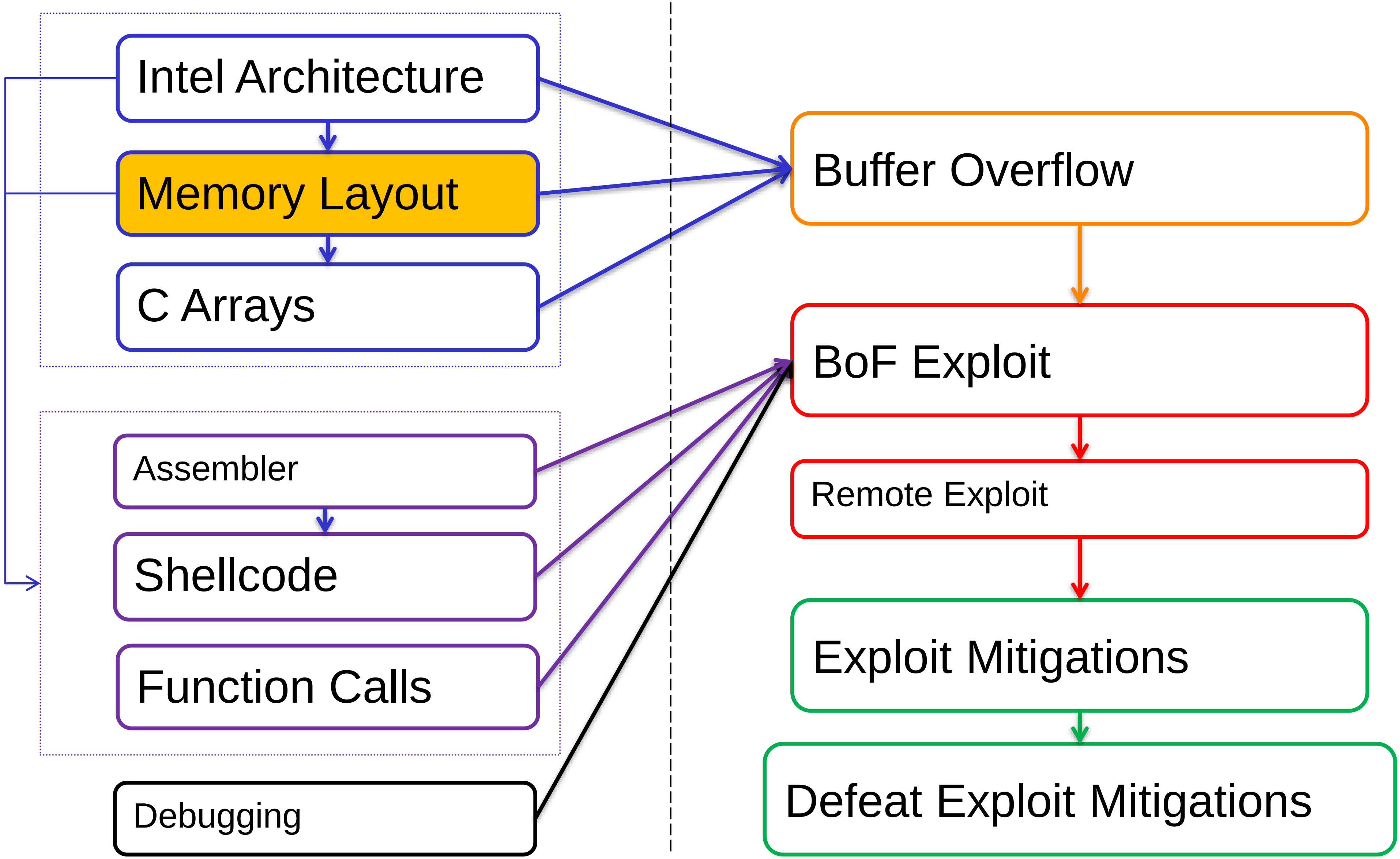
Memory Layout

Linux Userspace Process Memory Layout

Content


Intel Architecture

Memory Layout

C Arrays

Assembler

Shellcode

Function Calls

Debugging

Buffer Overflow

BoF Exploit

Remote Exploit

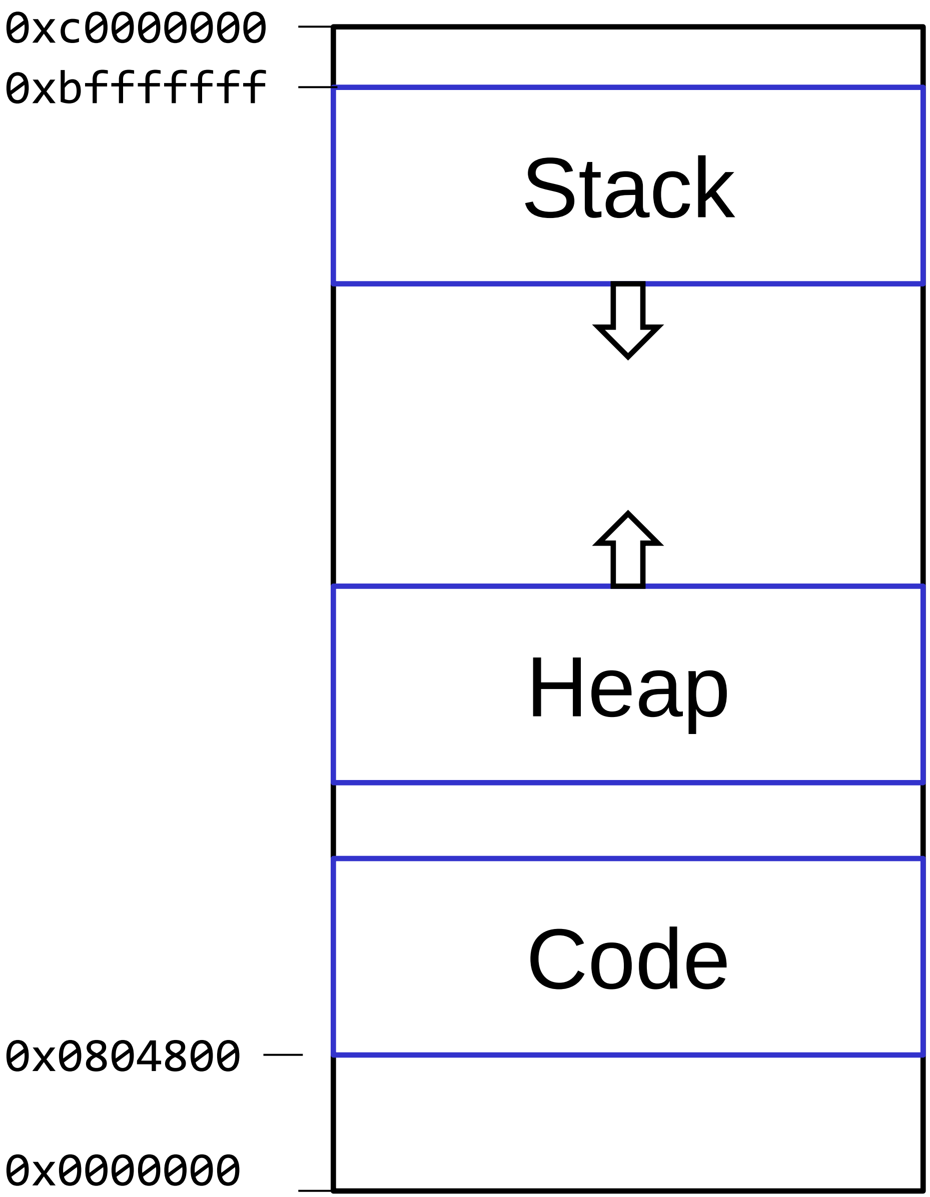
Exploit Mitigations

Defeat Exploit Mitigations

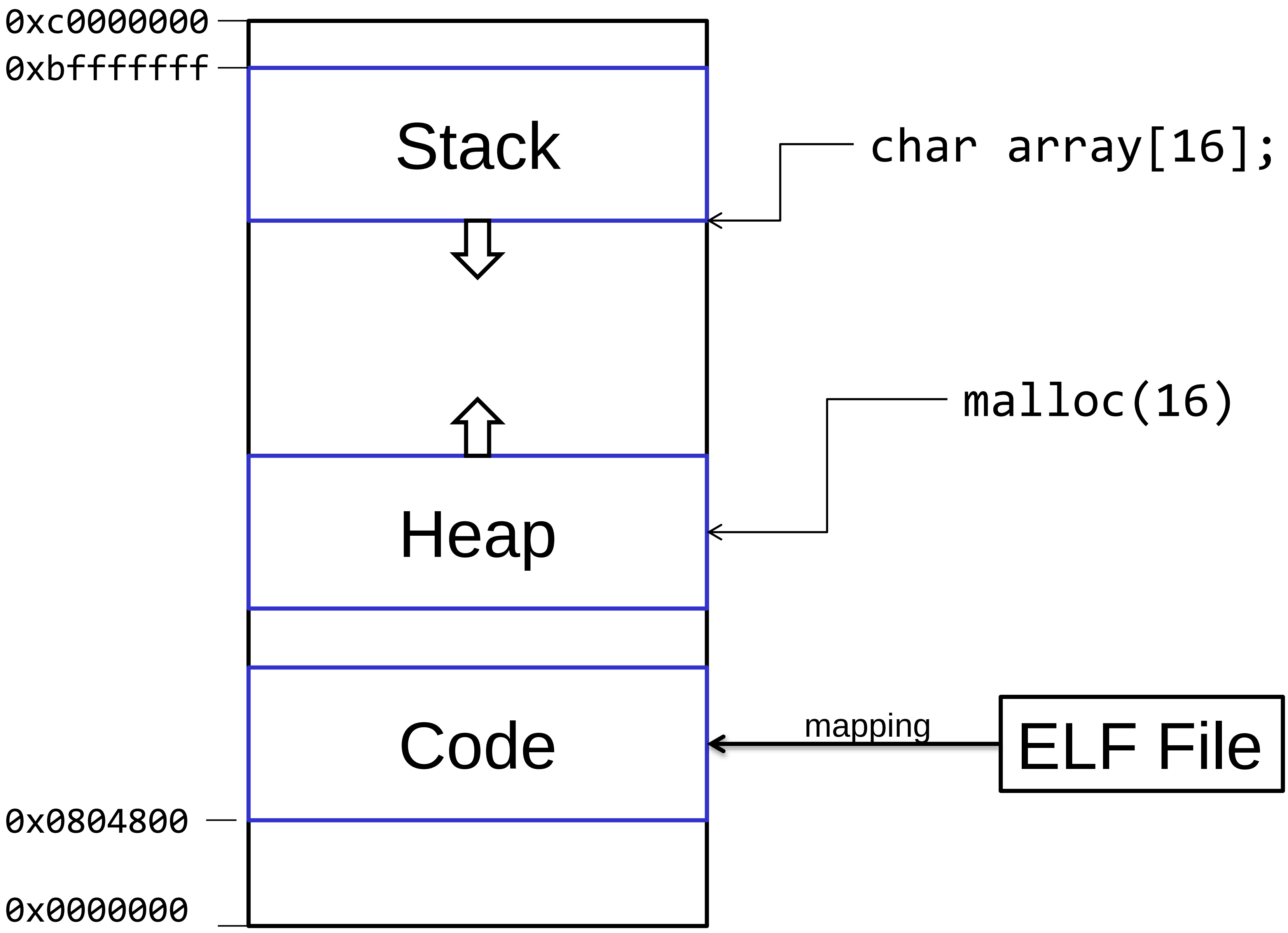
Userspace Memory Layout

In 32 bit

