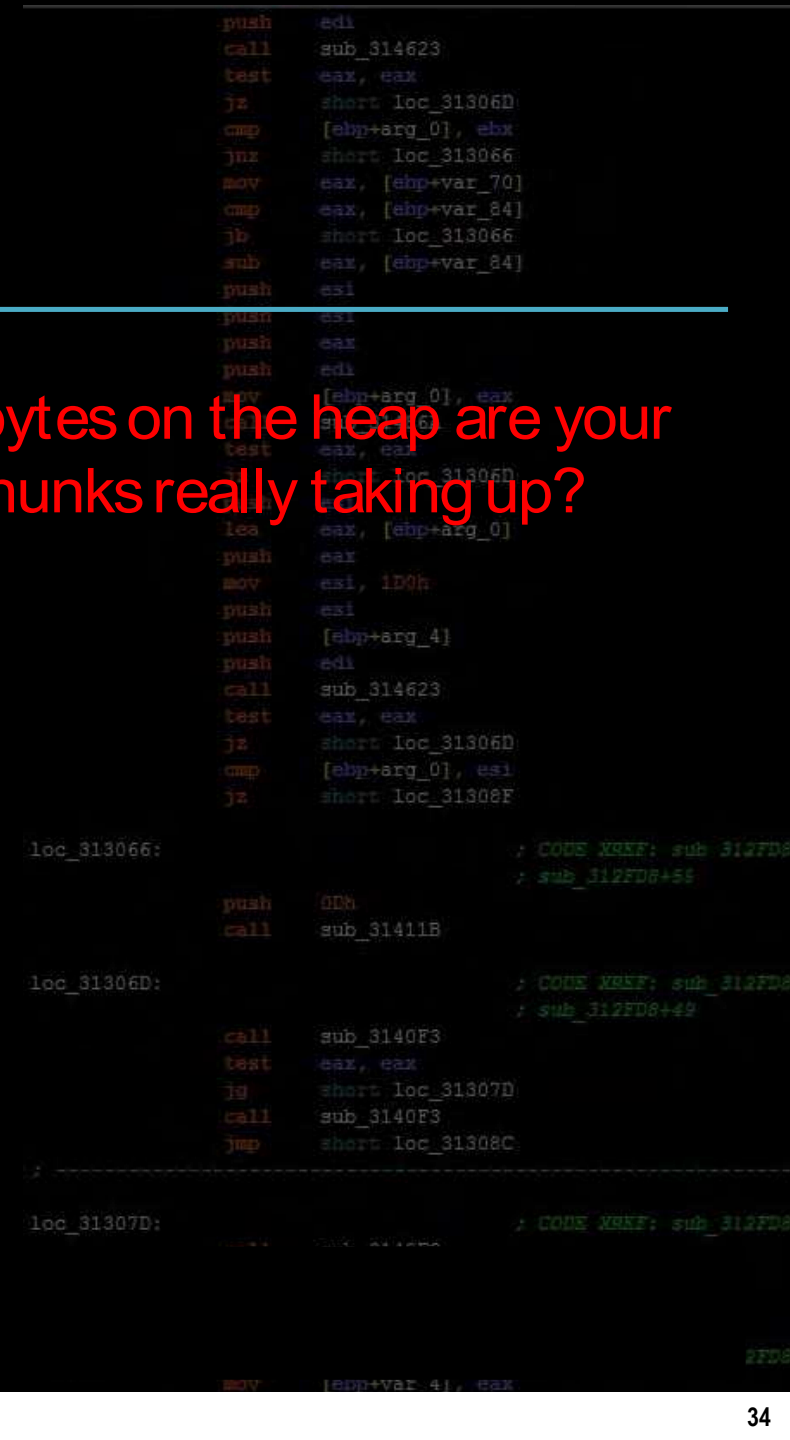
```
unsigned int *buffer = NULL;
buffer = malloc(0x100);

//Out comes a heap chunk
```



# Malloc Trivia

- malloc(28);
  - 40 bytes
- malloc(4);
  - 16 bytes
- malloc(10);
  - 24 bytes
- malloc(0);
  - 16 bytes

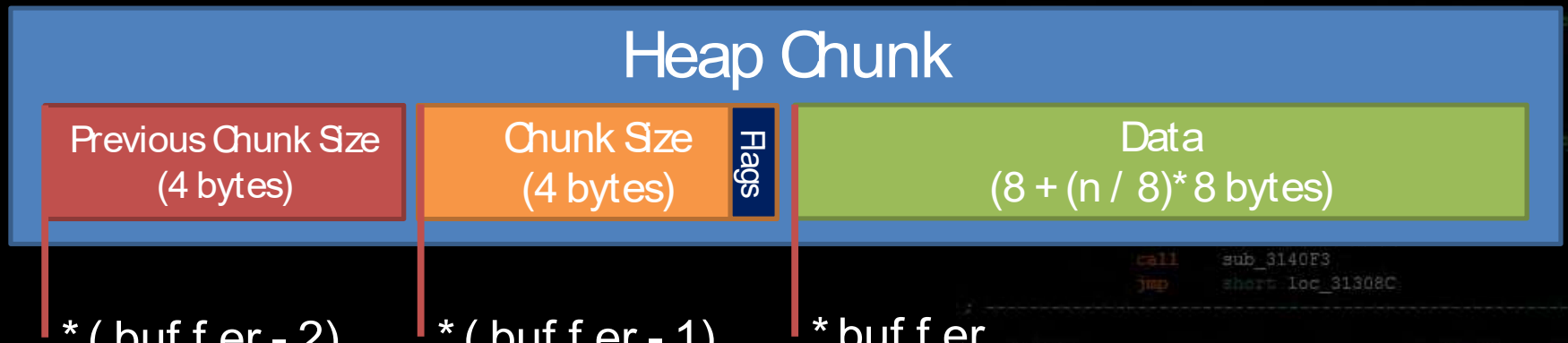
How many bytes on the heap are your malloc chunks really taking up?



