

# Introduction to ROP

chiliz

# whoami



- Lisa / chiliz
- Student in Automation and Mechatronics (HFU Tuttlingen, Germany)
- Bachelor Thesis at Bosch (Automated Security Testing & Fuzzing)
- CTF Player
- Blackhoodie Attendee in Luxembourg and at Troopers

# Agenda

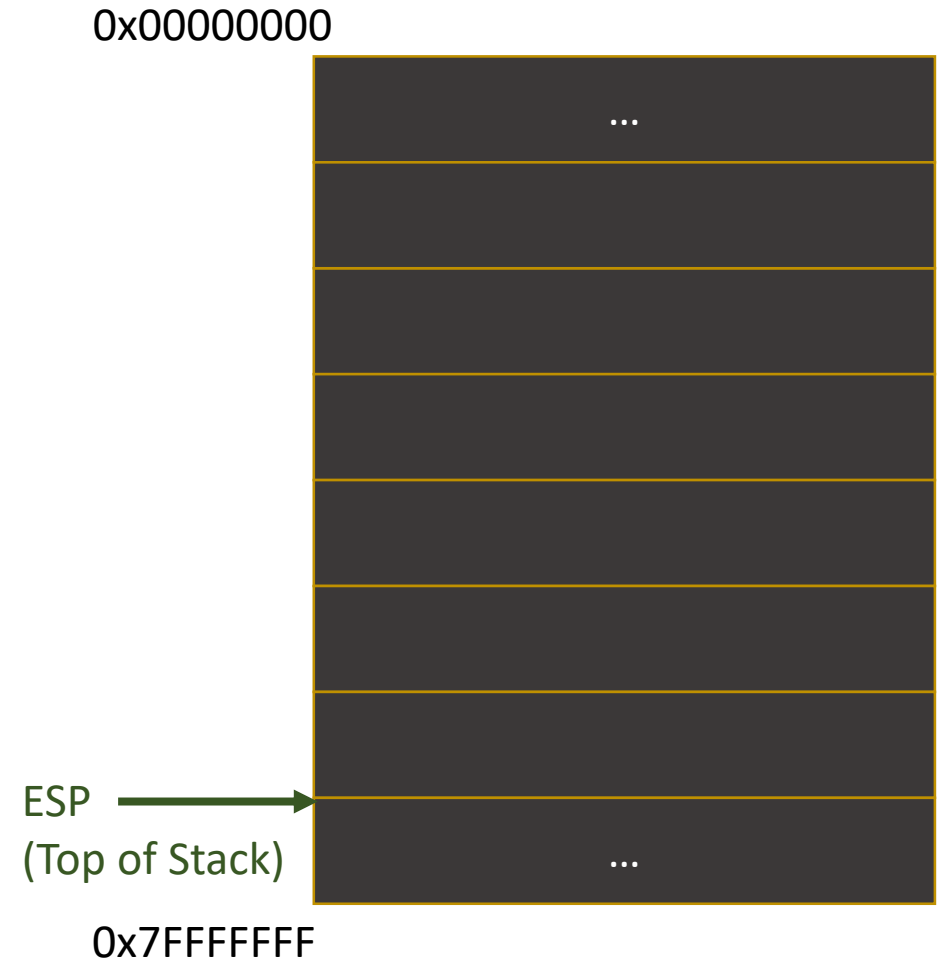
- Recap Buffer Overflow
- What is ROP and why do we want it?
- Demo & Exercise 32 Bit – Ret2libc
- Demo & Exercise 64 Bit – simple ROP-chain
- Demo & Exercise ASLR – Address leak & ROP-chain

# Recap Buffer Overflow

```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
main:
    ;eax holds pointer
    ;to argv[1]
    push eax
    call vuln(char*)
    ...
```



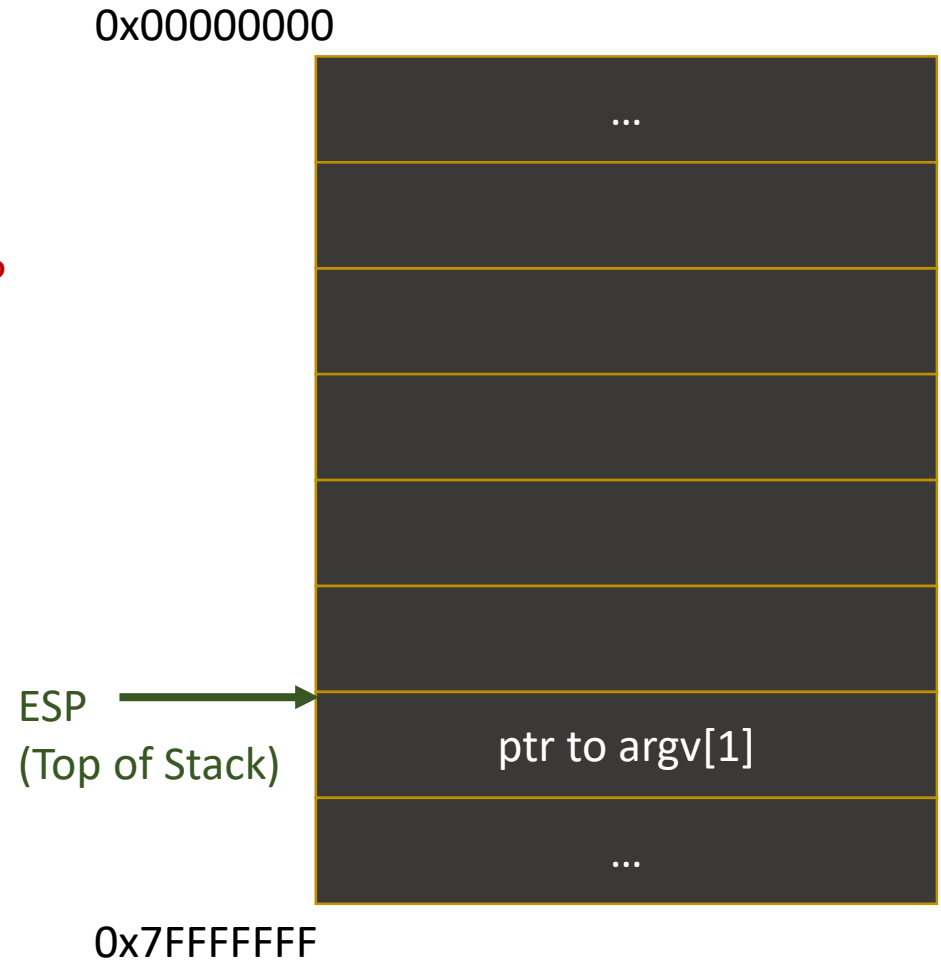
# Recap Buffer Overflow

```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
main:
    ;eax holds pointer
    ;to argv[1]
    push eax
    call vuln(char*)
    ...
```

← EIP



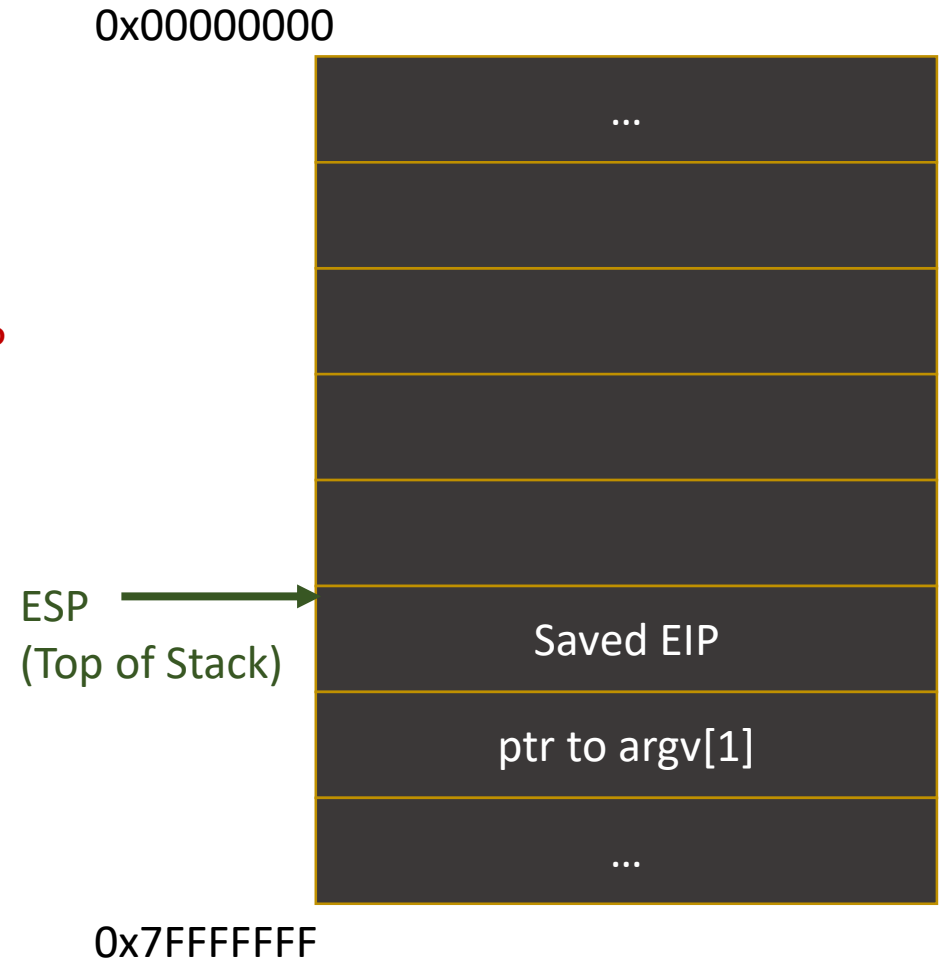
# Recap Buffer Overflow

```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
main:
    ;eax holds pointer
    ;to argv[1]
    push eax
    call vuln(char*)
    ...
```

