binary_exploitation.c

workshop.c

```
1
2   Workshop ‘Binary Exploitation’{
3
4
5       [stack, heap]
6
7
8
9       char speakers[] = "Dinis & Lucas";
10
11
12  }
13
14
```

Setup Guide {

< GDB >

< sudo apt-get install gdb >
< bash -c "$(curl -fsSL http://gef.blah.cat/sh)" >

< Pwn-tools >

< apt-get update >
< apt-get install python3 python3-pip python3-dev git libssl-dev libffi-dev build-essential >
< python3 -m pip install --upgrade pip >
< python3 -m pip install --upgrade pwntools >

< gcc-multilib >

< apt-get install gcc-multilib >

< Disable ASLR >

< echo 0 > /proc/sys/kernel/randomize_va_space >

< Python2 >

< apt install python2 >

}

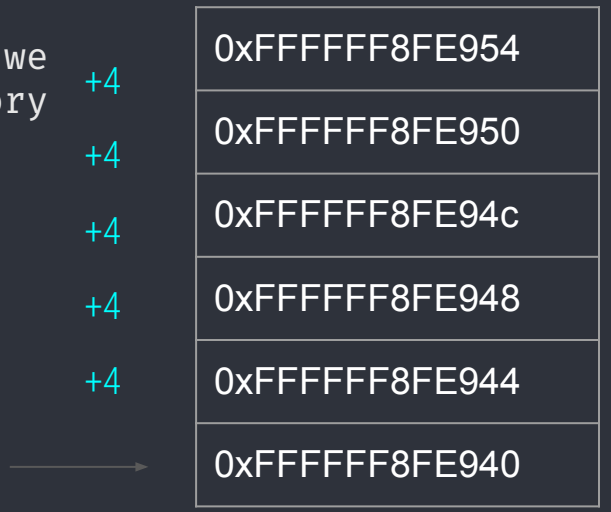
```
01 {

   [stack]

      < Theory >

}
```

Memory access in c {

< How does memory access work? >

< Having a pointer to somewhere we can access different parts of memory (Valid parts)>

}

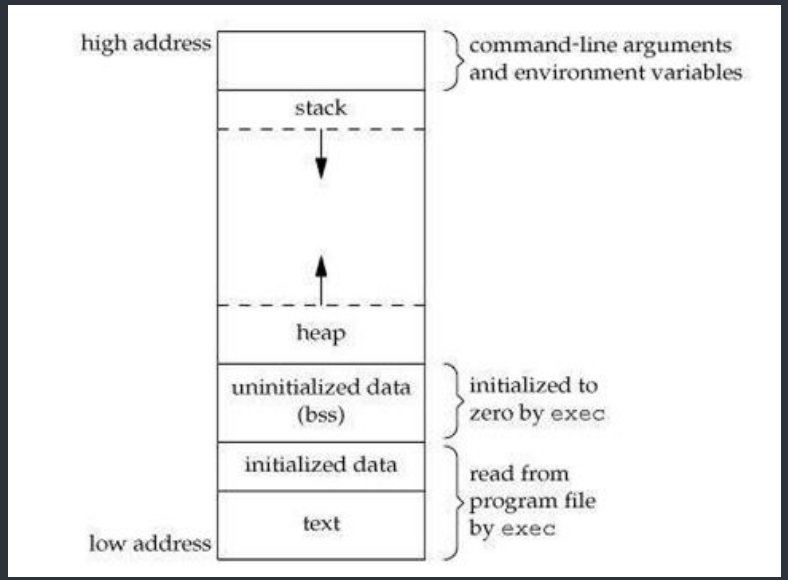


C memory layout {

< Stack grows down >

< Heap grows up >

}



Stack organization {

< Stack has 1 frame for each function call >

< Each function will store:

- Returning information (Address and frame)
- Local variables declared inside the function
- Arguments to possible function calls >