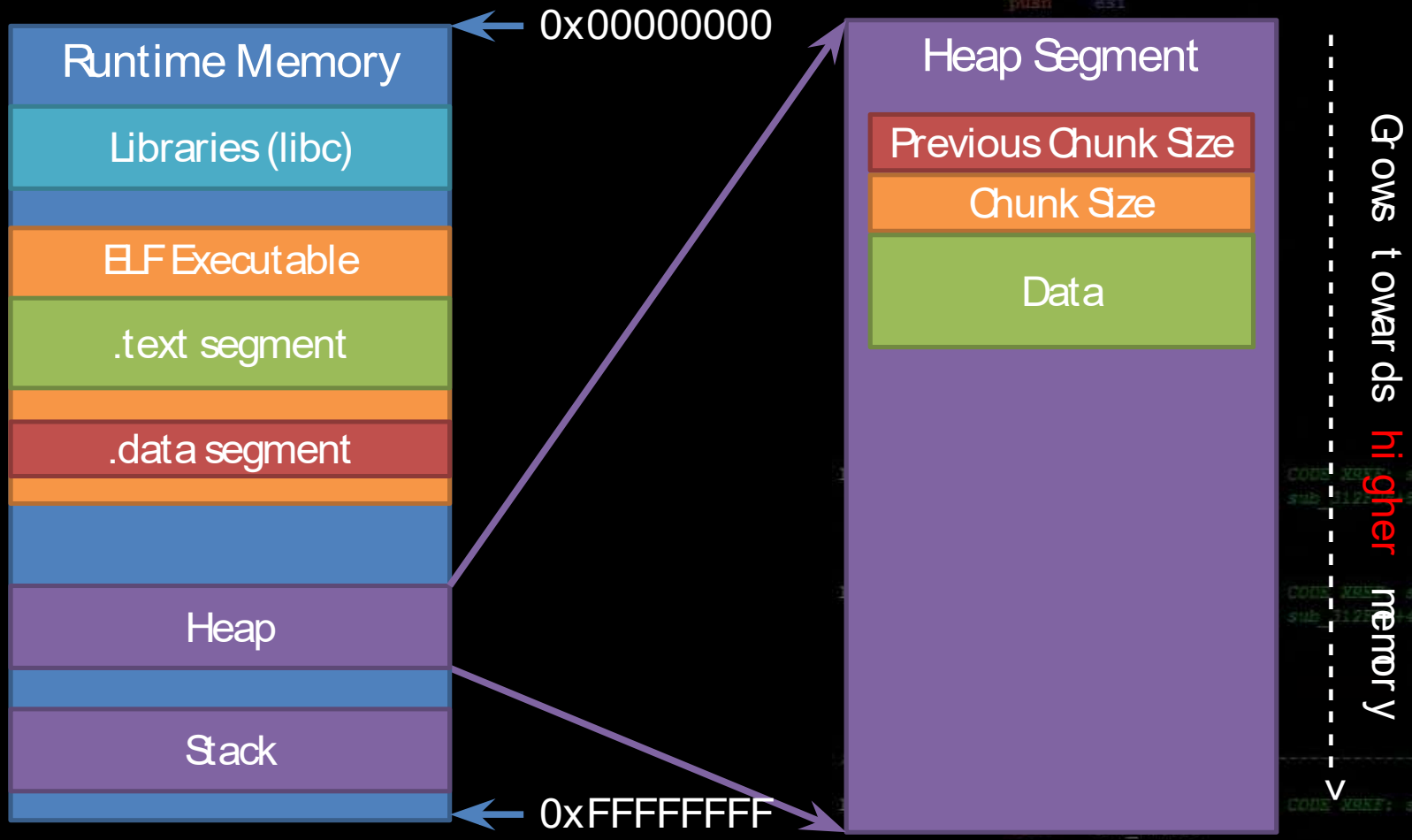
# Heap Chunks

- 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.

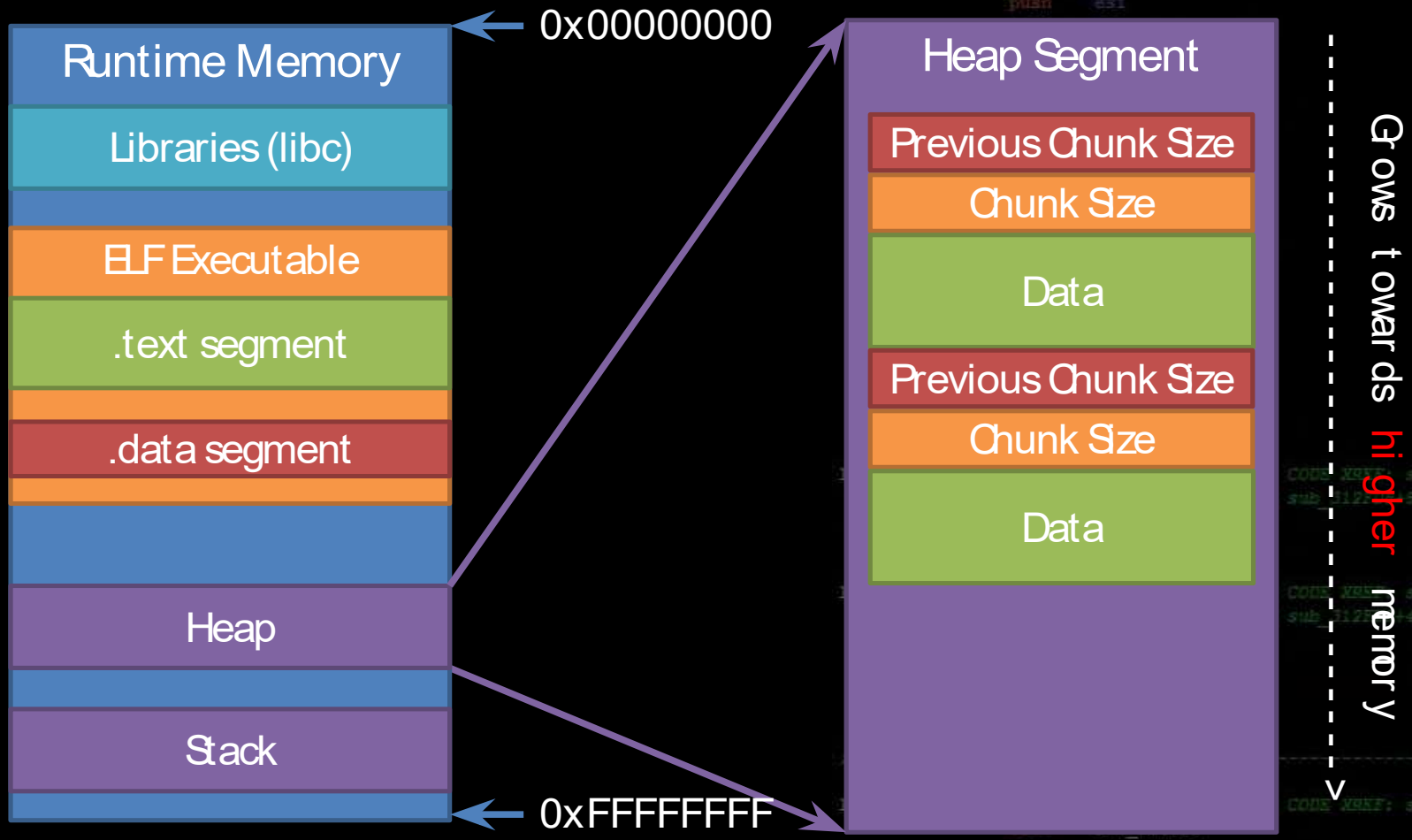
0x01 PREV_INUSE – set when previous chunk is in use
0x02 IS_MMAPPED – set if chunk was obtained with mmap()
0x04 NON_MAIN_ARENA – set if chunk belongs to a thread arena