← EIP



# Recap Buffer Overflow

```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}

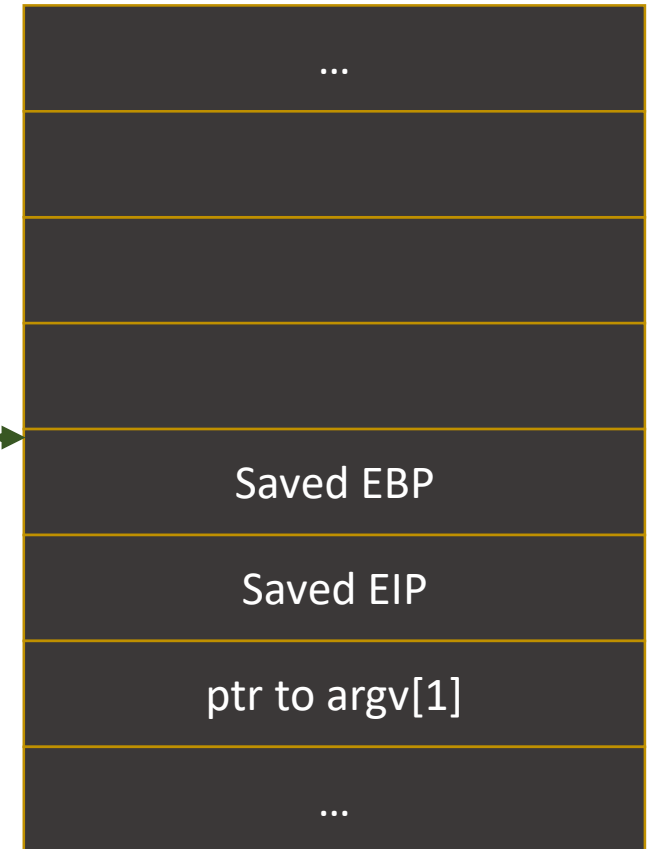
int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push ebp
    mov ebp, esp
    sub esp, 32
    push[ebp+8]; input
    lea eax, [ebp-32]
    push eax; buffer
    call strcpy
    add esp, 8
    leave
    ret
```

← EIP

ESP →  
(Top of Stack)

0x00000000



0x7FFFFFFF

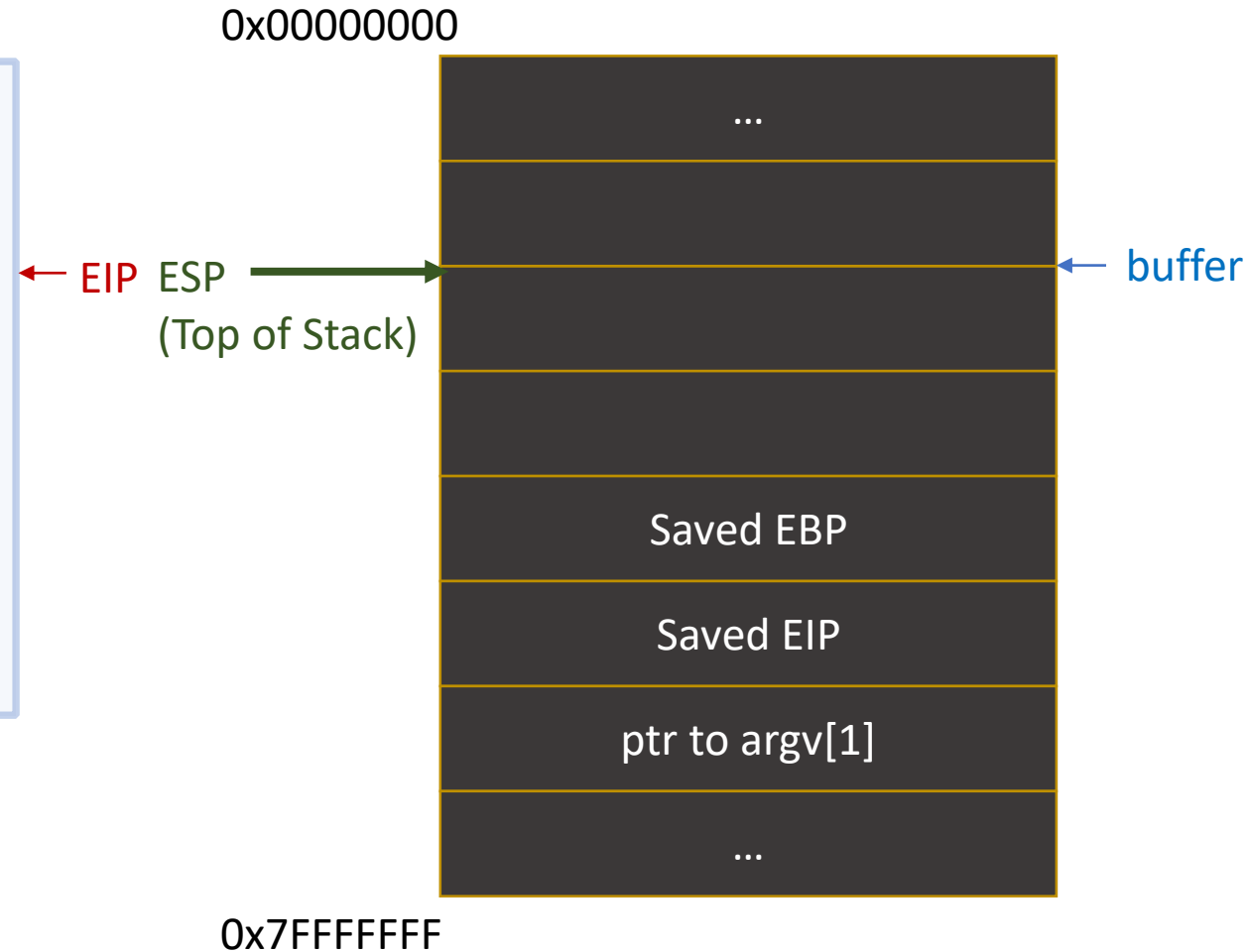


# Recap Buffer Overflow

```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push ebp
    mov ebp, esp
    sub esp, 32
    push[ebp+8]; input
    lea eax, [ebp-32]
    push eax; buffer
    call strcpy
    add esp, 8
    leave
    ret
```