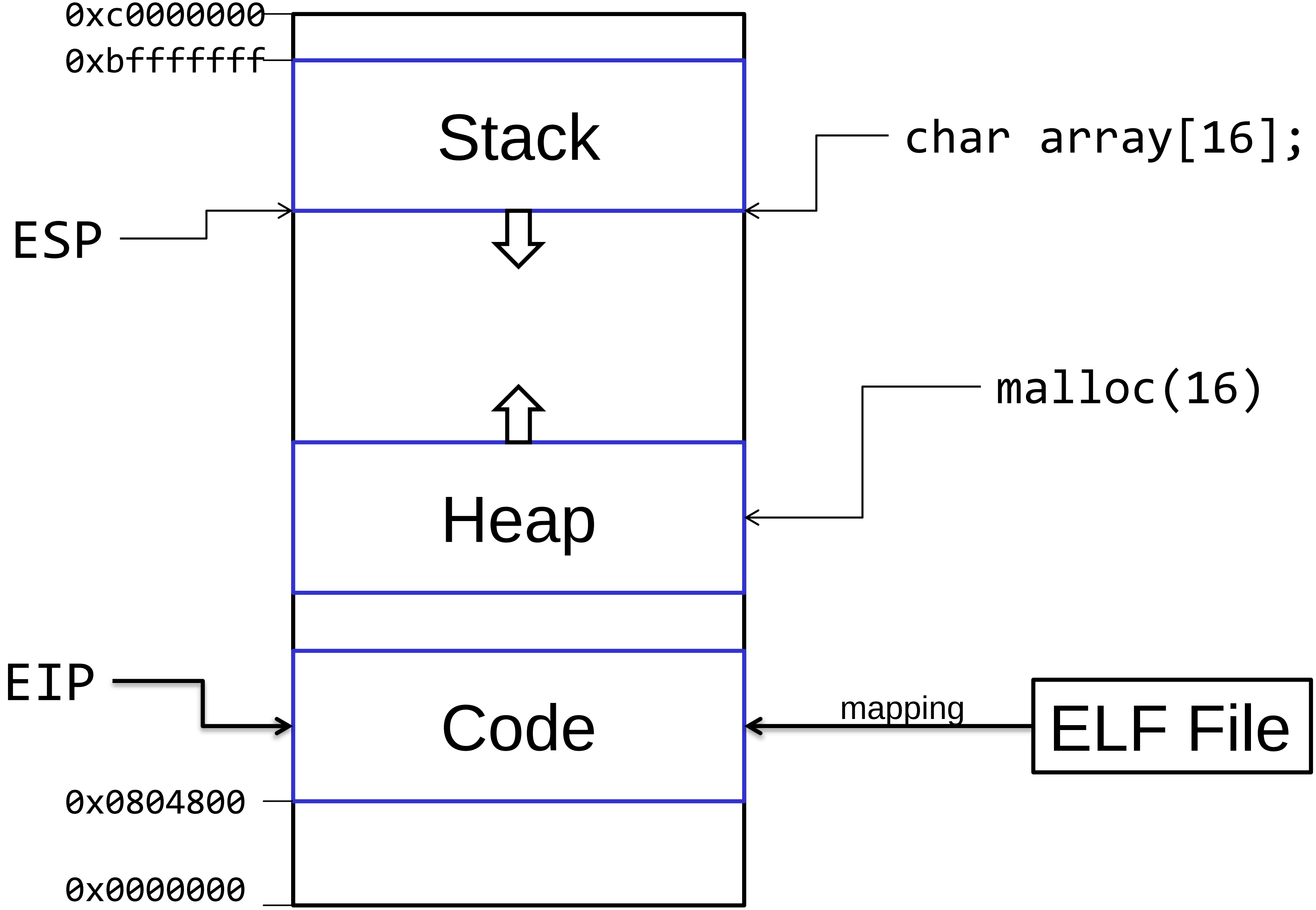
x32 Memory Layout



x32 Memory Layout



x32 Memory Layout



x32 Memory Layout

Memory regions:

Stack

- There’s one contiguous memory region containing the stack for the process
- LIFO – Last in, First Out
- Contains function **local variables**
- Also contains: **Saved Instruction Pointer (SIP)**
- Current function adds data to the top (bottom) of the stack

Heap

- There’s one contiguous memory region containing the heap
- Memory allocator returns specific pieces of the memory region
- For **malloc()**
- Also contains: heap management data

Code

- Compiled program code

ELF Format

How do programs on disk look like

ELF Format

Programs (e.g. Firefox) are stored in ELF files

ELF: Executable and Linkable Format

- Previously: “a.out” (Linux 1.2)
- Like COFF, PE (EXE), COM, …

ELF types:

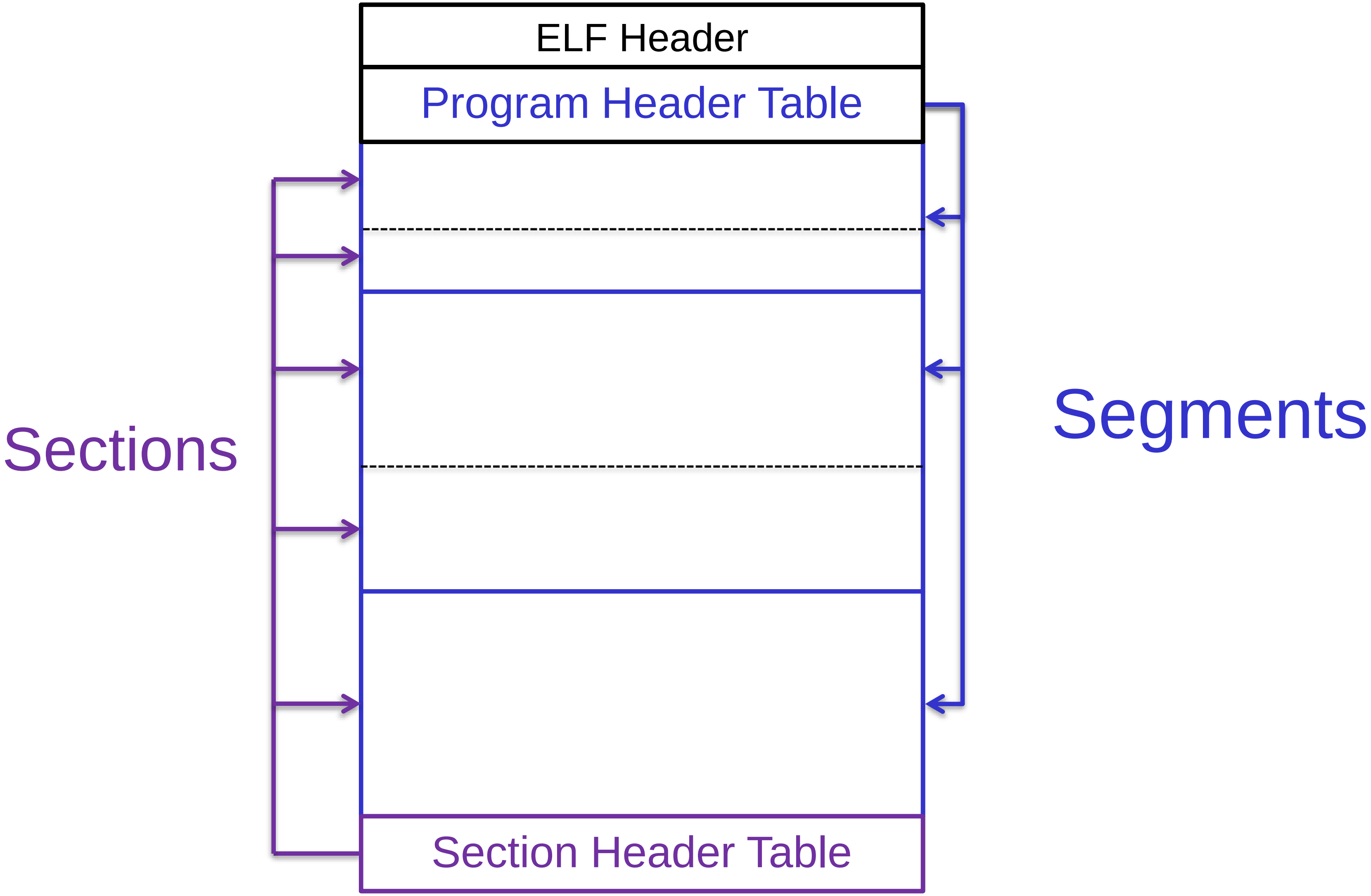
- ET_EXEC: Executable File
- ET_REL: Relocatable File
- ET_DYN: Shared Object File

ELF “views”:

- Sections
- Segments

```
$ readelf –l <binary>
```

ELF Format - Sections and Segments



ELF Format - Segment View

```
     Program Headers:
       Type           Offset             VirtAddr           PhysAddr
                      FileSiz            MemSiz              Flags  Align
       PHDR           0x00000040         0x0000400040       0x0000000000400040
                      0x000001c0         0x00000001c0        R E    8
       INTERP         0x00000200         0x0000400200       0x0000000000400200
                      0x0000001c         0x000000001c        R      1
02     LOAD           0x00000000         0x0000400000       0x0000000000400000
                      0x00000b24         0x0000000b24        R E    200000
03     LOAD           0x00000b28         0x0000600b28       0x0000000000600b28
                      0x00000270         0x0000000278        RW     200000
       DYNAMIC        0x00000b40         0x0000600b40       0x0000000000600b40
                      0x000001e0         0x00000001e0        RW     8
       NOTE           0x0000021c         0x000040021c       0x000000000040021c
                      0x00000044         0x0000000044        R      4
       GNU_EH_FRAME   0x000009ac         0x00004009ac       0x00000000004009ac
                      0x00000044         0x0000000044        R      4
07     GNU_STACK      0x00000000         0x0000000000       0x0000000000000000
                      0x00000000         0x0000000000        RW     10
```