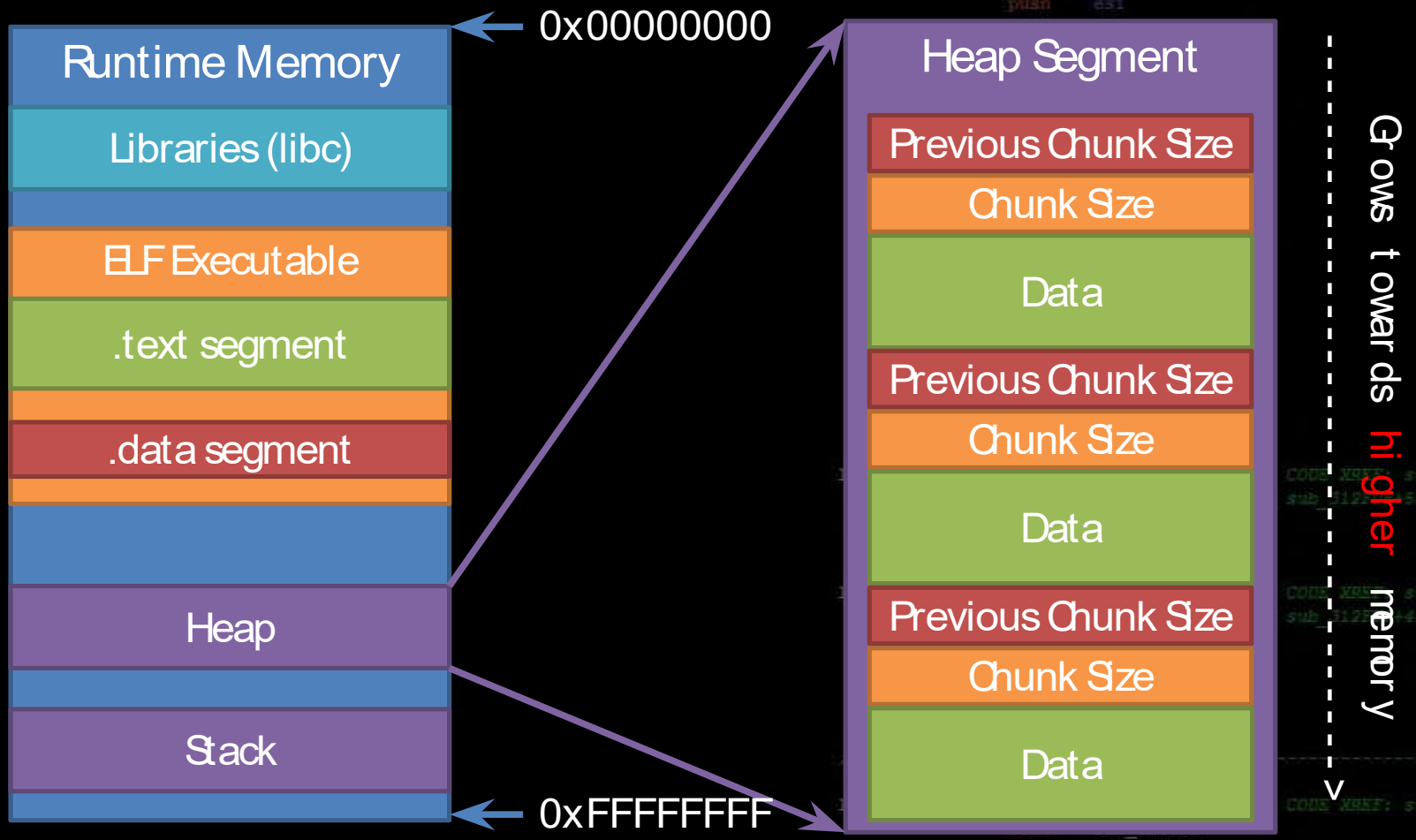
# Heap Allocations



# Heap Allocations

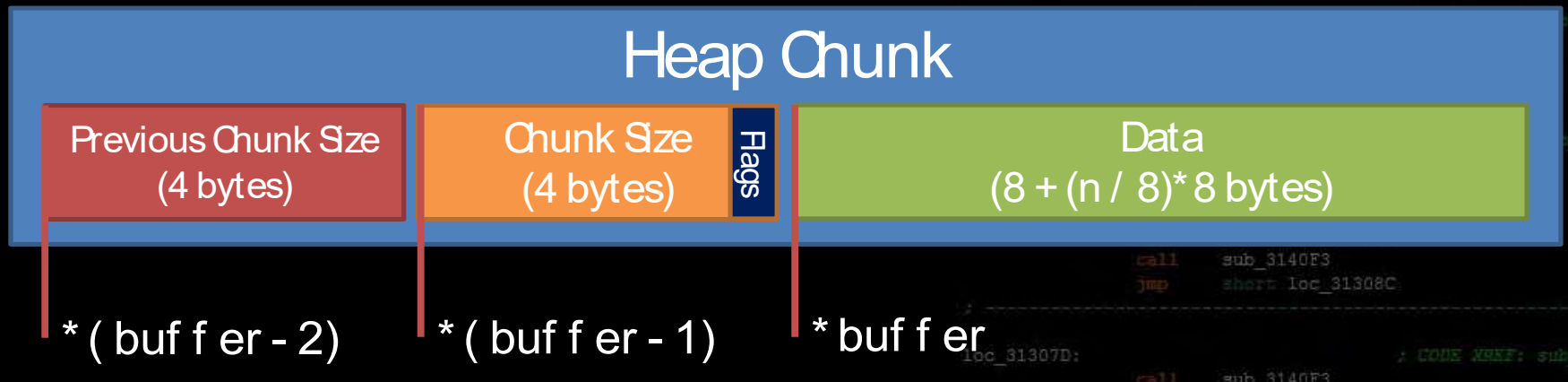


# Heap Allocations



# Heap Chunks – In Use

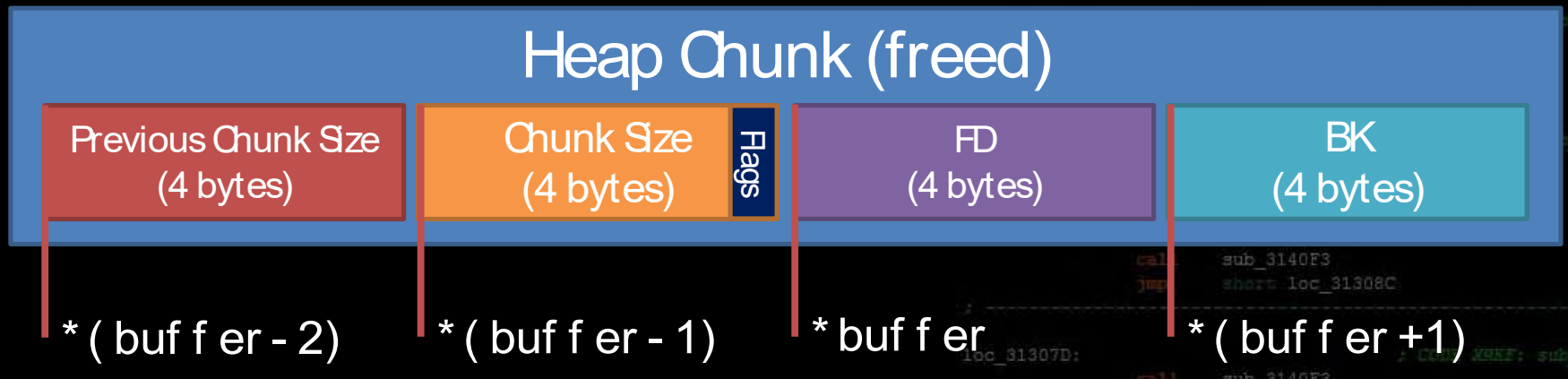
- Heap chunks exist in two states
  - in use (malloc'd)
  - free'd



# Heap Chunks – Freed

free( buffer );