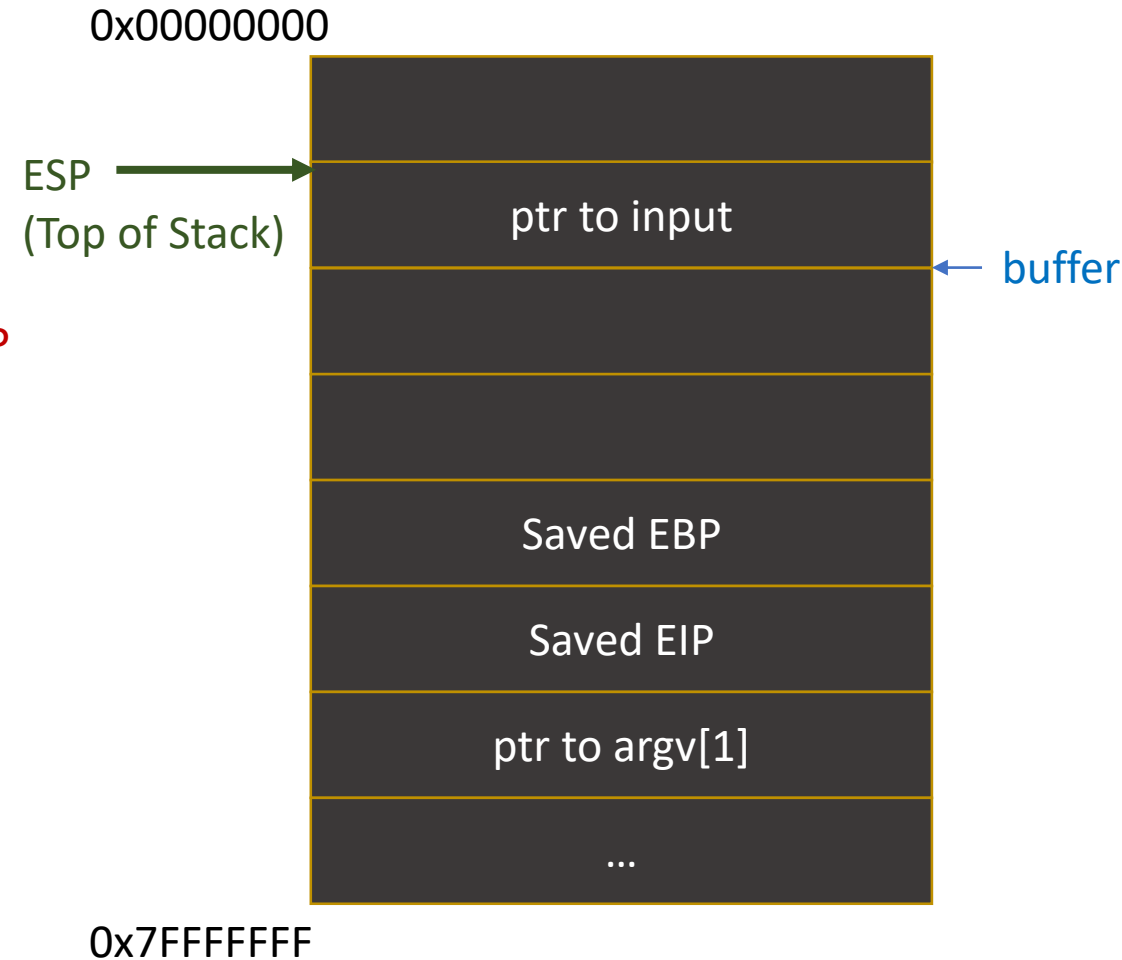


# Recap Buffer Overflow

```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push ebp
    mov ebp, esp
    sub esp, 32
    push[ebp+8]; input
    lea eax, [ebp-32]
    push eax; buffer
    call strcpy
    add esp, 8
    leave
    ret
```

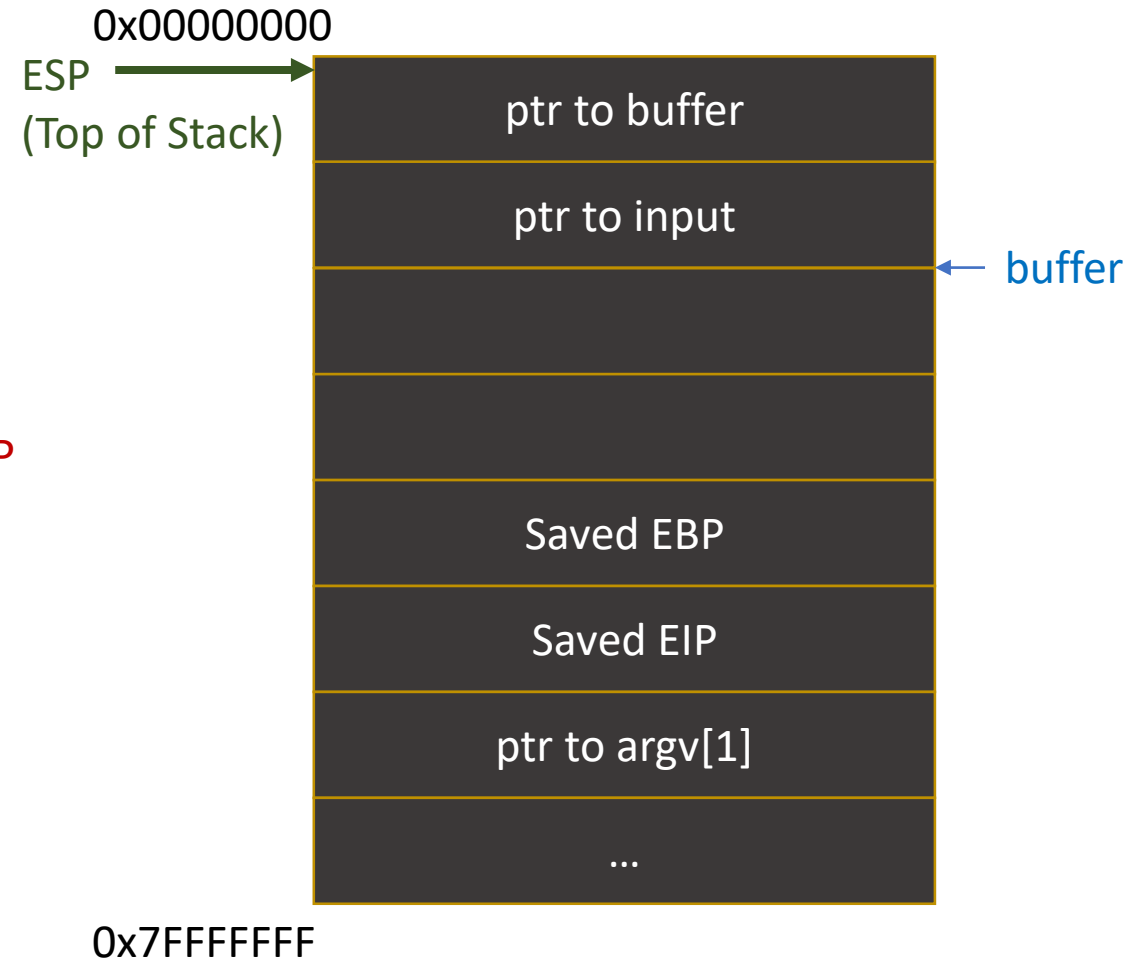


# Recap Buffer Overflow

```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push ebp
    mov ebp, esp
    sub esp, 32
    push[ebp+8]; input
    lea eax, [ebp-32]
    push eax; buffer
    call strcpy
    add esp, 8
    leave
    ret
```

