SP2023 Week 04

PWN IV

Kevin

Announcements

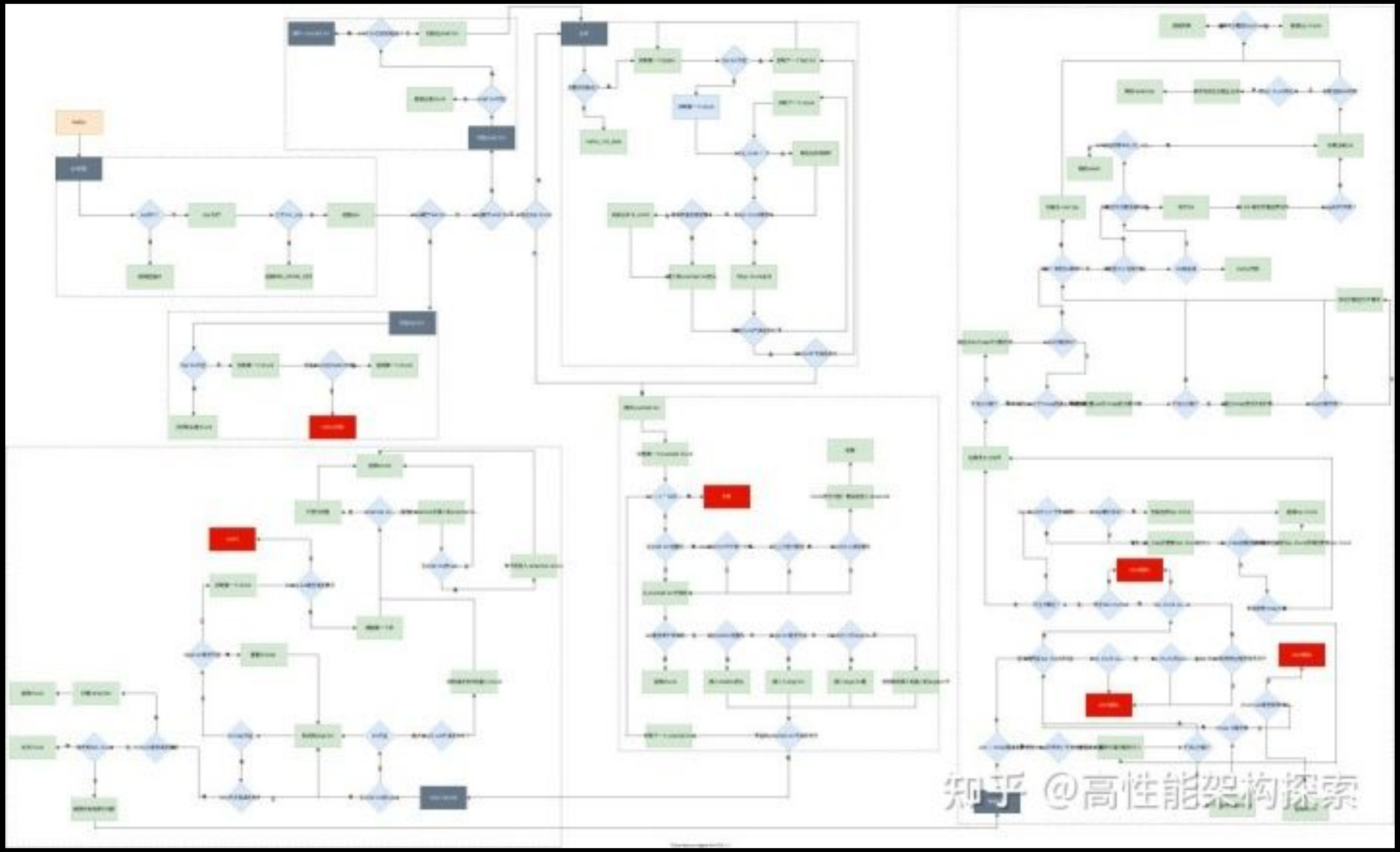
- WiCyS Palentines Day Social (2023-02-20) (**tomorrow!**)
    - Make crafts, eat snacks, meet friends!

- Cyber Tractor Challenge (application due 2023-03-13)
    - Travel to Des Moine to learn how to secure John Deere equipment

- ICSSP Informational Meeting (2023-03-02)
    - Scholarship and government internship opportunity
    - 5pm @ Siebel CS 2405