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

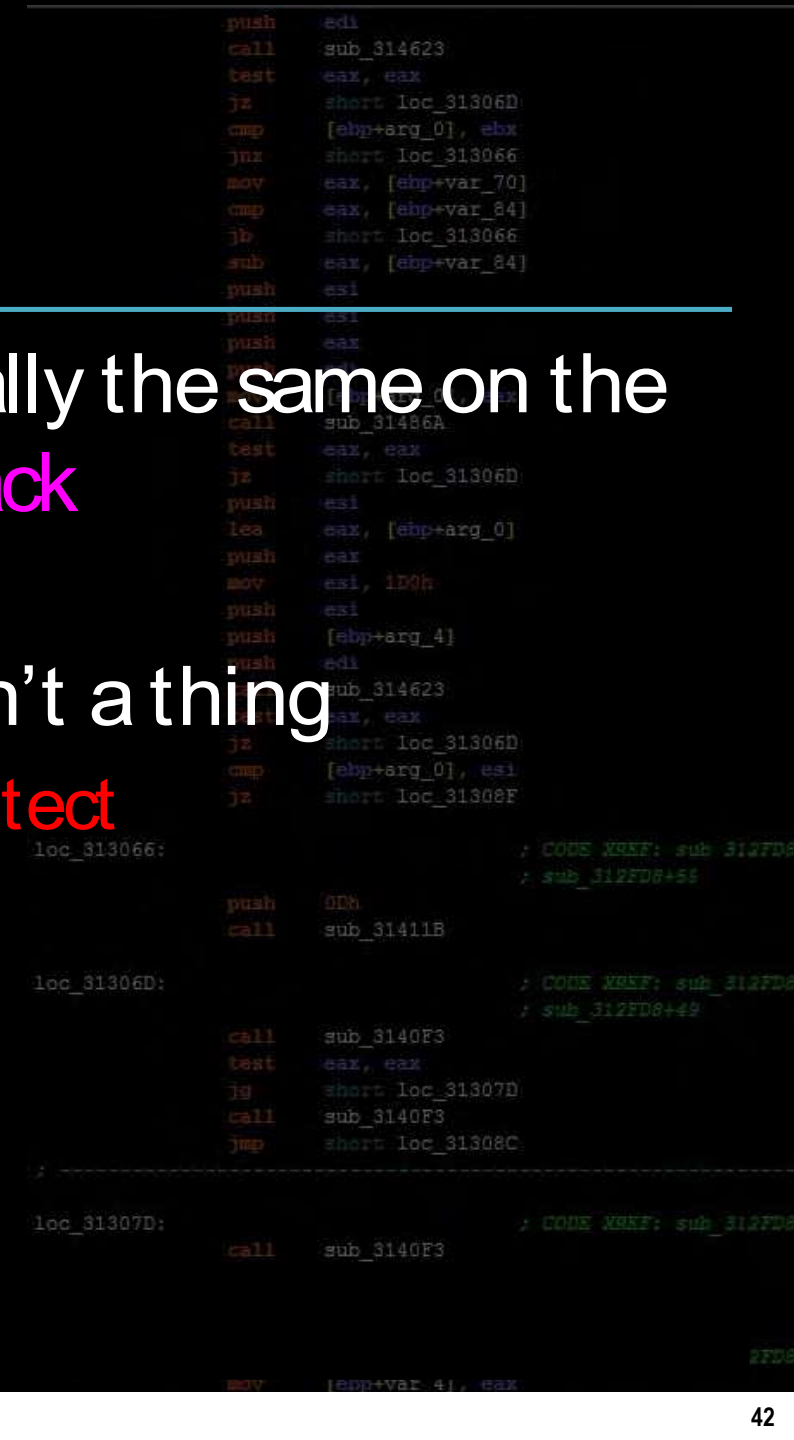


# Heap exploitation

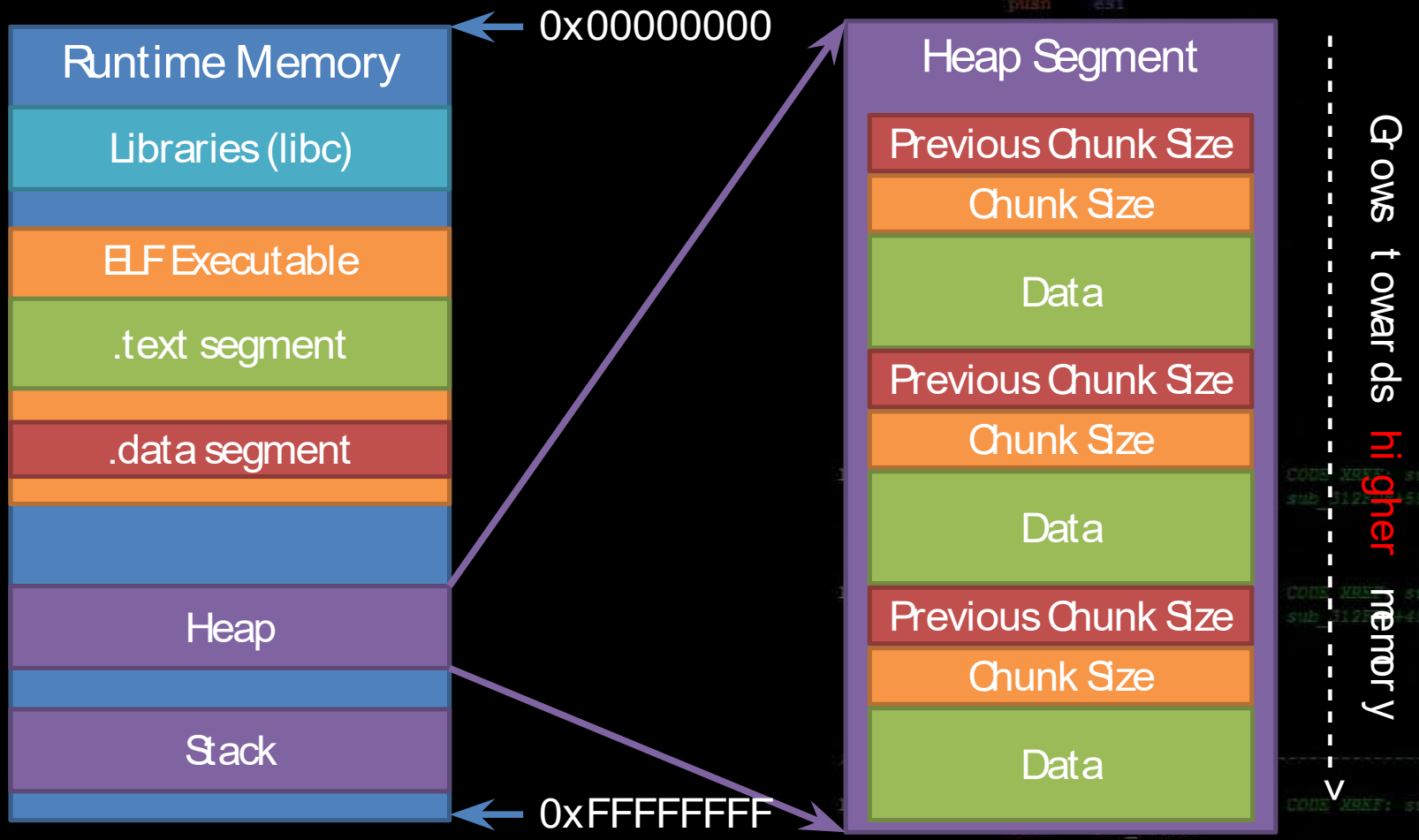
- Heap Overflows
- Use After Free
- Heap Spraying