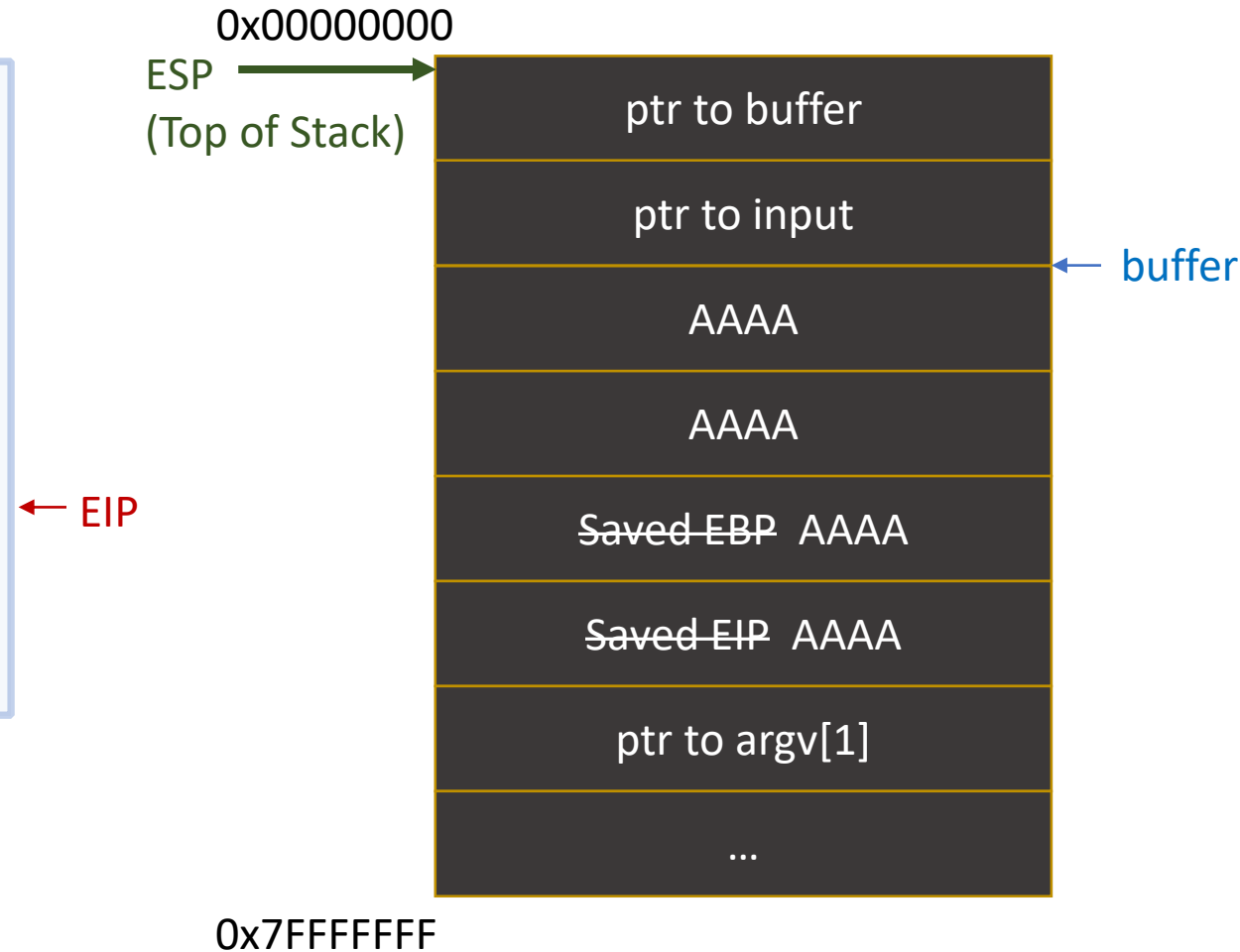


# Recap Buffer Overflow

```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push ebp
    mov ebp, esp
    sub esp, 32
    push[ebp+8]; input
    lea eax, [ebp-32]
    push eax; buffer
    call strcpy
    add esp, 8
    leave
    ret
```

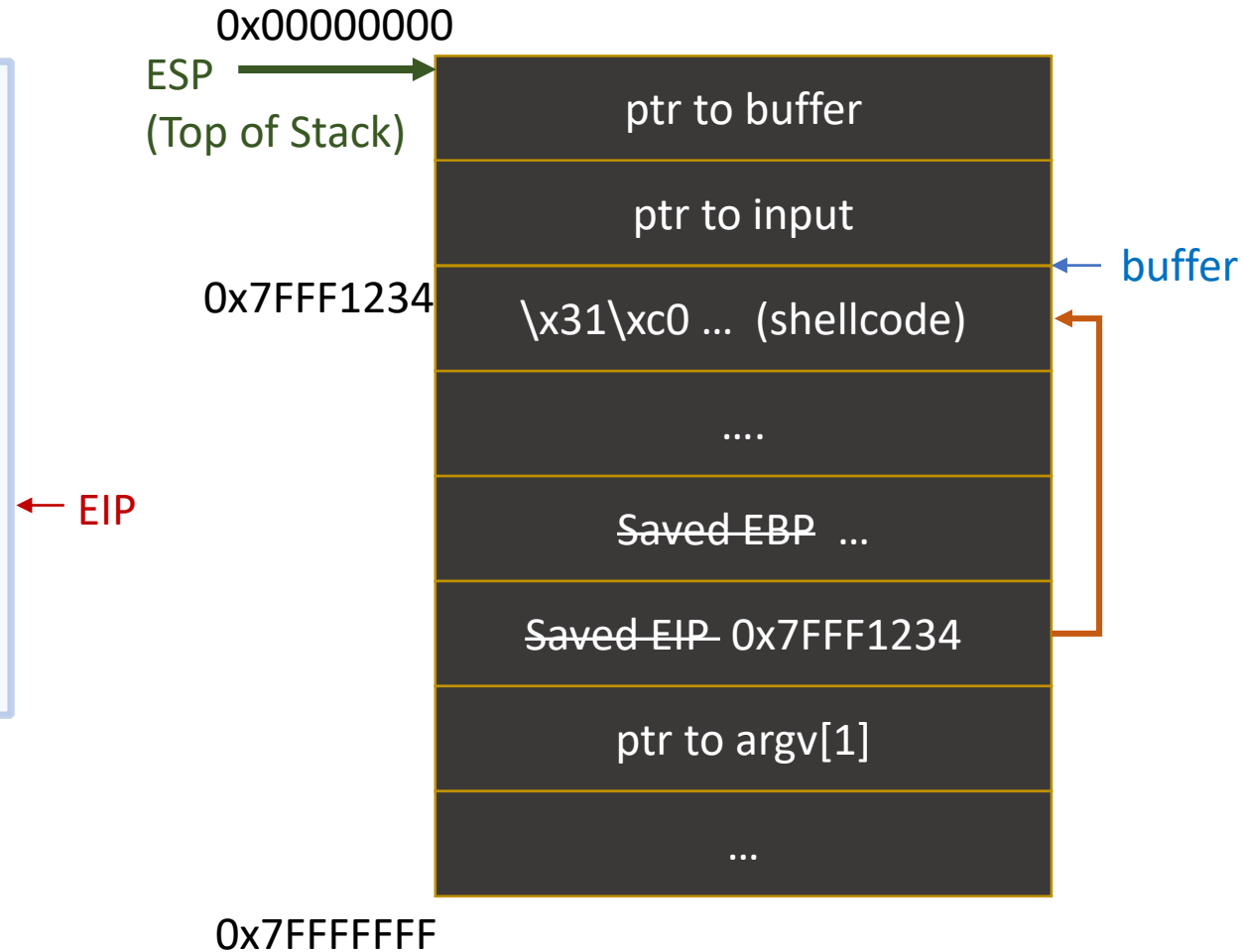


# Recap Buffer Overflow

```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}
```

```
int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push ebp
    mov ebp, esp
    sub esp, 32
    push[ebp+8]; input
    lea eax, [ebp-32]
    push eax; buffer
    call strcpy
    add esp, 8
    leave
    ret
```



# Recap Buffer Overflow

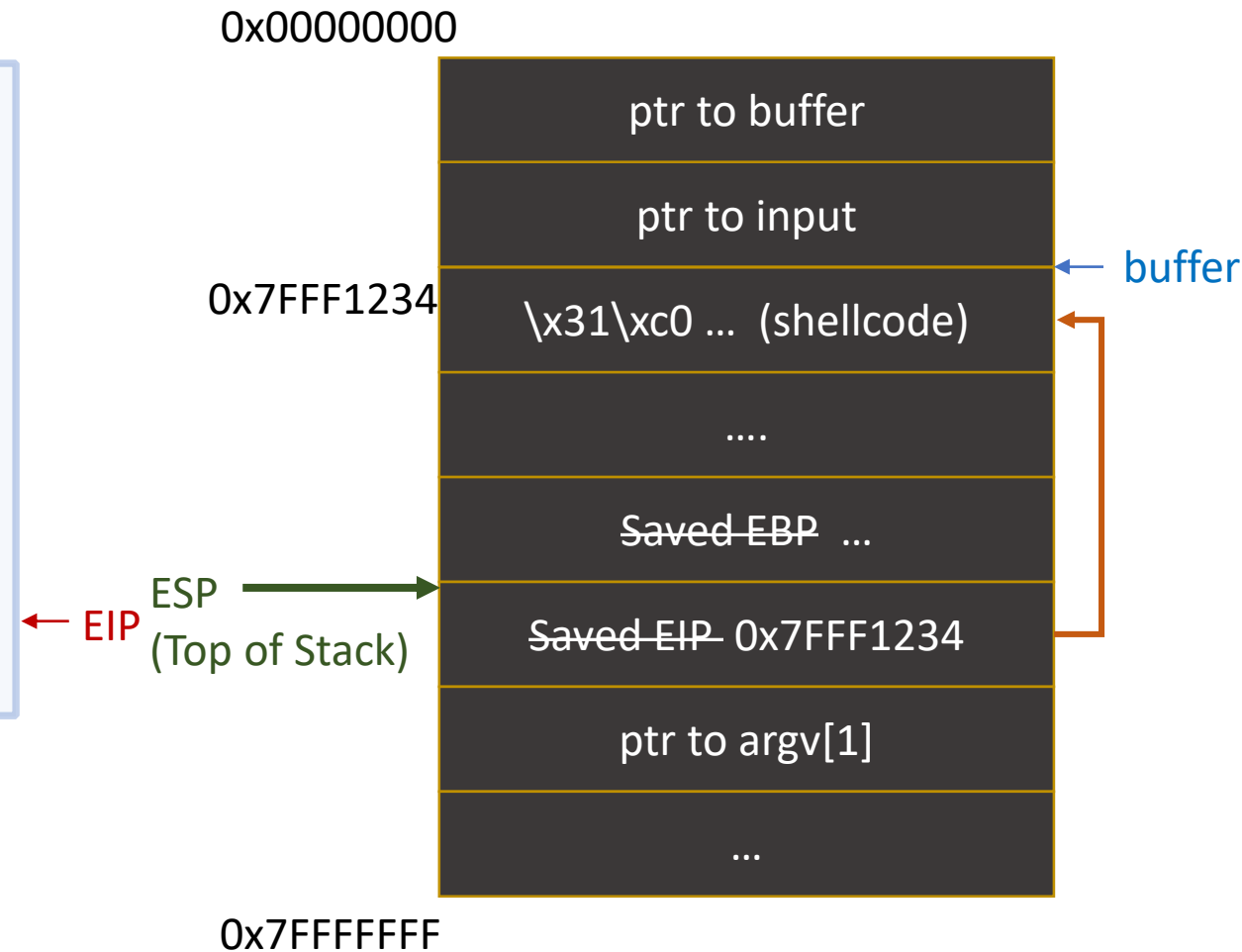
```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}
```

```
int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push ebp
    mov ebp, esp
    sub esp, 32
    push[ebp+8]; input
    lea eax, [ebp-32]
    push eax; buffer
    call strcpy
    add esp, 8
    leave
    ret
```