ctf.sigpwny.com

sigpwny{land_of_the_free}

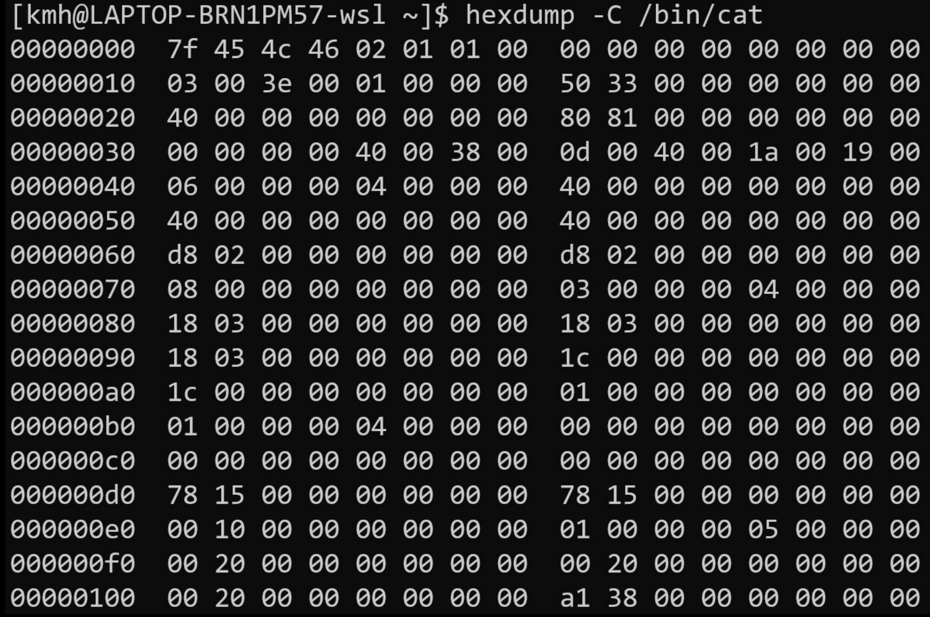


Review: what is pwn?

- More descriptive term: **binary exploitation**
- Exploits that abuse the mechanisms behind how compiled code is executed
- Most modern weaponized/valuable exploits fall under this category
- This is real stuff!!
    - Corollary: this is hard stuff. Ask for help, or if you don’t need help, help your neighbors :)