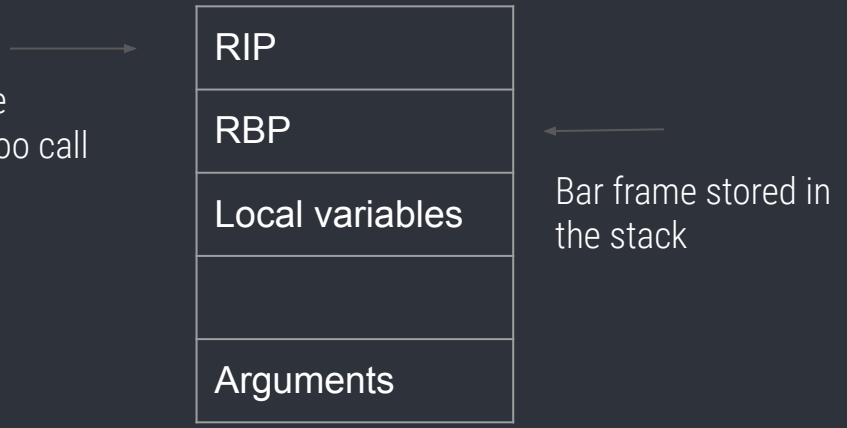
}

Function chaining example {

< Return information:

- RIP → Address to the function we are returning to so the code knows what to execute next.
- RBP → Frame of the function that called this function. >

< Foo is returning from bar >

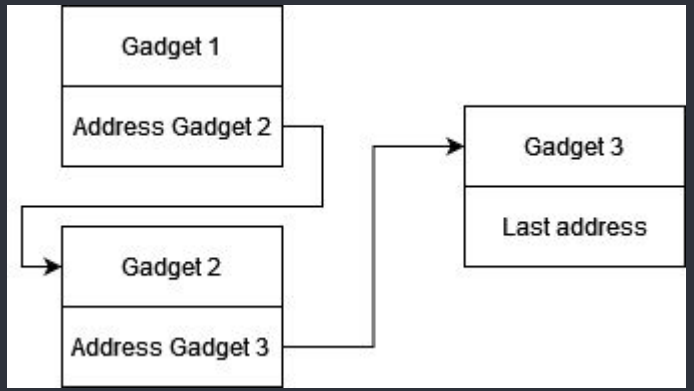


}

ROP {

< Return oriented programming:

- Small pieces of code with a return after.
- With this return we can jump to another piece with another return and so on. >



}

Return to libc {

< So what's the payload:

- Small pieces of code with a return after.
- With this return we can jump to another piece with another return and so on. >

}

```
03 {
    [heap]
        < Theory >
}
```

What is heap {

< Used to store dynamically allocated variables >

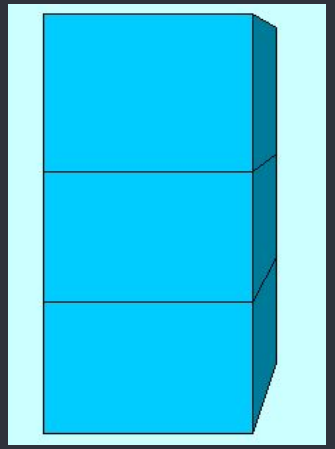
< Mandatory control of memory >

< “Infinite” memory >

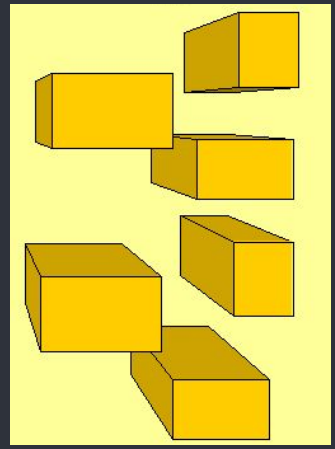
< Memory allocation functions (in C):
- malloc()
- calloc()
- realloc() >