Leave:  
mov esp, ebp  
pop ebp

Ret:  
"pop eip"



# Recap Buffer Overflow

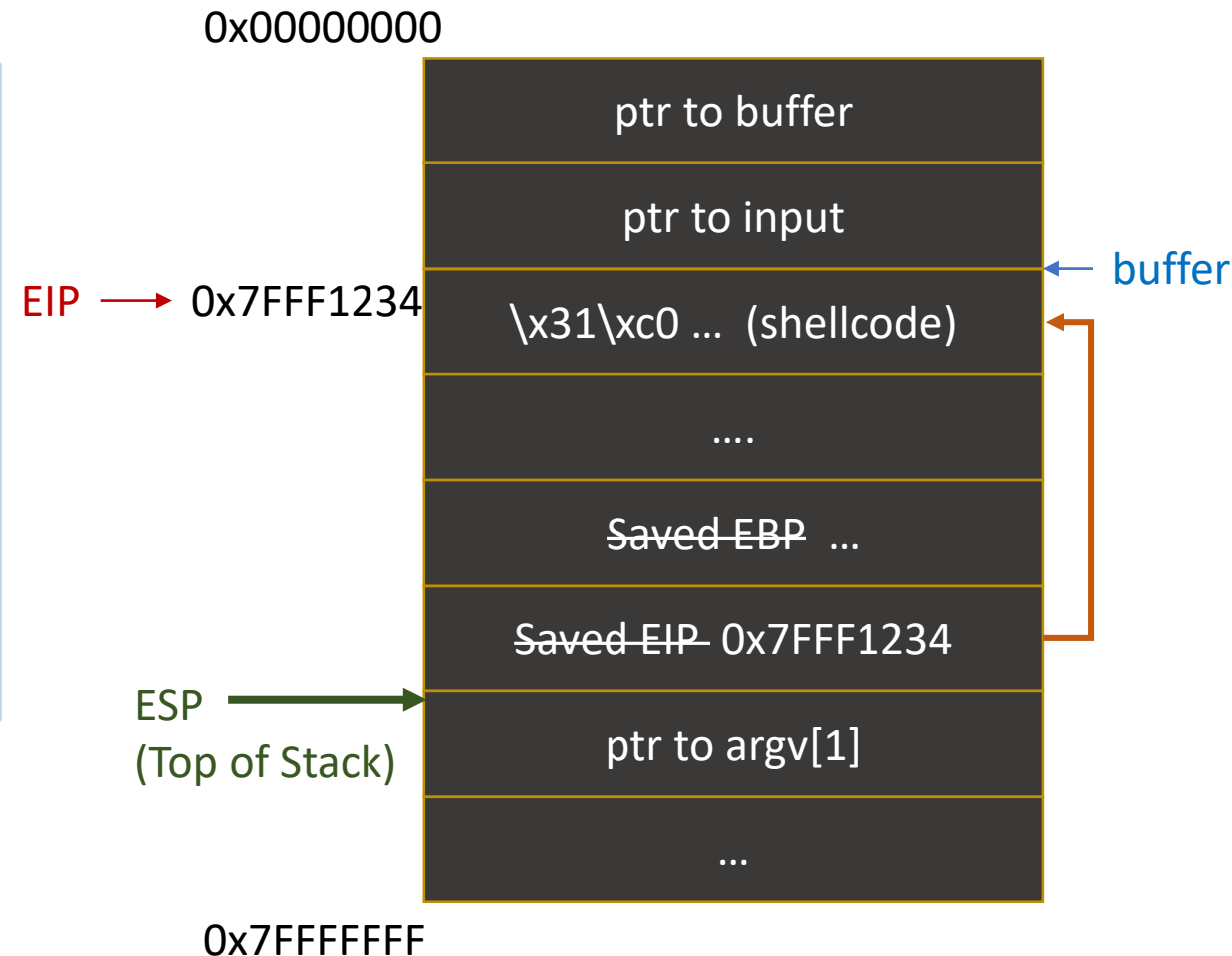
```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}
```

```
int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push ebp
    mov ebp, esp
    sub esp, 32
    push[ebp+8]; input
    lea eax, [ebp-32]
    push eax; buffer
    call strcpy
    add esp, 8
    leave
    ret
```

Leave:  
mov esp, ebp  
pop ebp

Ret:  
"pop eip"



# Return Oriented Programming (ROP)

## – Why do we want it?

- On modern systems the stack of a program is not executable anymore (security mechanism)

=> NX-Bit is set / Data Execution Prevention (DEP)

# Return Oriented Programming (ROP)

## – Why do we want it?

- On modern systems the stack of a program is not executable anymore (security mechanism)

=> NX-Bit is set / Data Execution Prevention (DEP)

- ROP is a technique to defeat this protection of a non-executable stack
- Basic Principle: Code Reuse





# Code Reuse

```
#include <stdio.h>
void win()
{
    printf("Congratulations!\n");
    execve("/bin/sh" ..);
}

int main()
{
    char buffer[20];
    printf("Enter some text:\n");
    scanf("%s", buffer);
    return 0;
}
```



# Code Reuse

```
#include <stdio.h>
void win()
{
    printf("Congratulations!\n");
    execve("/bin/sh" ..);
}

int main()
{
    char buffer[20];
    printf("Enter some text:\n");
    scanf("%s", buffer);
    return 0;
}
```

What can we do when there is no win function?



# Code Reuse

```
#include <stdio.h>
void win()
{
    printf("Congratulations!\n");
    execve("/bin/sh" ..);
}

int main()
{
    char buffer[20];
    printf("Enter some text:\n");
    scanf("%s", buffer);
    return 0;
}
```

What can we do when there is no win function?

⇒ libc (Standard C library) has always a win function: system

⇒ Goal: system("/bin/sh")



# The C standard library

- `libc`: implements C – standard functions (`printf`, `strcpy`..), and POSIX functions (`system`, wrapper for syscalls)
- Compiled as `.so` (shared object, a linux library)  
=> one of its header files is the famous `stdio.h`
- `libc.so.6` => symlink to latest libc- version (e.g. `libc-2.28.so`)
- Find it with `gdb->vmmap` or `ldd`
- Path most often `/usr/lib/libc-2.28.so`

# Ret2libc

Approach:

- Find Buffer Overflow
- Overwrite with this a stored return address with the address of a function in the libc (e.g. system)
- The libc function will be executed when the vuln function returns  
=> Ret2libc (simple and special case of ROP)

# 32 Bit Calling Convention Linux – Ret2libc

- Function arguments are pushed to the stack



# 32 Bit – Ret2libc

system("/bin/sh")

```
main:
...
; eax holds
; ptr to "/bin/sh"
push eax
call system
...
```