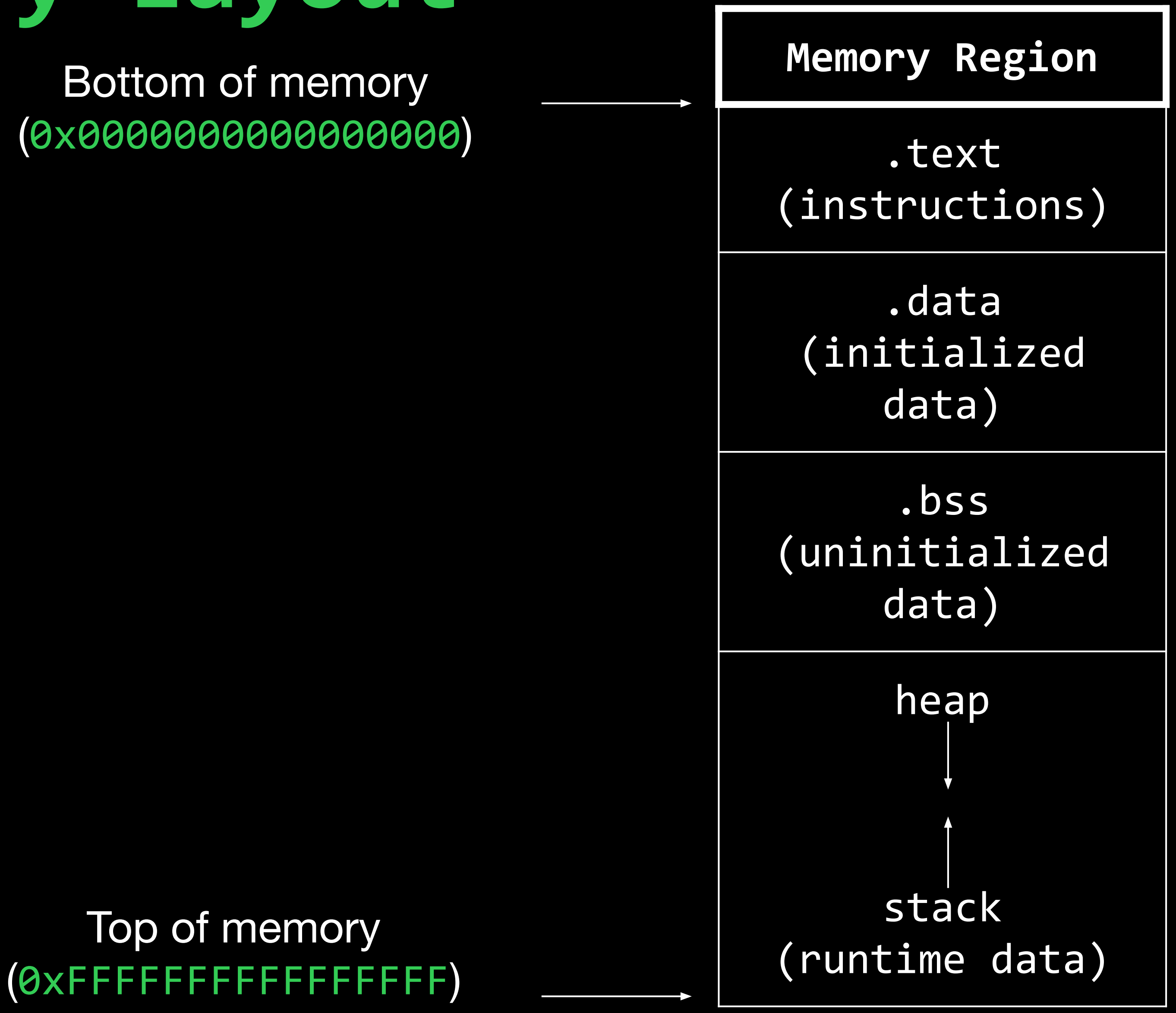
Memory Overview

- Programs are just a bunch of numbers ranging from 0 to 255 (**bytes**)
- Each number is stored at an "address" in the range `0x0-0xFFFFFFFFFFFFFFFF`
  - Think of it as a massive array/list
- Bytes in a program serves one of two purposes
  - **Instructions**: tells the processor what to do
  - **Data**: has some special meaning, used by the instructions
    - Examples: part of a larger number, a letter, a memory address



```
[kmh@LAPTOP-BRN1PM57-wsl ~]$ hexdump -C /bin/cat
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
00000010  03 00 3e 00 01 00 00 00  50 33 00 00 00 00 00 00
00000020  40 00 00 00 00 00 00 00  80 81 00 00 00 00 00 00
00000030  00 00 00 00 40 00 38 00  0d 00 40 00 1a 00 19 00
00000040  06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00
00000050  40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00
00000060  d8 02 00 00 00 00 00 00  d8 02 00 00 00 00 00 00
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00
00000080  18 03 00 00 00 00 00 00  18 03 00 00 00 00 00 00
00000090  18 03 00 00 00 00 00 00  1c 00 00 00 00 00 00 00
000000a0  1c 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00
000000b0  01 00 00 00 04 00 00 00  00 00 00 00 00 00 00 00
000000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000000d0  78 15 00 00 00 00 00 00  78 15 00 00 00 00 00 00
000000e0  00 10 00 00 00 00 00 00  01 00 00 00 05 00 00 00
000000f0  00 20 00 00 00 00 00 00  00 20 00 00 00 00 00 00
00000100  00 20 00 00 00 00 00 00  a1 38 00 00 00 00 00 00
```

Memory Layout



Bottom of memory
(0x0000000000000000)

Top of memory
(0xFFFFFFFFFFFFFFFF)

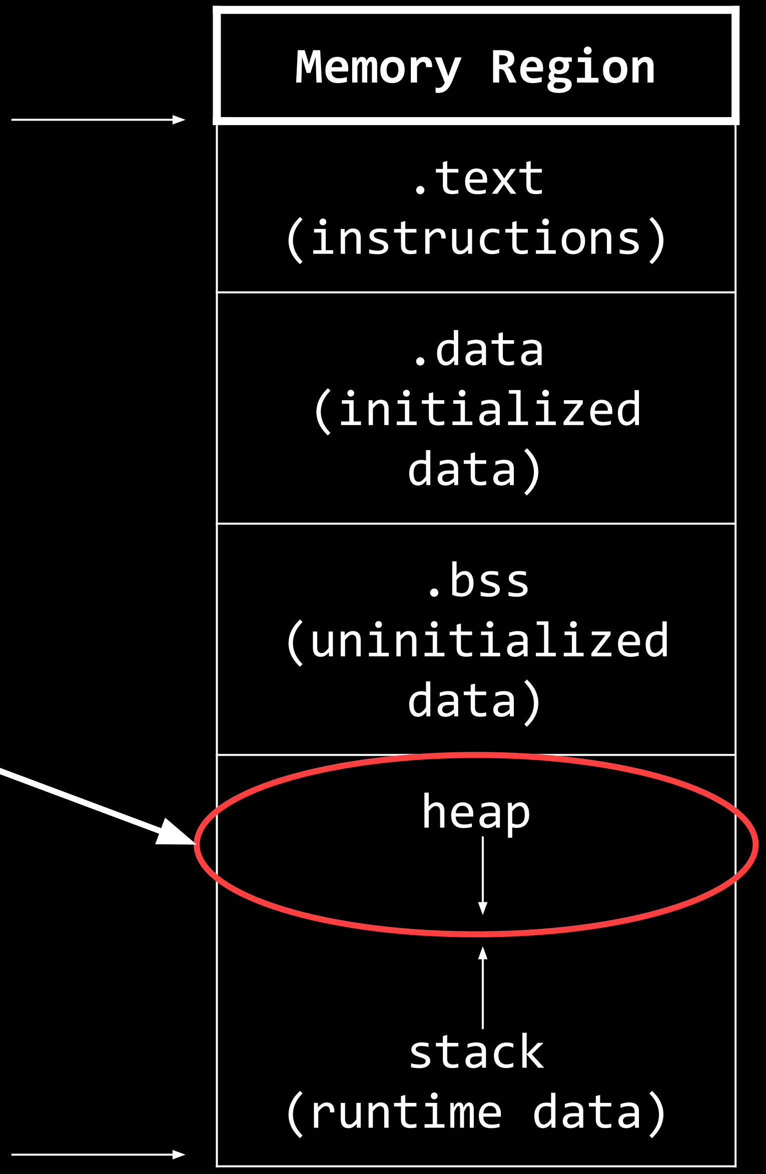
Memory Region
.text (instructions)
.data (initialized data)
.bss (uninitialized data)
heap ↓ ↑ stack (runtime data)

Memory Layout

Bottom of memory
(0x0000000000000000)

Today we’ll be
looking at the heap!