ELF Format

```
$ readelf -l challenge0

Section to Segment mapping:
  Segment Sections...
   00
   01      .interp
   02      .interp .note.ABI-tag .note.gnu.build-id .gnu.hash
           .dynsym .dynstr .gnu.version .gnu.version_r
           .rela.dyn .rela.plt .init .plt .text .fini .rodata
           .eh_frame_hdr .eh_frame
   03      .init_array .fini_array .jcr .dynamic .got .got.plt
           .data .bss
   04      .dynamic
   05      .note.ABI-tag .note.gnu.build-id
   06      .eh_frame_hdr
   07
```

ELF Format

Sections:

- .text: Executable instructions
- .bss: Unitialized data (usually the heap)
- .data: initialized data
- .rodata: Read-Only data
- .got: Global Offset Table
- .plt: Procedure Linkage Table
- .init/.fini: Initialization instructions (“glibc”)

ELF Format

```
Program Headers:
     Type            Offset                  PhysAddr
                     FileSiz                 Flags  Align
(02) LOAD            0x0000000000000000      0x0000000000400000
                     0x0000000000000b24      R E    200000
(03) LOAD            0x0000000000000b28      0x0000000000600b28
                     0x0000000000000270      RW     200000
(07)  GNU_STACK      0x0000000000000000      0x0000000000000000
                     0x0000000000000000      RW     10
```

```
   02      .init .plt .text .fini .rodata
   03      .got .got.plt .data .bss
   07
```