← EIP

0x00000000

ESP →

ptr to „/bin/sh“

0x7FFFFFFF

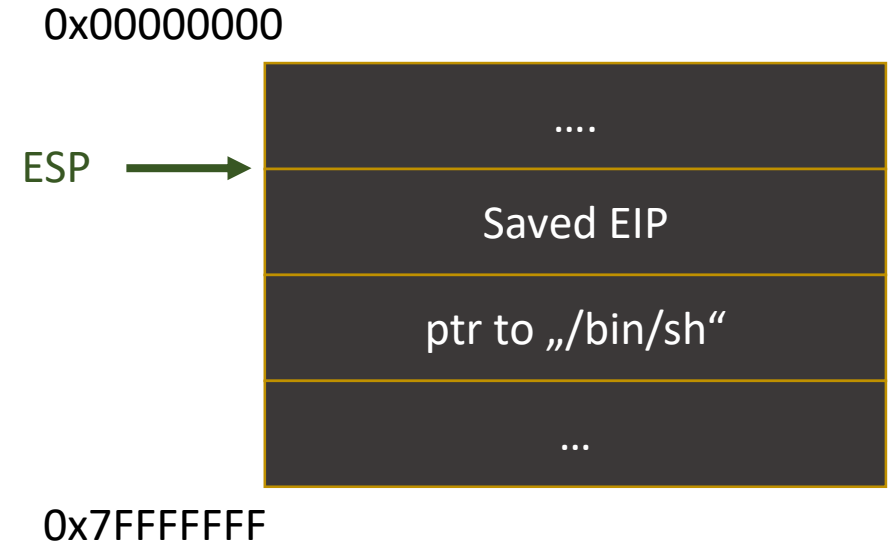


# 32 Bit – Ret2libc

`system("/bin/sh")`

```
main:
...
; eax holds
; ptr to "/bin/sh"
push eax
call system
...
```

← EIP



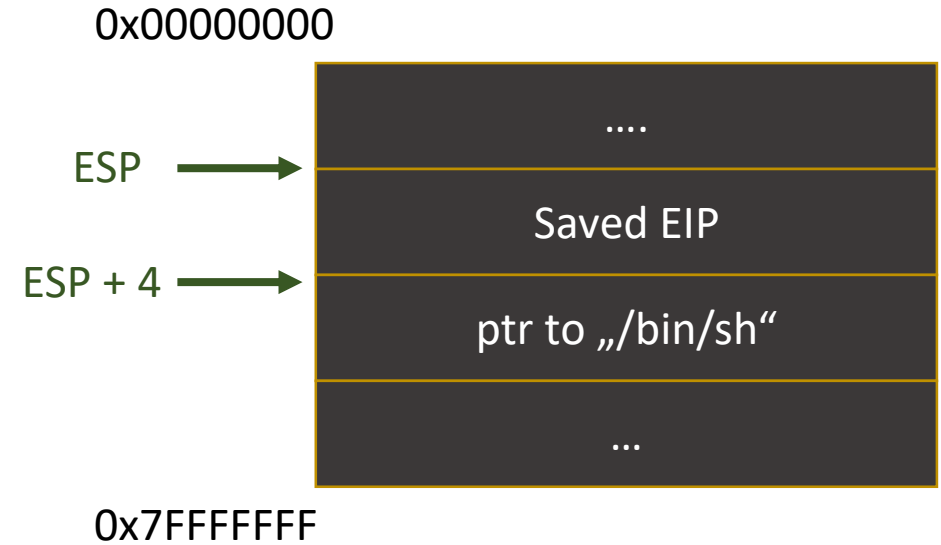


# 32 Bit – Ret2libc

`system("/bin/sh")`

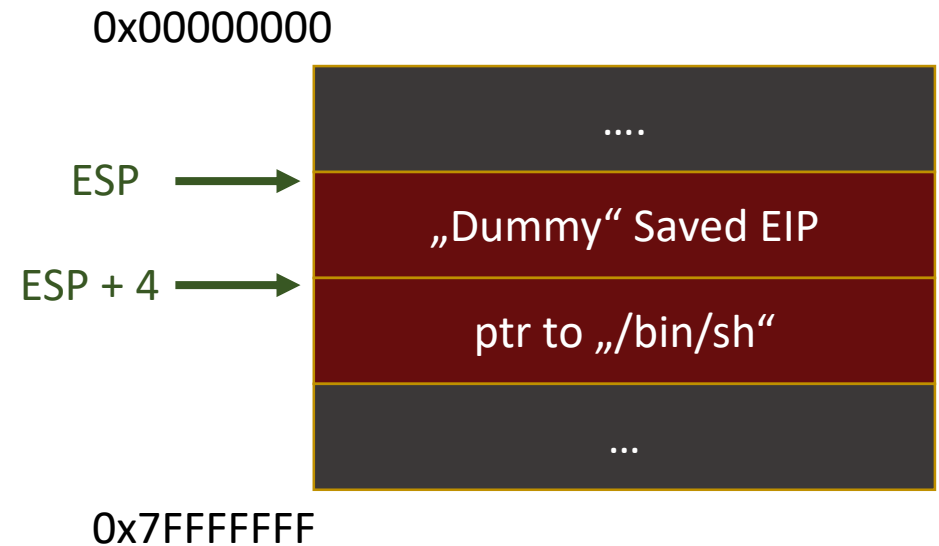
```
main:
...
; eax holds
; ptr to "/bin/sh"
push eax
call system
...
```

← EIP



# 32 Bit – Ret2libc

- We want to call:  
`system("/bin/sh")`
- system expects the argument after the saved EIP, which means `esp+4` at the time of the call.



# 32 Bit – Ret2libc

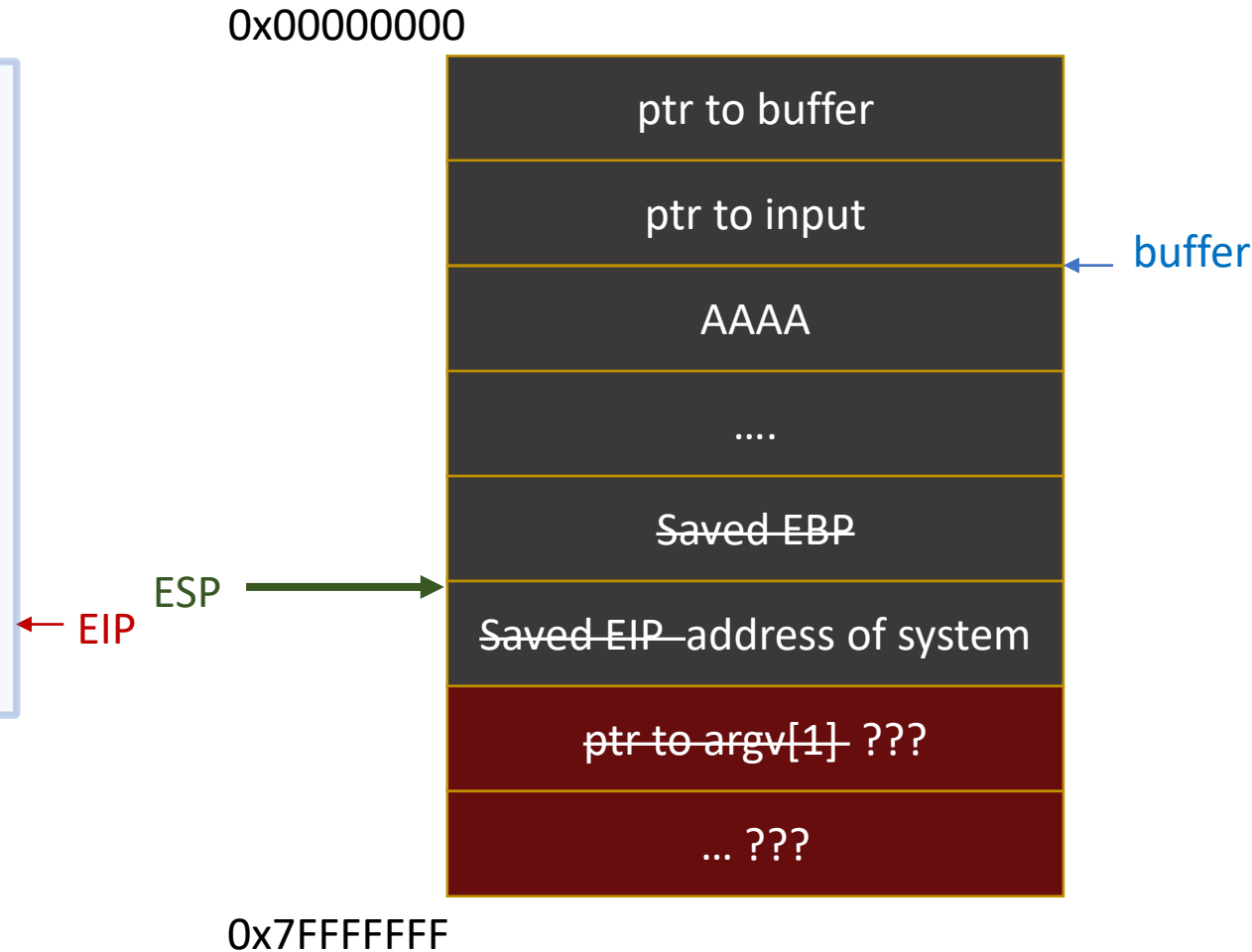
```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push ebp
    mov ebp, esp
    sub esp, 32
    push[ebp+8]; input
    lea eax, [ebp-32]
    push eax; buffer
    call strcpy
    add esp, 8
    leave
    ret
```