# Heap Overflows

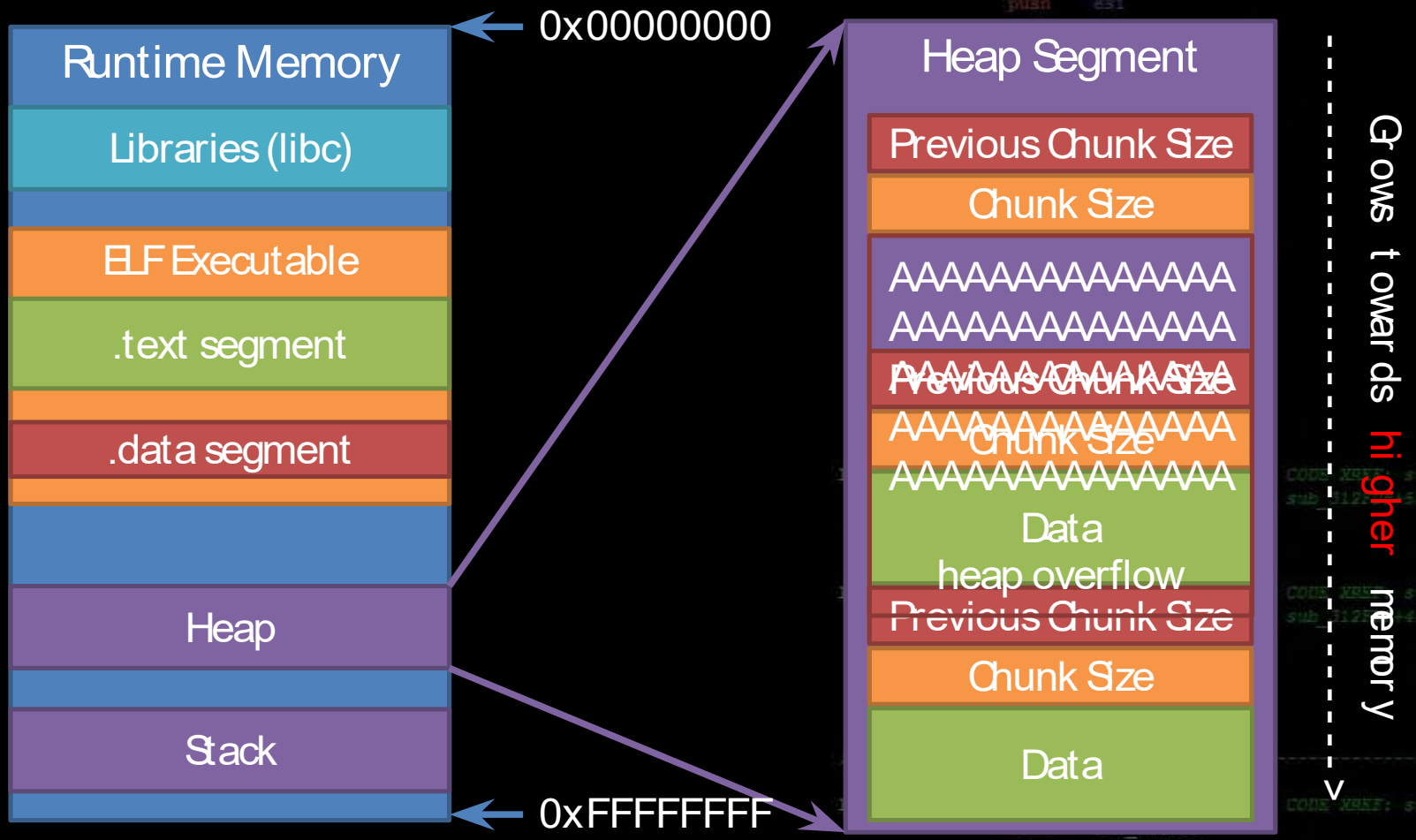
- Buffer overflows are basically the same on the heap as they are on the stack
- Heap cookies/canaries aren't a thing
  - No 'return' addresses to protect



# Heap Overflows



# Heap Overflows



# Heap Overflows

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

```
cool_guy = malloc(sizeof(struct toystr));
lame_guy = malloc(sizeof(struct toystr));
```

# Heap Overflows

```
cool guy = malloc(sizeof(struct toystr));
lame guy = malloc(sizeof(struct toystr));

cool guy->message = &print_cool;
lame guy->message = &print_meh;
```

# Heap Overflows

The C library function **strcspn()** calculates the length of the number of characters before the 1st occurrence of character present in the string.

```
cool guy = malloc(sizeof(struct toystr));
lame guy = malloc(sizeof(struct toystr));

cool guy->message = &print_cool;
lame guy->message = &print_meh;

printf("Input cool guy's name: ");
fgets(cool guy->buffer, 200, stdin);    // oopz...
cool guy->buffer[strcspn(cool guy->buffer, "\n")] = 0;
```

Silly heap overflow

# Heap Overflows

The C library function **strcspn()** calculates the length of the number of characters before the 1st occurrence of character present in the string.

```
cool guy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

cool guy->message = &print_cool;
lameguy->message = &print_meh;

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

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

Silly heap overflow

# Heap Overflows

The C library function **strcspn()** calculates the length of the number of characters before the 1st occurrence of character present in the string.

```
cool guy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

cool guy->message = &print_cool;
lameguy->message = &print_meh;

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

printf("Input lameguy's name: ");
fgets(lameguy->buffer, 20, stdin);
lameguy->message(lameguy->buffer);