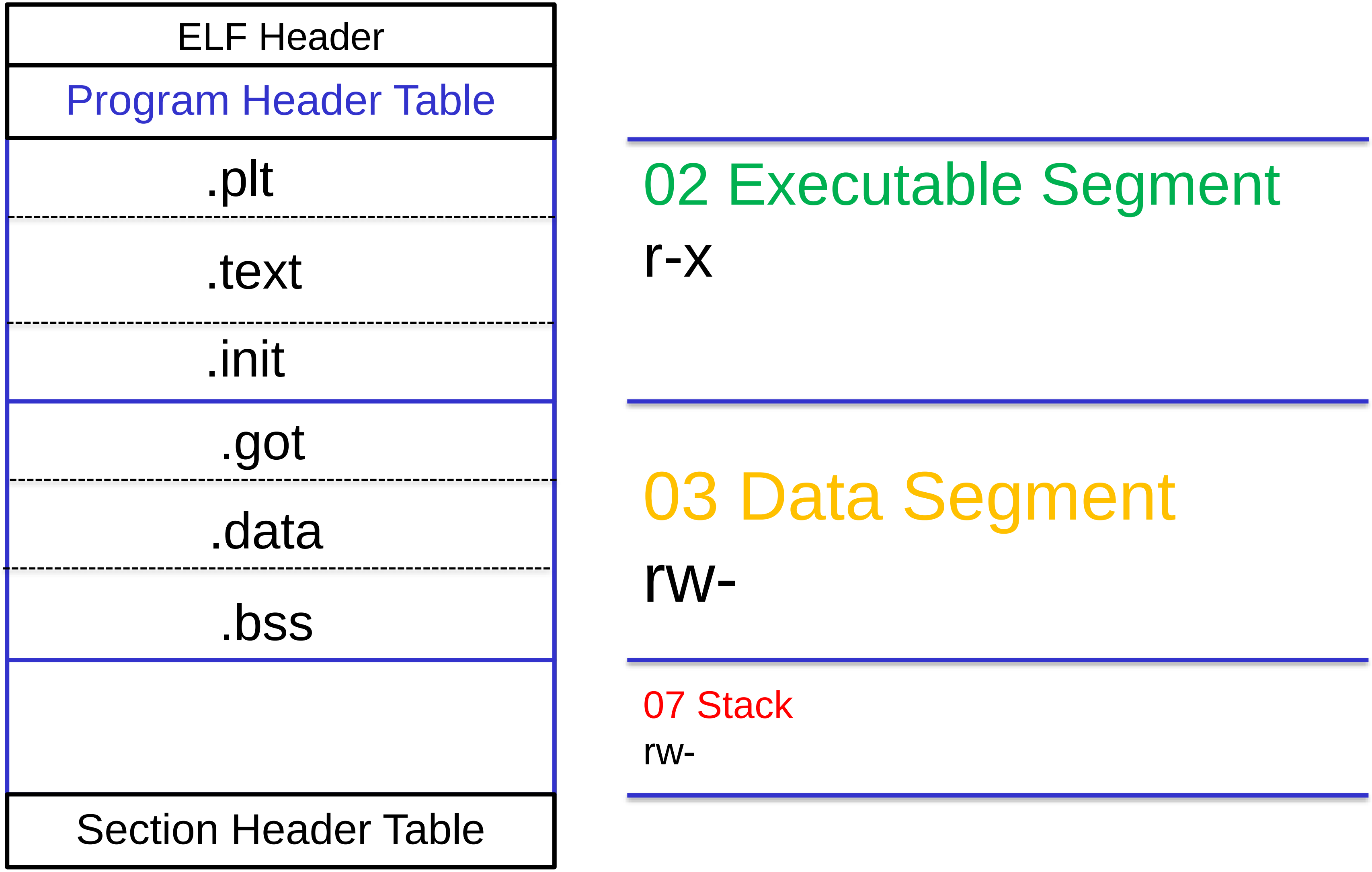
Executable Code R/E

Heap Data R/W

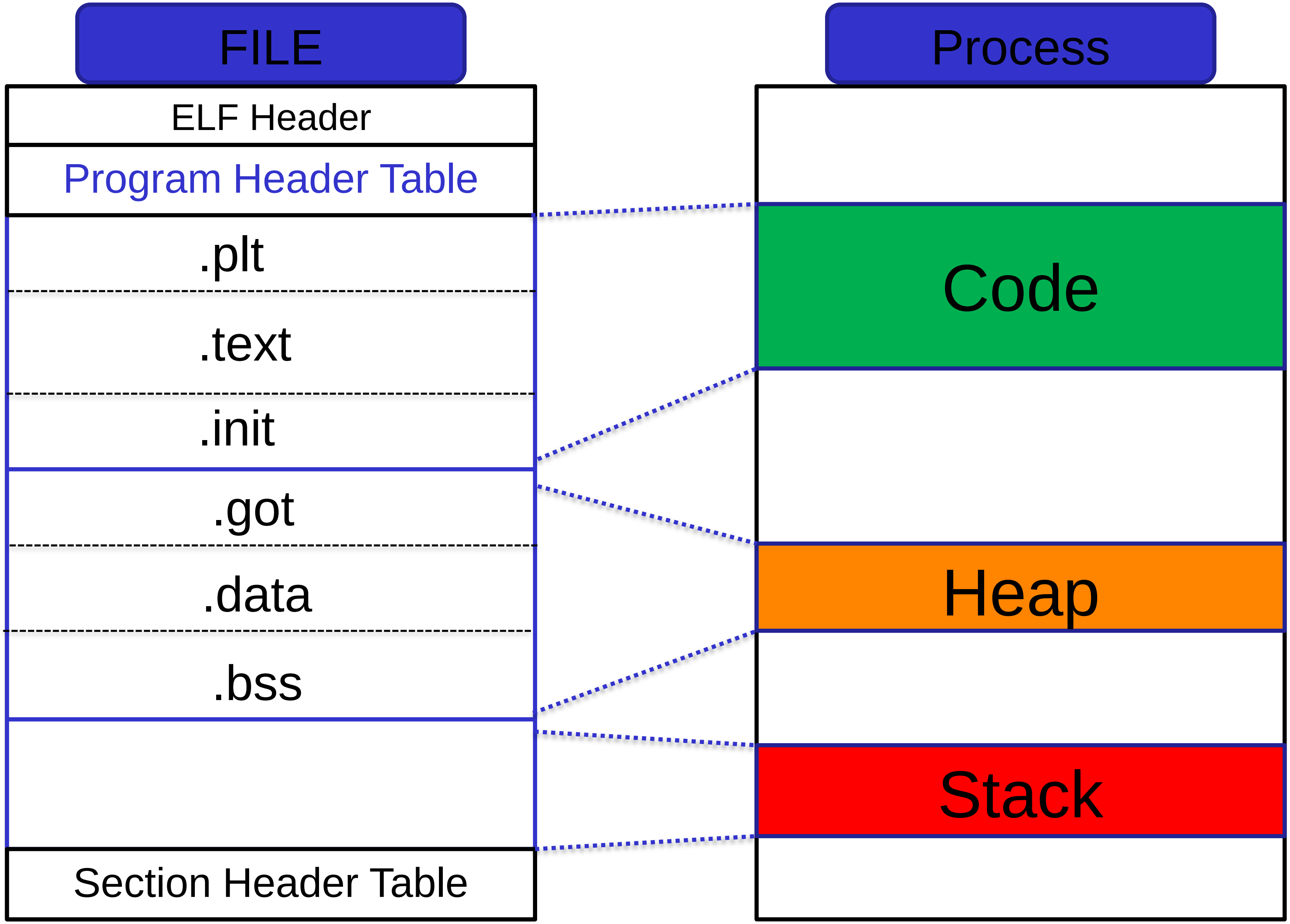
Stack Data R/W

ELF Loader

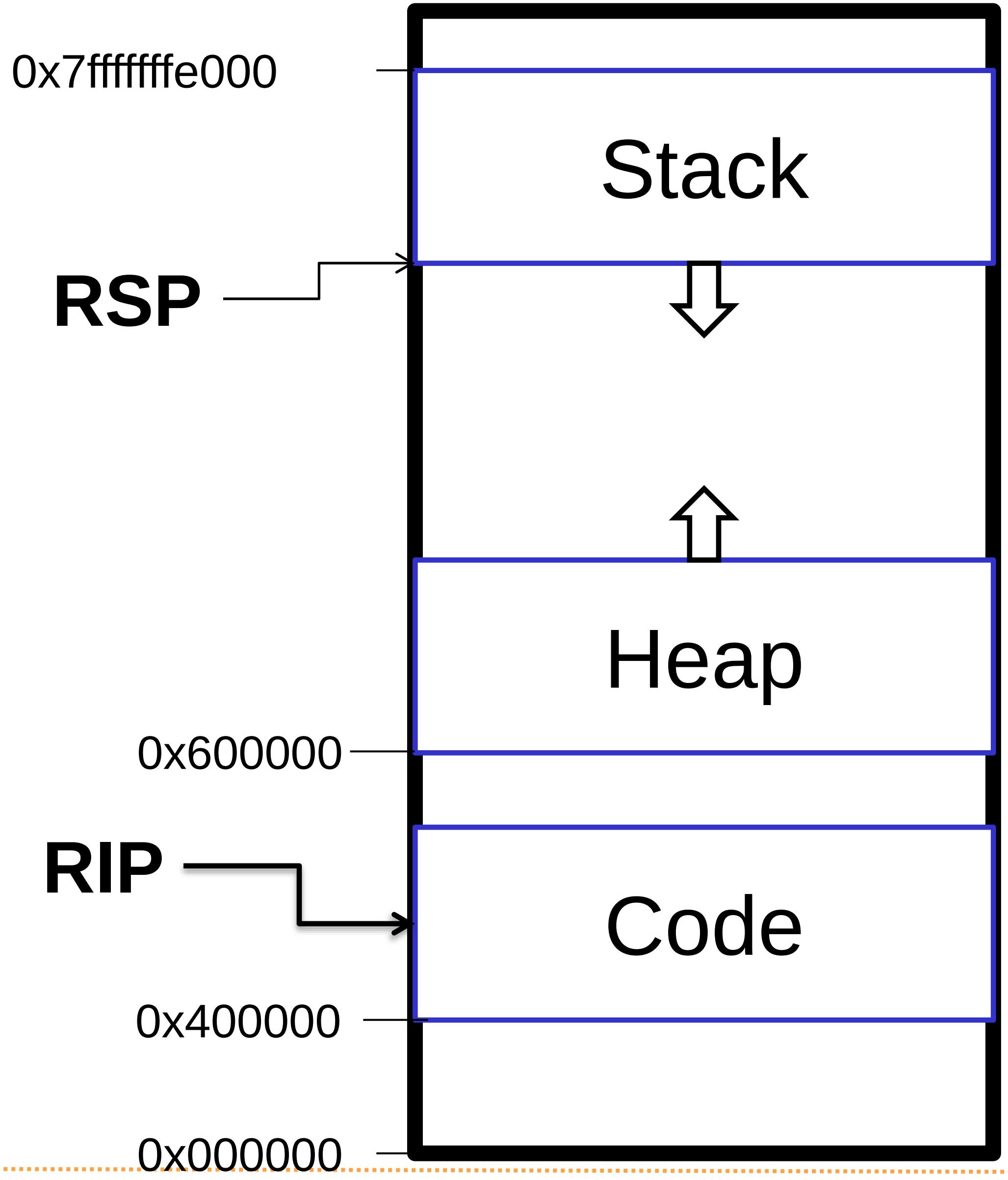
ELF Format


ELF Header
Program Header Table
.plt
.text
.init
.got
.data
.bss
Section Header Table
02 Executable Segment
r-x
03 Data Segment
rw-
07 Stack
rw-

ELF Format


FILE
ELF Header
Program Header Table
.plt
.text
.init
.got
.data
.bss
Section Header Table
Process
Code
Heap
Stack

x64 Memory Layout



Stack, Heap, Code from ELF File By Example

some static and dynamic binary analysis

ELF Format - Example C code

```c
char *globalVar = "Global";

void main(void) {
        char stackVar[16];
        char *heapVar = (char *) malloc(4);

        printf("Global var: %p\n", globalVar);
        printf("Heap var  : %p\n", heapVar);
        printf("Stack var : %p\n", stackVar);
}
```

ELF Format - ELF Analysis

```
Global var: 0x400654
Heap var   : 0x601010
Stack var  : 0x7fffffffe990



(2)   LOAD                  0x0000000000400000
                            R E       200000
(3)   LOAD                  0x0000000000600b28
                            RW        200000
(7)   GNU_STACK             0x0000000000000000
                            RW        10
```

ELF Format

See it at runtime

```
# cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 391694     /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 391694     /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 391694     /bin/cat
…