Top of memory
(0xFFFFFFFFFFFFFFFF)



Memory Region

.text
(instructions)

.data
(initialized
data)

.bss
(uninitialized
data)

heap

stack
(runtime data)

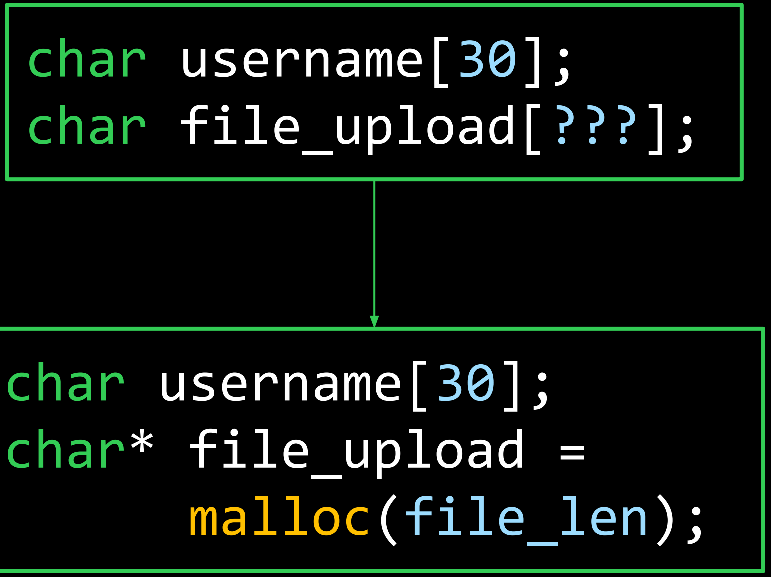
Dynamic Memory Allocation

- So far we've only seen static memory allocation
    - A **fixed amount of memory** is allocated on the stack or in .data or .bss
    - What if you don't know how much memory you'll need until runtime?

```
char username[30];
char file_upload[???];
```

Dynamic Memory Allocation

- So far we've only seen static memory allocation
  - A **fixed amount of memory** is allocated on the stack or in .data or .bss
  - What if you don't know how much memory you'll need until runtime?
- **malloc** allows you to manage memory **dynamically**
  - Request any amount of memory through a function call

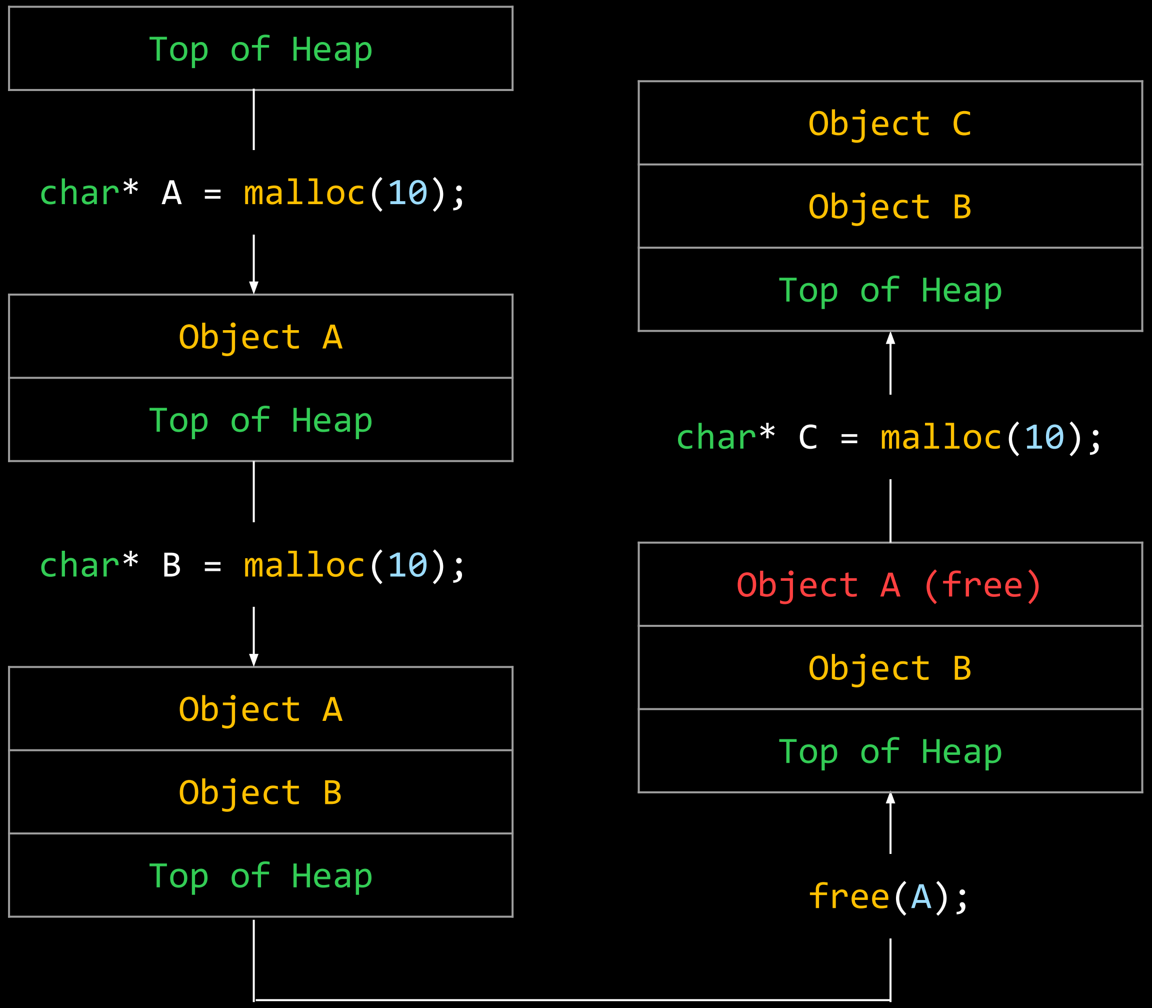


```c
char username[30];
char file_upload[???];
```

```c
char username[30];
char* file_upload =
      malloc(file_len);
```