cool guy->message(cool guy->buffer);
lameguy->message(lameguy->buffer);
```

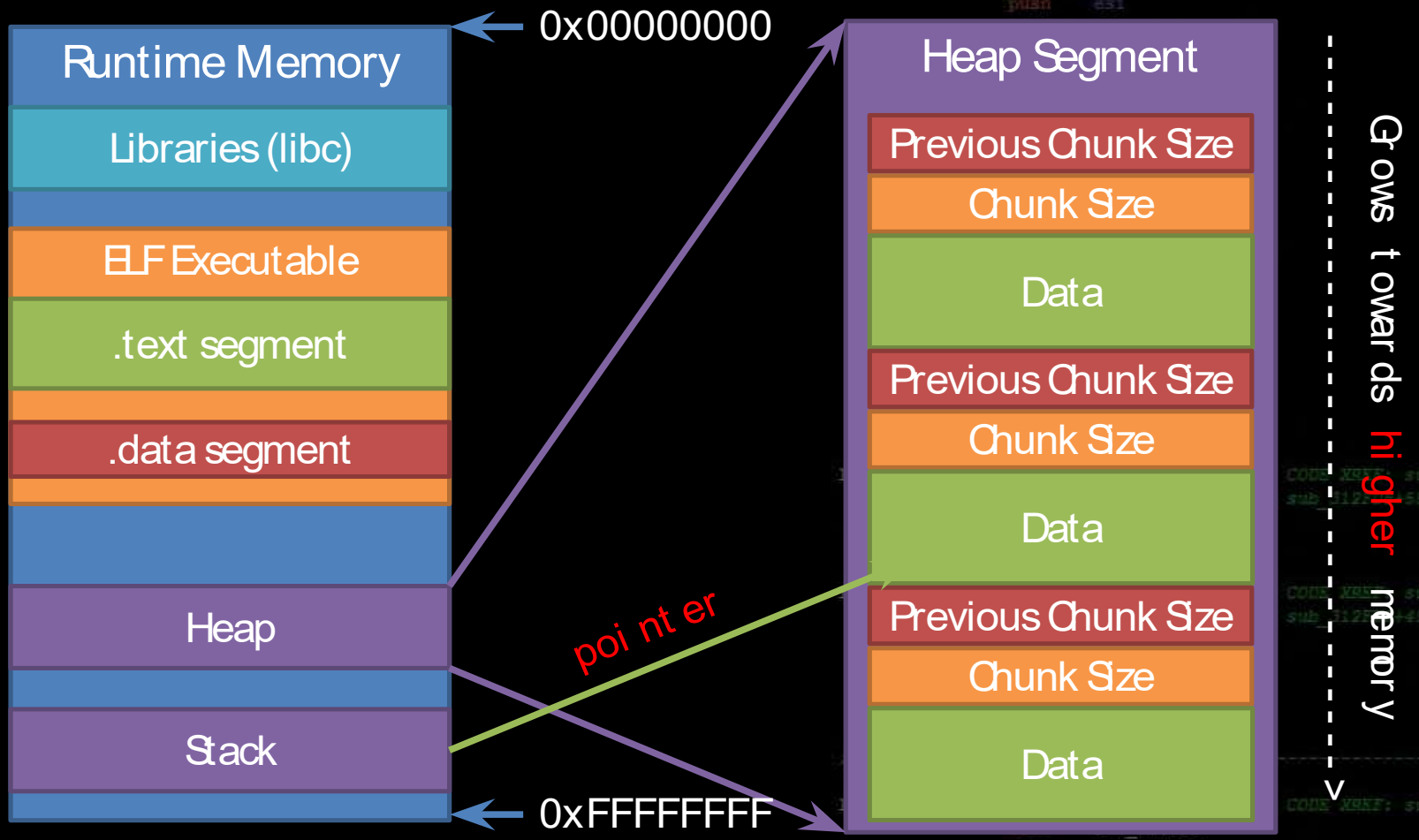
Silly heap overflow

Overwritten function pointer!

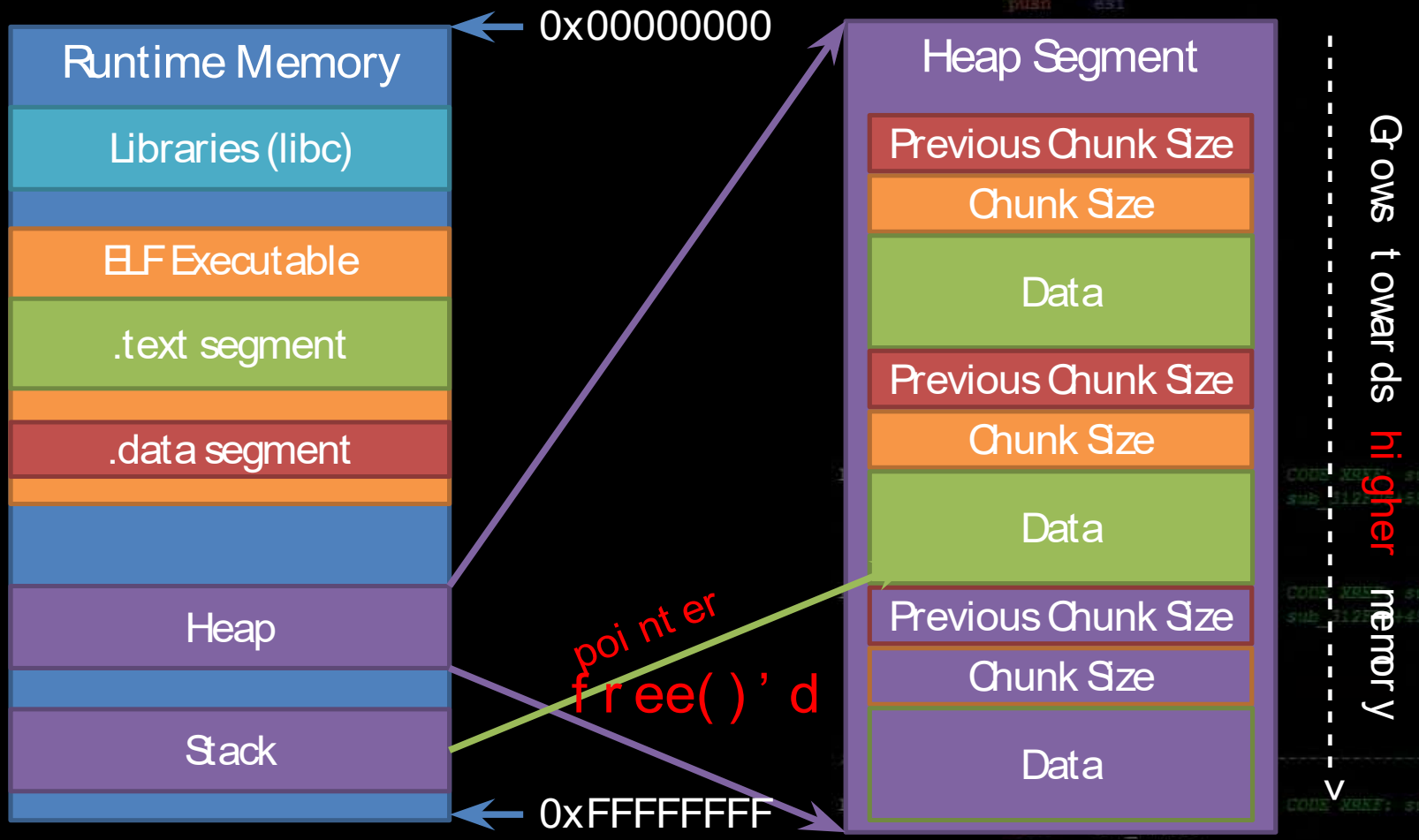
# Use-after-free attacks

- 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

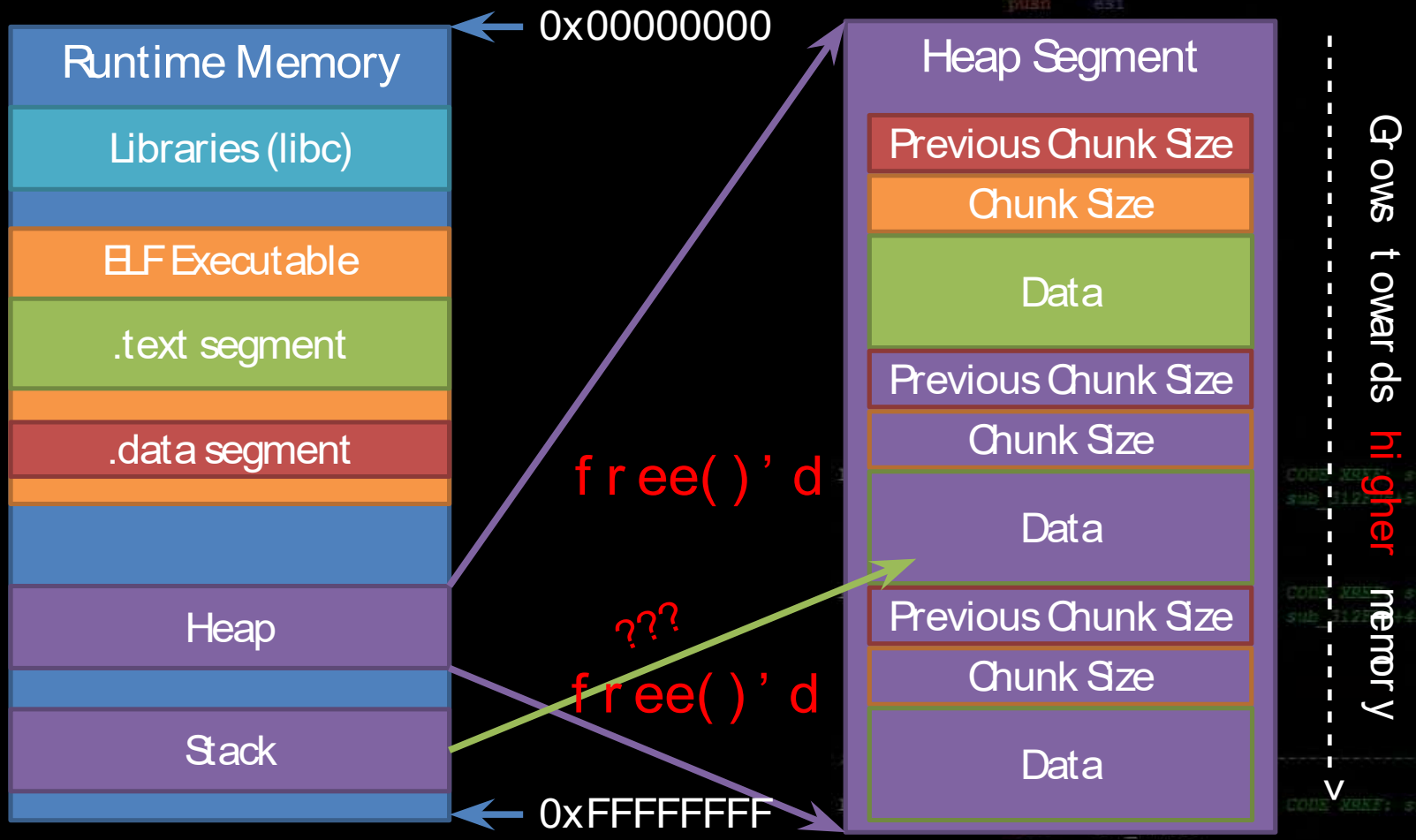
# Use After Free



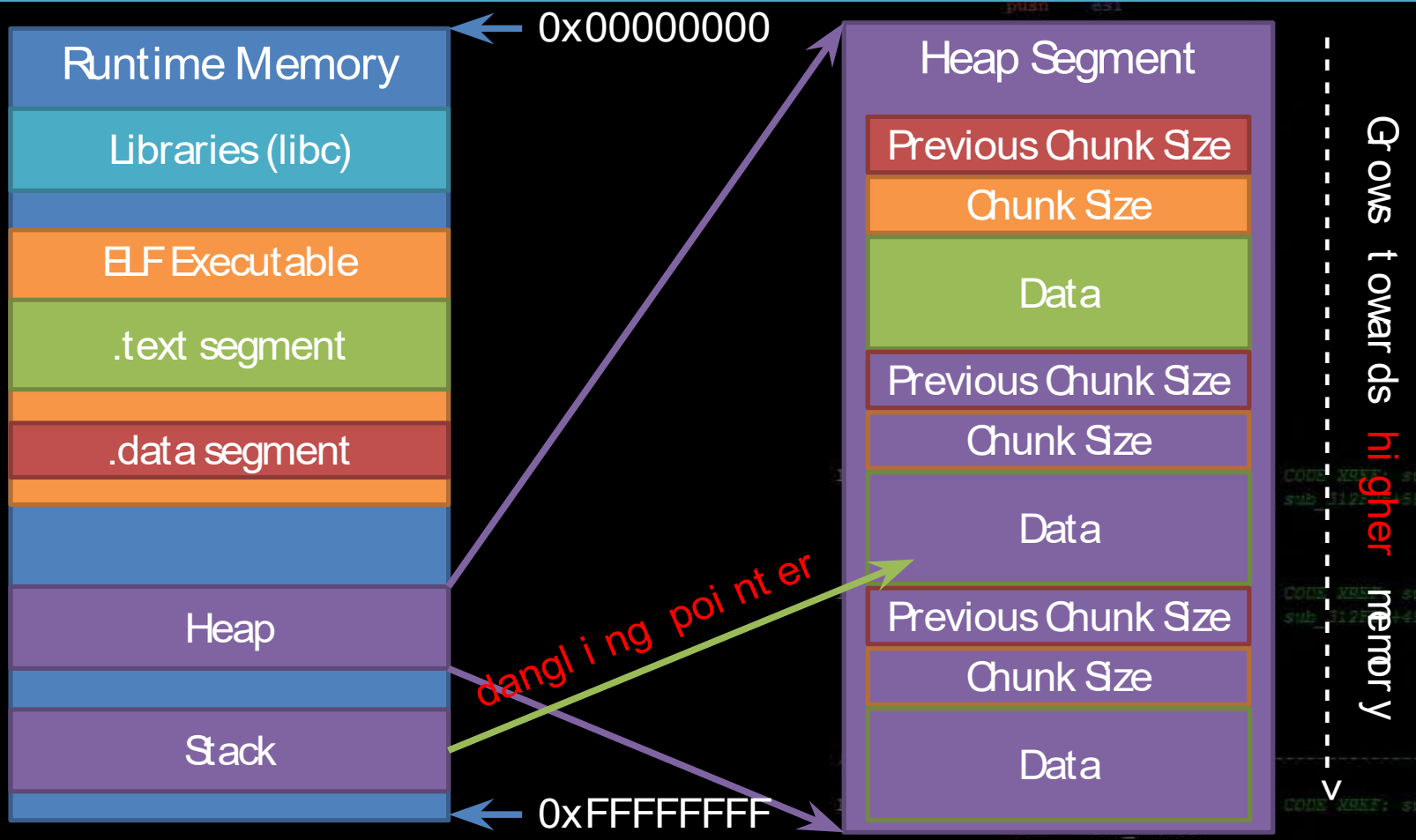