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0 [stack]
```

ELF Format

Show Code section, and disassemble:

```
$ objdump -d ./challenge1
./challenge1:     file format elf64-x86-64
Disassembly of section .init:
0000000000400588 <_init>:
…
000000000040077f <handleData>:
  40077f:   55                push   %rbp
  400780:   48 89 e5          mov    %rsp,%rbp
  400783:   48 83 ec 30       sub    $0x30,%rsp
  400787:   48 89 7d d8       mov    %rdi,-0x28(%rbp)
  40078b:   48 89 75 d0       mov    %rsi,-0x30(%rbp)
```

ELF Format

The process of creating a process from an ELF file is called:

- "Linking and Loading"

Sections:

- Are for compiler (gcc), to link several object files together (.o)

Segments:

- Are for the loader, to create the process
- Each segment consists of one ore more sections

ELF Format

Recap:

- Program Code is stored in ELF Files
- ELF Files contain segments
- Segments are copied 1:1 in the memory to create a process (of that program)
- A process has generally three important segments:
  - Code segment (the actual compiled code)
  - Heap (global allocations with malloc())
  - Stack (local variables of functions)

Challenges

Challenges:

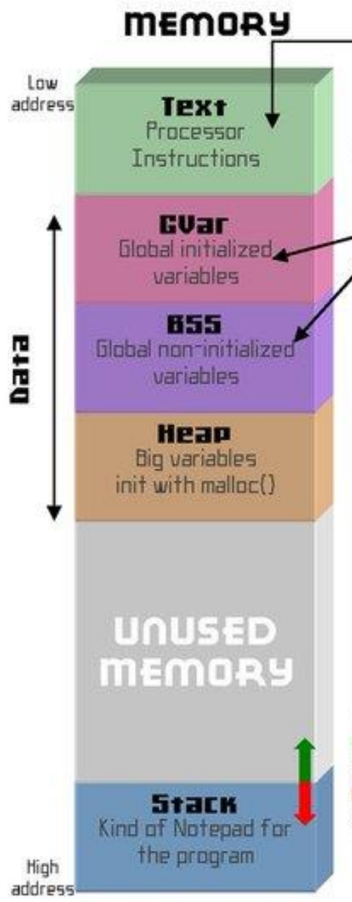
https://exploit.courses

- Challenge 0: Introduction to memory layout – basic
- Challenge 1: Introduction to memory layout - advanced
- (Challenge 4: Introduction to hex numbers, code and GDB)