< Memory release functions (in C):
- free() >

}



‘STACK’

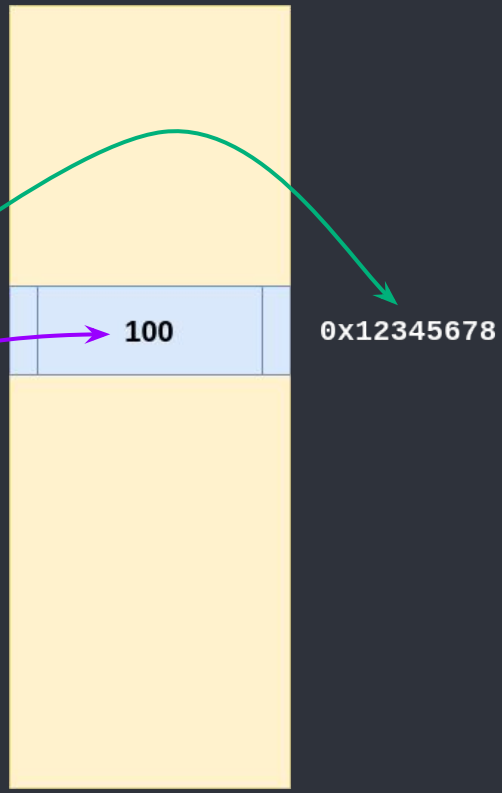


‘HEAP’

How to use heap? {

```c
#include<stdio.h>
#include<stdlib.h>

int main() {
        int *num;
        num = (int*) malloc(sizeof(int));

        *num = 100;

        printf("%d", *num);

}
```

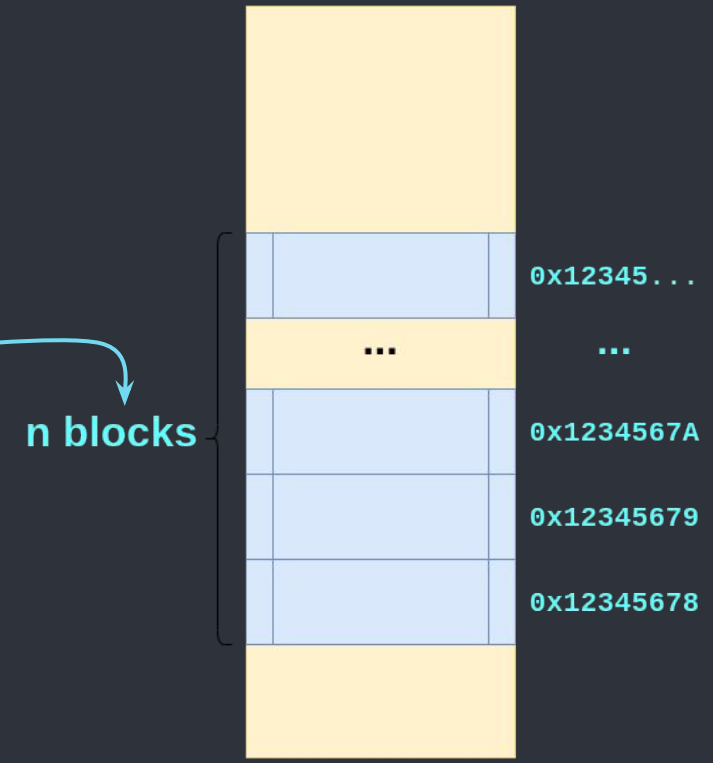
}



But really… why? {

```c
int main() {
        int *array;
        int n, i;

        // Get the number of elements for the array
        printf("Enter number of elements:");
        scanf("%d", &n);

        array = (int*)malloc(n * sizeof(int));

        // fill array with [1, 2, 3, ...]
        for (i = 0; i < n; ++i) {
            array[i] = i + 1;
        }

        // Print array
        printf("array: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", array[i]);
        }
        return 0;
}
```