# Use After Free



# Use After Free



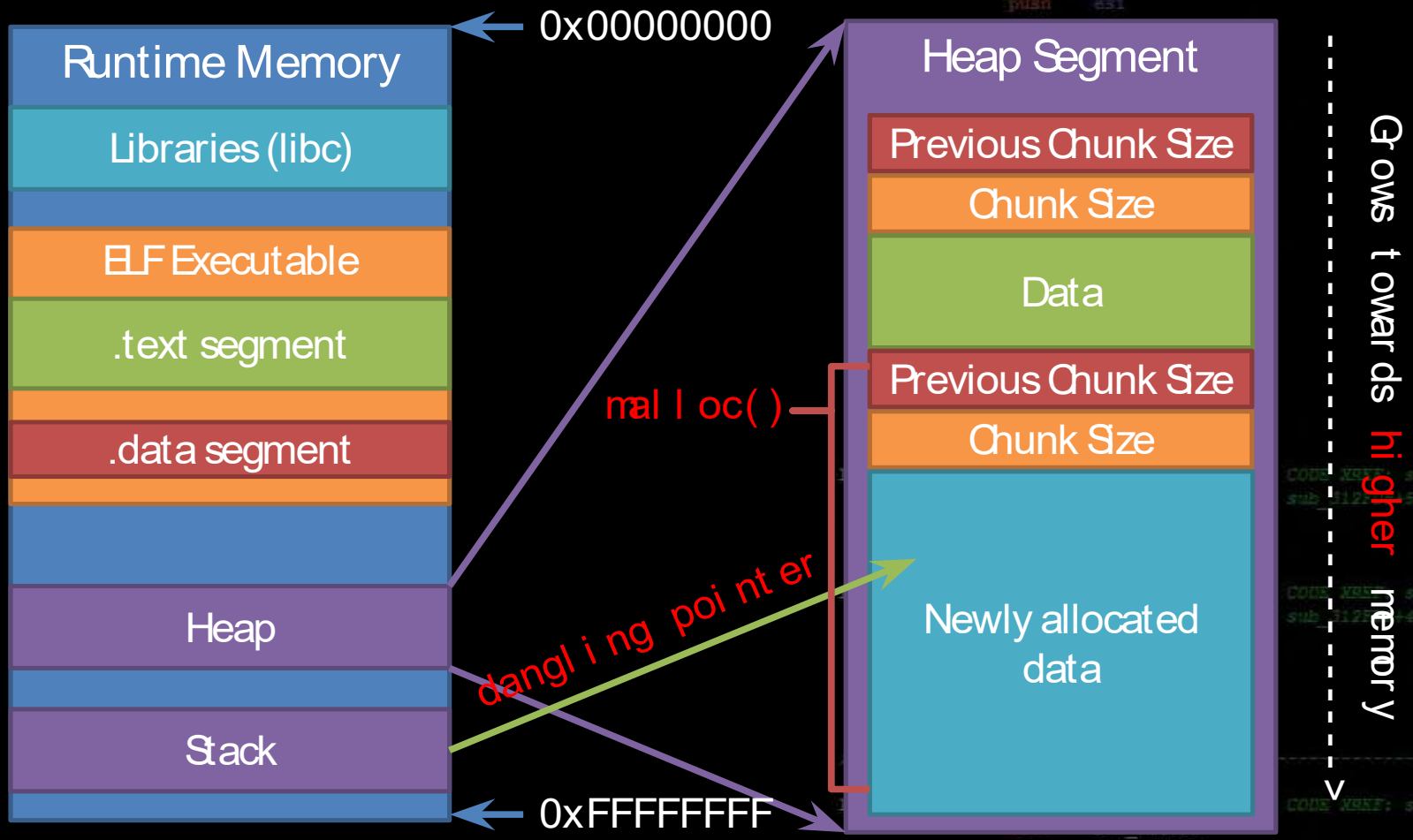
# Use After Free



# Dangling pointer

- A leftover 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



# 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()'

# 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

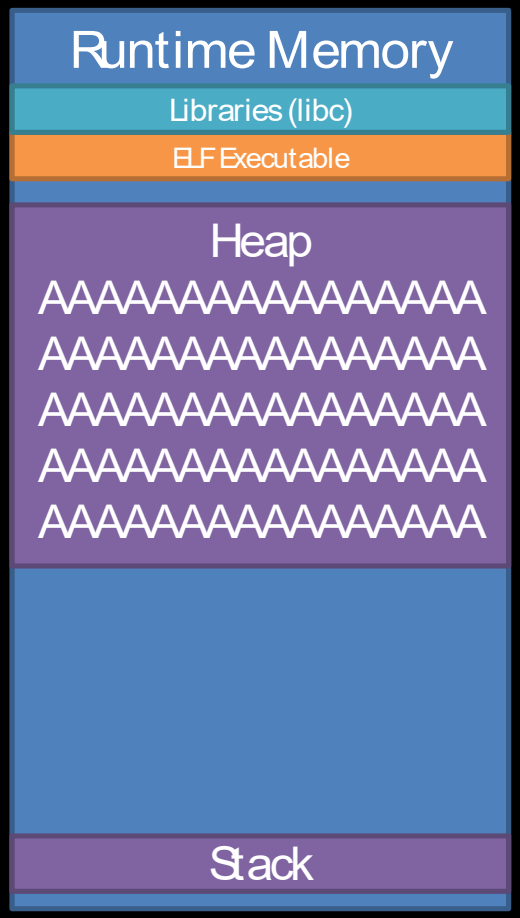
0x09104000 – Top of heap