Inside malloc

- malloc will allocate new memory at the top of the heap
- free can be used to mark memory as no longer in-use
    - malloc will reuse freed memory if possible
    - freed memory is placed in a free list
        - Don’t worry about this too much for now, just know that the most recently freed memory is used first



Top of Heap

```
char* A = malloc(10);
```

Object A

Top of Heap

```
char* B = malloc(10);
```

Object A

Object B

Top of Heap

```
free(A);
```

Object A (free)

Object B

Top of Heap

```
char* C = malloc(10);
```

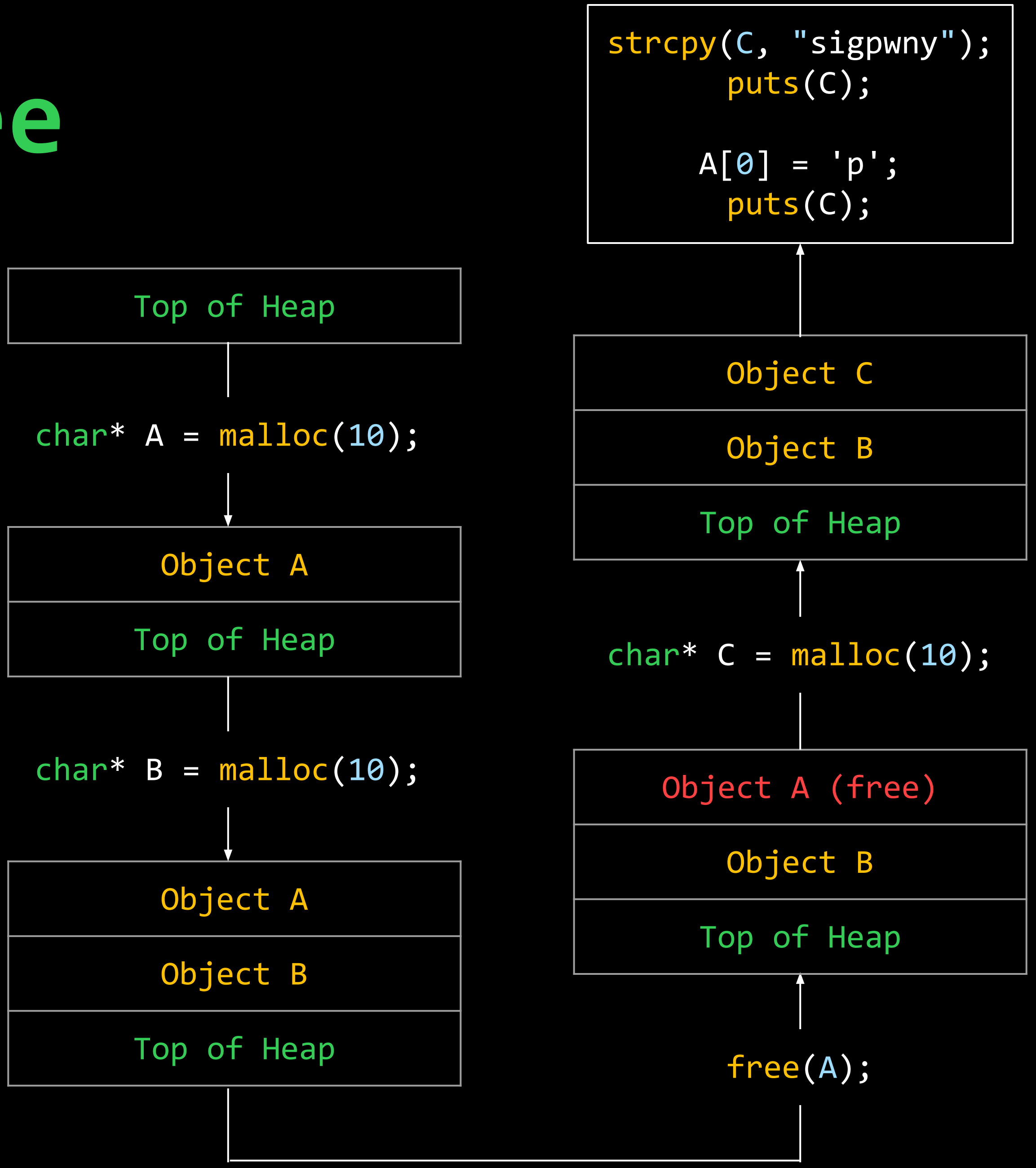
Object C

Object B

Top of Heap

Use After Free

- What happens if the programmer accidentally **uses freed memory?**



Top of Heap

```
char* A = malloc(10);
```

Object A

Top of Heap

```
char* B = malloc(10);
```

Object A

Object B

Top of Heap

```
free(A);
```

Object A (free)

Object B

Top of Heap

```
char* C = malloc(10);
```

Object C

Object B

Top of Heap

```
strcpy(C, "sigpwny");
      puts(C);

     A[0] = 'p';
      puts(C);
```