Leave:  
mov esp, ebp  
pop ebp

Ret:  
"pop eip"



# 32 Bit – Ret2libc

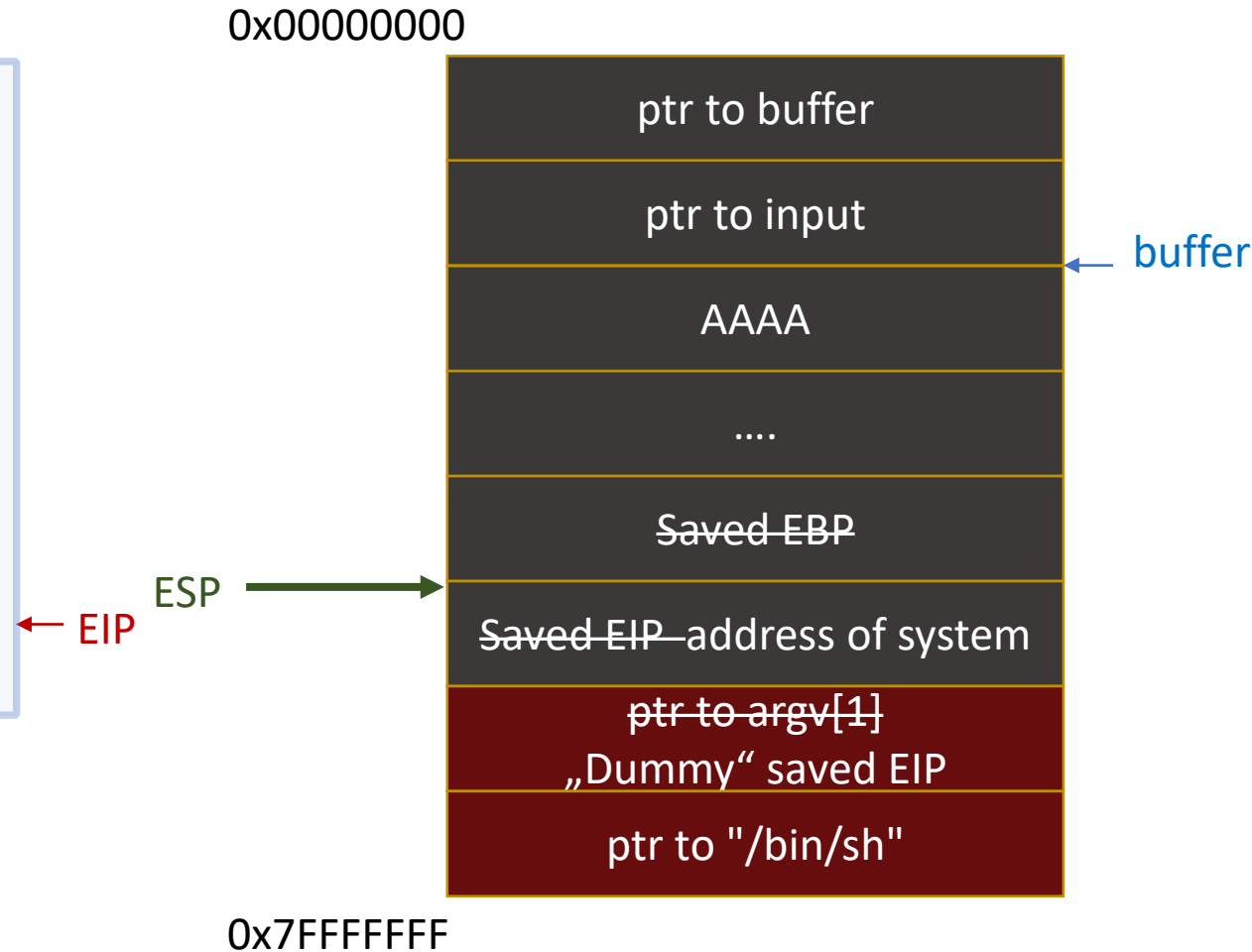
```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}
```

```
int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push ebp
    mov ebp, esp
    sub esp, 32
    push[ebp+8]; input
    lea eax, [ebp-32]
    push eax; buffer
    call strcpy
    add esp, 8
    leave
    ret
```

Leave:  
mov esp, ebp  
pop ebp

Ret:  
"pop eip"