```
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

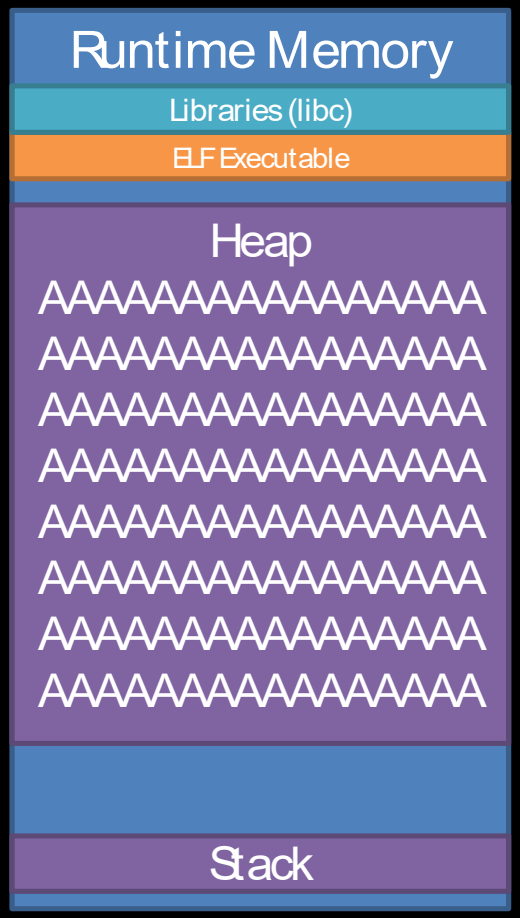
0x09104000 – Top of heap

```
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

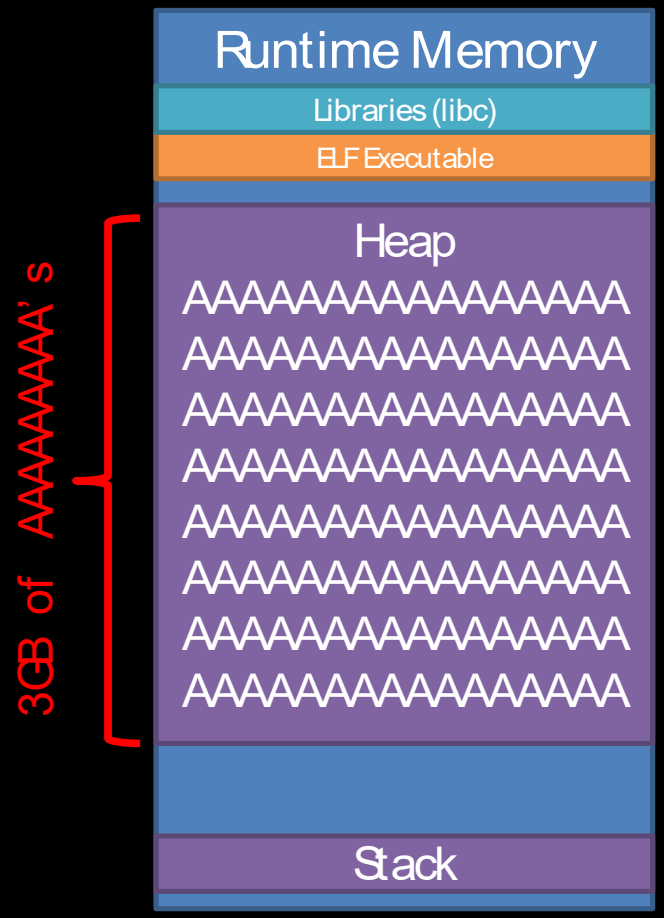
```
filler = "AAAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```

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

- 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)

- 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
  - 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

*End of Lecture 5*