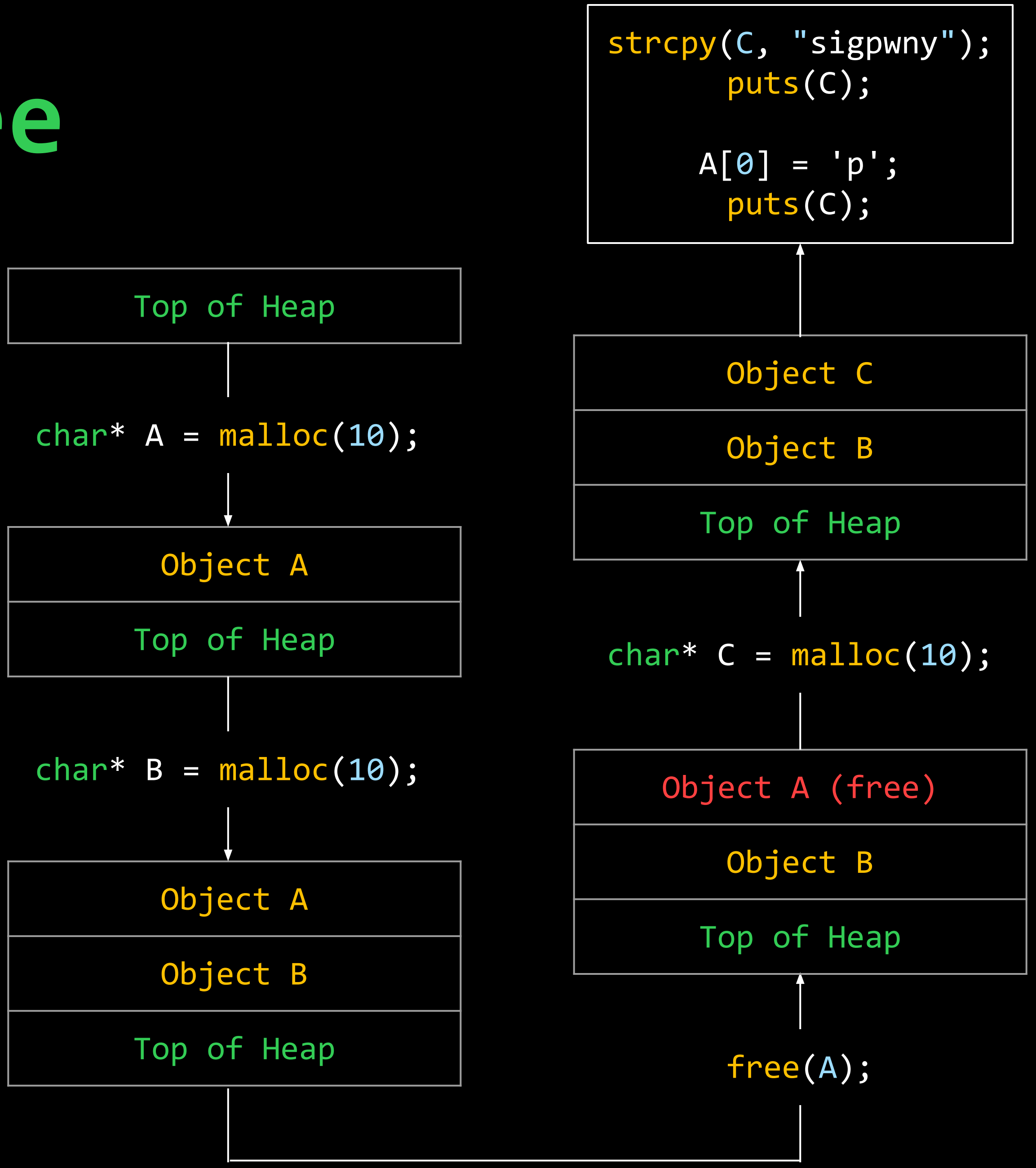
Use After Free

- What happens if the programmer accidentally **uses freed memory**?

Program Output:

```
sigpwny
pigpwny
```

Writing to A modifies C!



Top of Heap

```
char* A = malloc(10);
```

Object A

Top of Heap

```
char* B = malloc(10);
```

Object A

Object B

Top of Heap

```
free(A);
```

Object A (free)

Object B

Top of Heap

```
char* C = malloc(10);
```

Object C

Object B

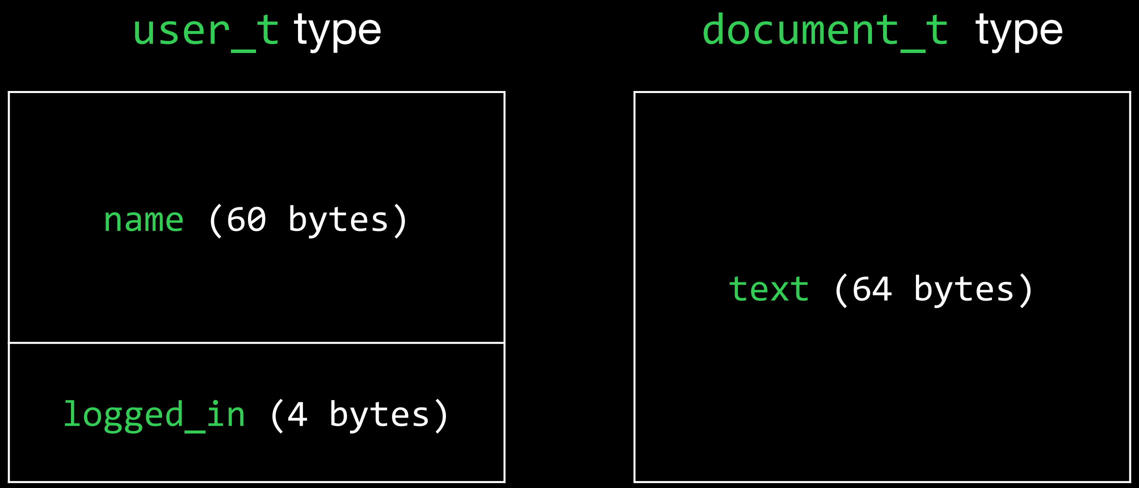
Top of Heap

```
strcpy(C, "sigpwny");
      puts(C);

     A[0] = 'p';
      puts(C);
```