}



binary_exploitation.c

workshop.c

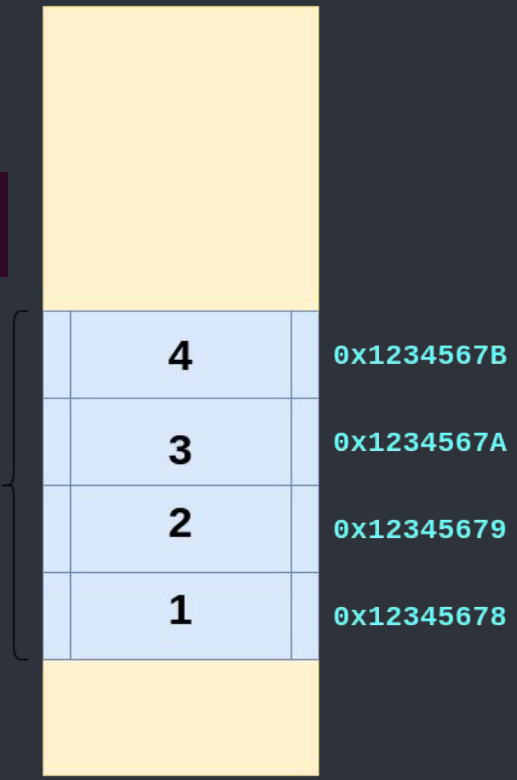
But really… why? {

```
botto@botto:~/Desktop/workshop_stack_heap_overflow$ ./heap_array
Enter number of elements:4
array: 1, 2, 3, 4,
```

< Runtime memory allocation >

< Customizable size of array >

}



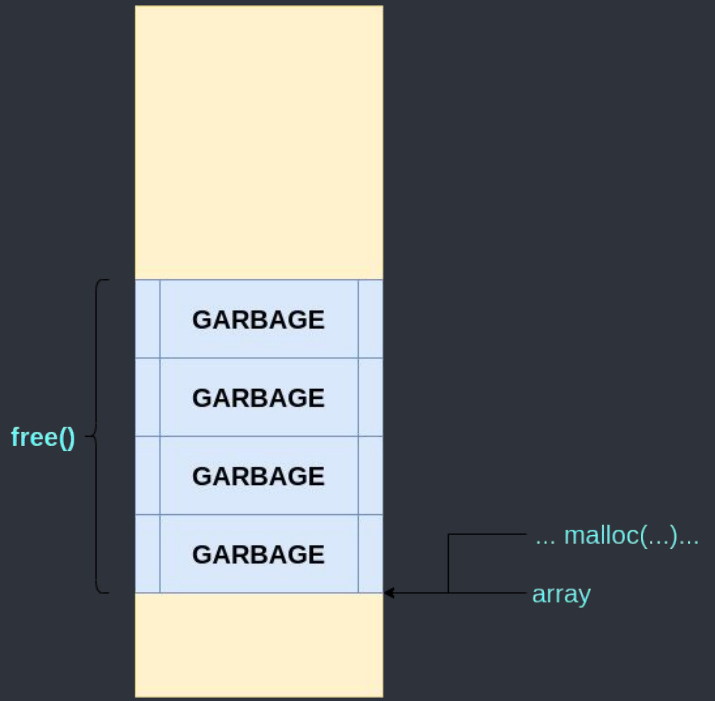
free() is shady… {

< Good:
- clears memory block
- reutilization of memory>

< Bad:
- doesn’t remove pointer from freed variable, leading to duplicated pointers >

< Fastbin: list of recently freed addresses >

}



Stack vs Heap {

< Stack:

- Temporary memory (values accessible inside the function scope).
- Automatic memory management
- Faster
- Less storage space

>

< Heap:

- Persistent data.
- Manual memory management
- Slower
- Much more storage (size of system RAM)
- Complex/dynamic structures

>

}

```
04 {

   [heap overflow]

      < Practice >

}
```