Memory Segmentation Cheat Sheet

0x0ff.info production, inspired by bases-hacking.org

MEMORY



Low address
Text
Processor Instructions
GVar
Global initialized variables
BSS
Global non-initialized variables
Heap
Big variables init with malloc()
Data
UNUSED MEMORY
Stack
Kind of Notepad for the program
High address

```
//exemple.c

int global1 = 1;
char global2;

void func(int nb1,int nb2,char str)
{
   char intern;
   char buffer[10];
}

int main() {
   int nb; //in the stack
   nb = 24;

   func(nb, global1, global2);

   return 0;
}
```

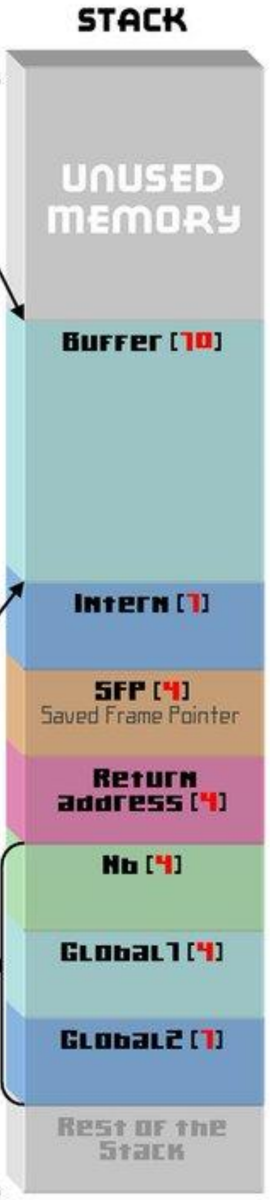
PUSHING
Insert data
POPPING
Remove data

EBP/FP/LBP
Pointer on various elements in the stack frame

SP/ESP/RSP End Stack Pointer
Last element address

STACK

Low address



High address

GLOBAL REGISTERS

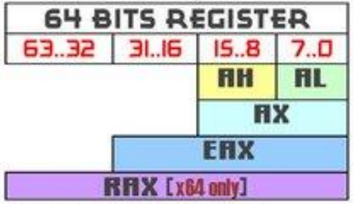
Used for general purpose

X86

AL/AH/AX/EAX/RAX
BL/BH/BX/EBX/RBX
CL/CH/CX/ECX/RCX
DL/DH/DX/EDX/RDX

ARM

r0-r12



INDEX POINTERS

Use for Strings operations

X86

SI/ESI/RSI : Source index
DI/EDI/RDI : Destination index

INSTRUCTION POINTER

The current instruction address.

X86

IP/EIP/RIP

ARM

PC

SEGMENT REGISTERS

Use to easily reade/write to memory

X86

CS : Code	ES : Extra data #1
DS : Data	FS : Extra data #2
SS : Stack	GS : Extra data #3

MEMORY ALIGNMENT

Data must be aligned on 4,8,16... Bytes, depending on your system.

EXAMPLE
There is an 3 bytes long empty gap between intern and SFP.

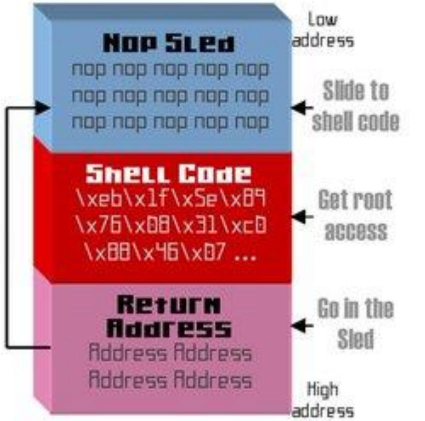
BUFFER OVERFLOW

when input is longer than the allocated memory space.

[Stack based]
Smart overwrite of return address

EXAMPLE
Put a 22 bytes long string and overwrite intern Return address.

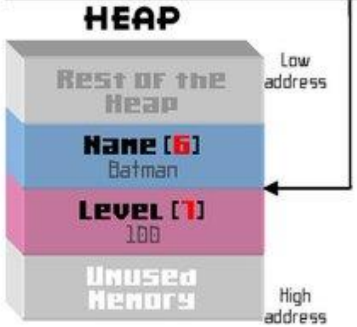
EXPLOIT ANATOMY



[Heap based]
Smart overwrite of others variables like file name.

EXAMPLE
You have to enter a 6 letters name in the character builder of a game. You enter Batman\x64, to overwrite level variable and get max stats !

HEAP



HOW TO WEAKEN A PROGRAM Training

LINUX

Disable ASLR (random memory address) :

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

Disable non-executable stack :

```
$ gcc -z execstack ...
```

Disable stack protector :

```
$ gcc -fno-stack-protector ...
```

Force 32-bits compilation mode :

```
$ gcc -m32 ...
```