Challenge Introduction

- You can create **users** and **documents**, which are allocated on the heap
- Goal: become “admin”, and run `shell`



`user_t` type

`name` (60 bytes)

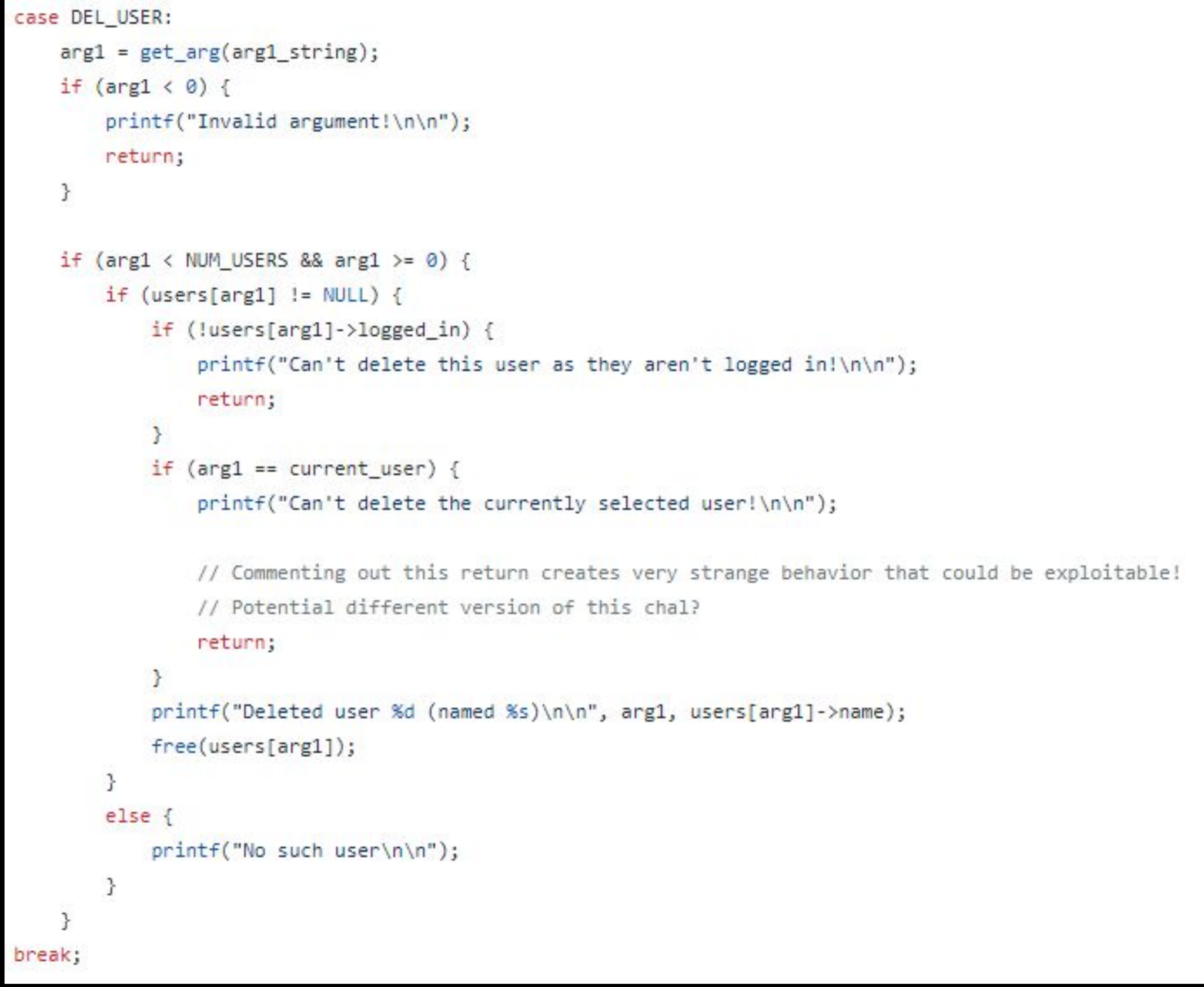
`logged_in` (4 bytes)

`document_t` type

`text` (64 bytes)

Important: these are the same size!

Heap 0 and 1 Bug



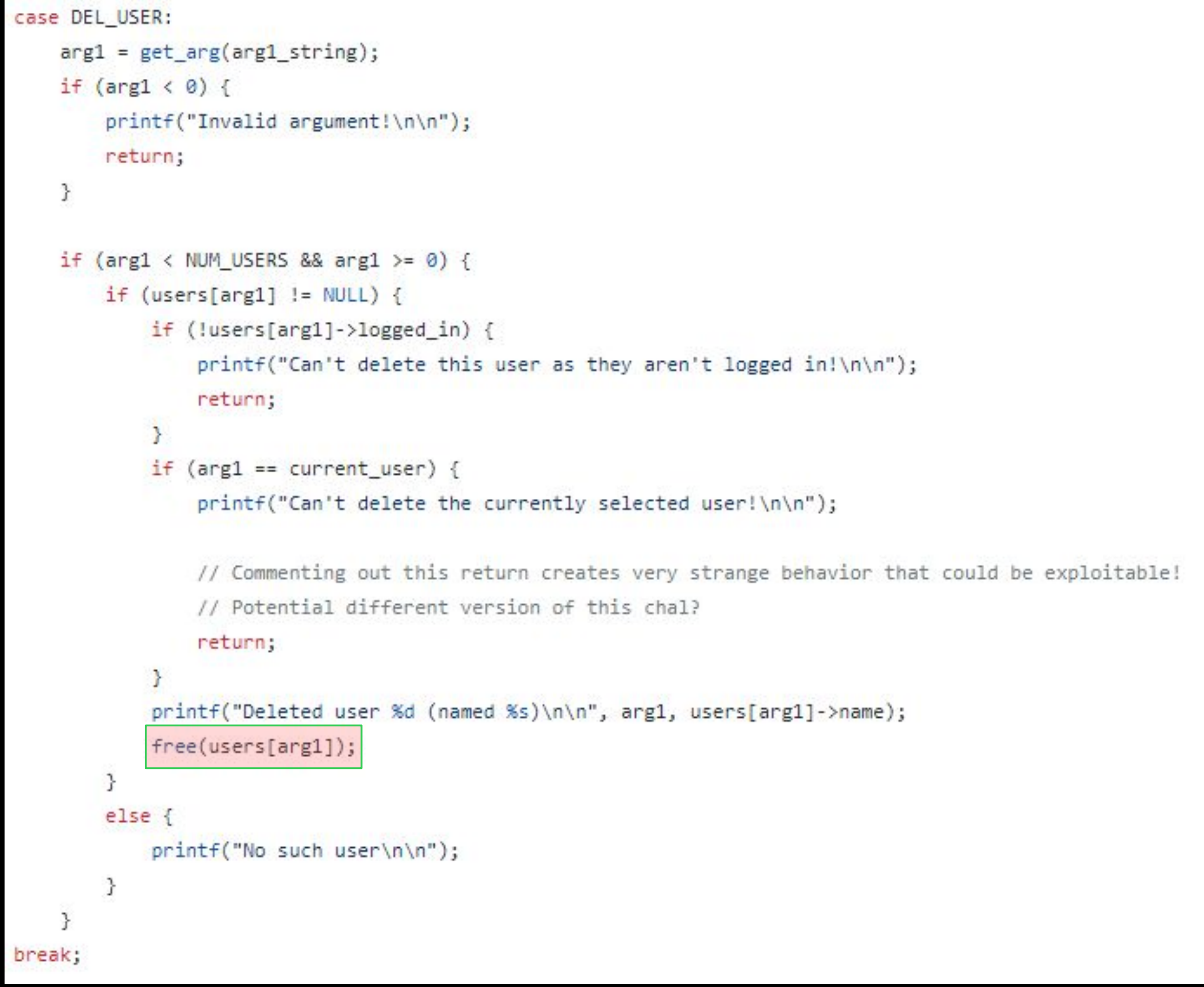
```
case DEL_USER:
    arg1 = get_arg(arg1_string);
    if (arg1 < 0) {
        printf("Invalid argument!\n\n");
        return;
    }

    if (arg1 < NUM_USERS && arg1 >= 0) {
        if (users[arg1] != NULL) {
            if (!users[arg1]->logged_in) {
                printf("Can't delete this user as they aren't logged in!\n\n");
                return;
            }
            if (arg1 == current_user) {
                printf("Can't delete the currently selected user!\n\n");

                // Commenting out this return creates very strange behavior that could be exploitable!
                // Potential different version of this chal?
                return;
            }
            printf("Deleted user %d (named %s)\n\n", arg1, users[arg1]->name);
            free(users[arg1]);
        }
        else {
            printf("No such user\n\n");
        }
    }
break;
```

Heap 0 and 1 Bug

- User is freed, but not removed from the array
- The freed memory can still be used by other operations (**use after free**)



```
case DEL_USER:
    arg1 = get_arg(arg1_string);
    if (arg1 < 0) {
        printf("Invalid argument!\n\n");
        return;
    }

    if (arg1 < NUM_USERS && arg1 >= 0) {
        if (users[arg1] != NULL) {
            if (!users[arg1]->logged_in) {
                printf("Can't delete this user as they aren't logged in!\n\n");
                return;
            }
            if (arg1 == current_user) {
                printf("Can't delete the currently selected user!\n\n");

                // Commenting out this return creates very strange behavior that could be exploitable!
                // Potential different version of this chal?
                return;
            }
            printf("Deleted user %d (named %s)\n\n", arg1, users[arg1]->name);
            free(users[arg1]);
        }
        else {
            printf("No such user\n\n");
        }
    }
break;
```

Next Meetings

2023-02-20 **- This Monday**

- WiCyS Palentines Day Social
- Make crafts and eat snacks with Women in Cybersecurity

2023-02-23 **- Next Thursday**

- REV III with Richard
- Learn about VM obfuscation and side channels

2023-02-26 **- Next Sunday**

- Nintendo DSi Browser Exploit
- Nathan will share how he hacked the DSi web browser

Challenges!

- Meeting flag:
    - `sigpwny{land_of_the_free}`
- Go through the challenges in the PWN IV category
    - This lecture provided the necessary knowledge for Heap 0 through 2
    - Heap 0-2 can be solved by directly interacting with the server (i.e. no echo or pwntools)
- **Don't** test by running the binaries locally!
    - Your malloc version may be different from the server's.

SIGPwny