# 32 Bit – Ret2libc

```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}

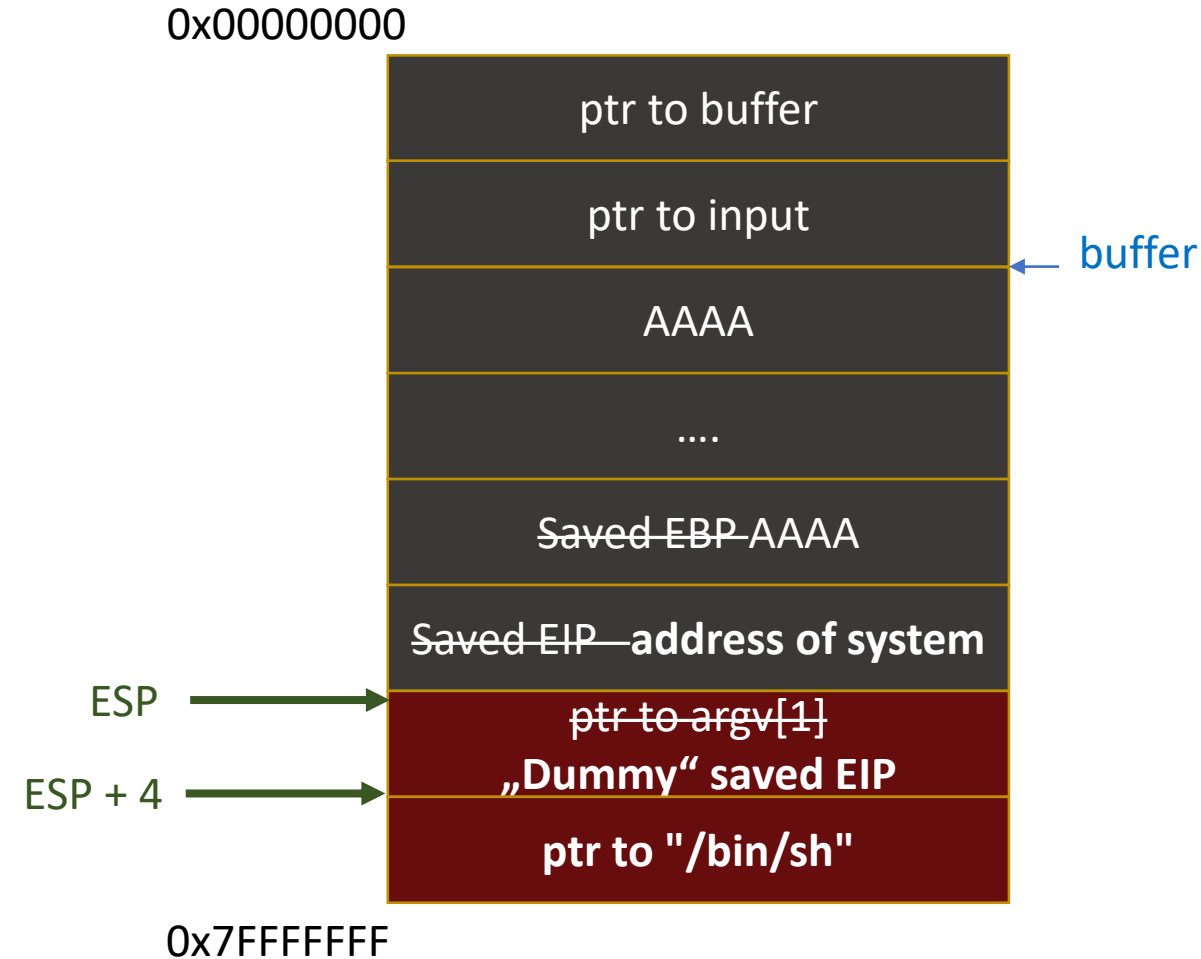
int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

system :  
...

← EIP

Leave:  
mov esp, ebp  
pop ebp

Ret:  
"pop eip"



# 32 Bit – Ret2libc

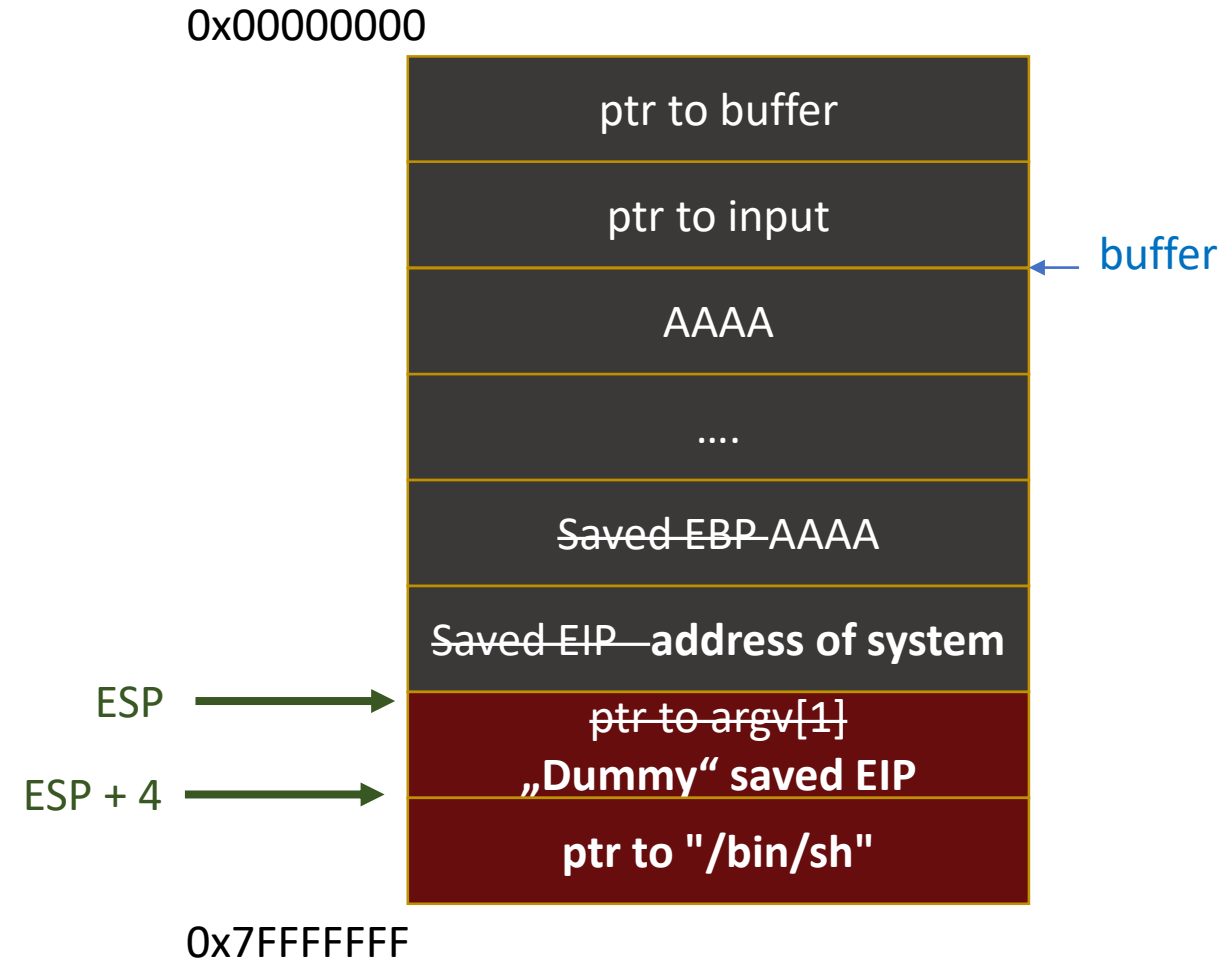
Payload = "A" \* 32

+ "AAAA" (~~saved EBP~~)

+ address of system (~~saved EIP~~)

+ "BBBB" (Dummy saved EIP)

+ address of "/bin/sh"



# libc: address of `system` and `"/bin/sh"`

0x7FFFFFFF

stack



libc

`"/bin/sh"`

`system(){ ...}`

Libc-base



heap

.text

0x00000000

$\text{Address\_System} = \text{Libc-Base} + \text{Offset to system()}$

$\text{Address\_Bin\_Sh} = \text{Libc-Base} + \text{Offset to "/bin/sh"}$



# Many roads lead to Rome ...

	Libc base	Offset system	Offset "/bin/sh"
<b>Command line</b>	<b>ldd</b> ./binary	<b>readelf -s</b> /path/to/libc   <b>grep system</b>	<b>strings -tx</b> /path/to/libc   <b>grep /bin/sh</b>
<b>gdb-peda</b>	⇒ run  ⇒ <b>vmmap</b>	⇒ run  absolute address (if ASLR is disabled): ⇒ <b>p system</b>	⇒ run  absolute address (if ASLR is disabled): ⇒ <b>searchmem /bin/sh</b>
<b>Hopper/IDA</b>		<b>search in labels</b> for system	<b>search in Strs</b> for "/bin/sh"



# 64 Bit – Calling convention Linux

- Arguments are stored in RDI, RSI, RDX, RCX, R8, R9, XMM0–7 (in this order)
- Return value of a function is stored in RAX

# 64 Bit – Calling convention

`system("/bin/sh")`

```
.binsh:  
.string "/bin/sh"  
  
main :  
    mov rdi, OFFSET.binsh  
    call system
```

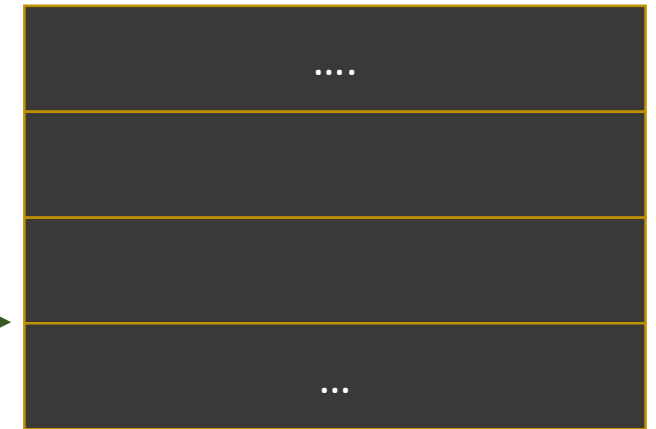
← RIP

0x0000...

RSP



0x7FFF...



RDI

ptr to „/bin/sh“

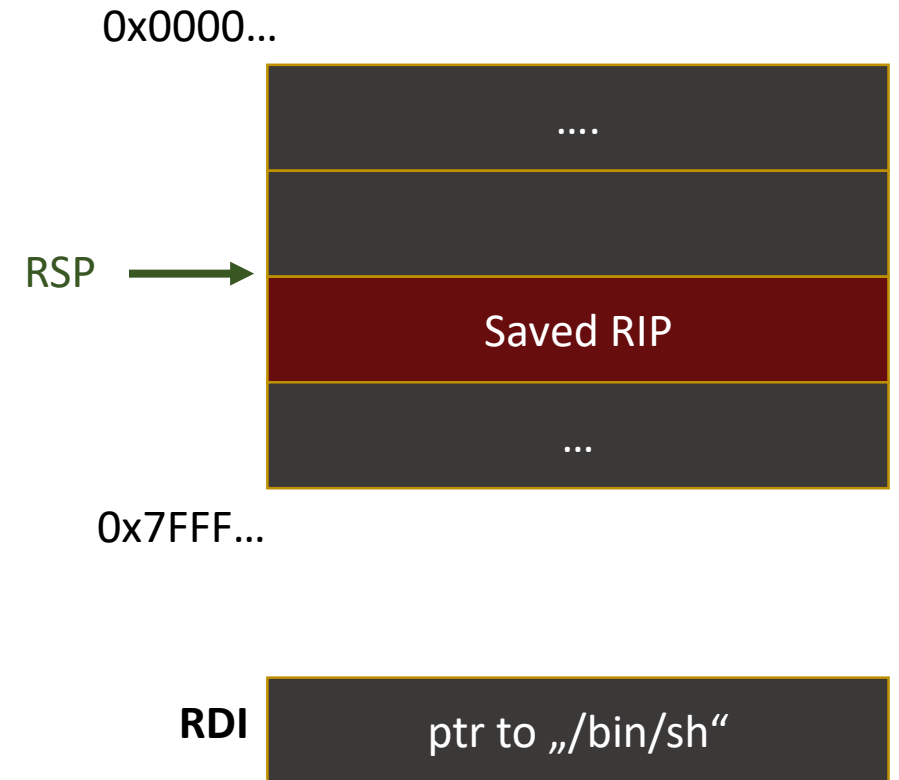


# 64 Bit – Calling convention

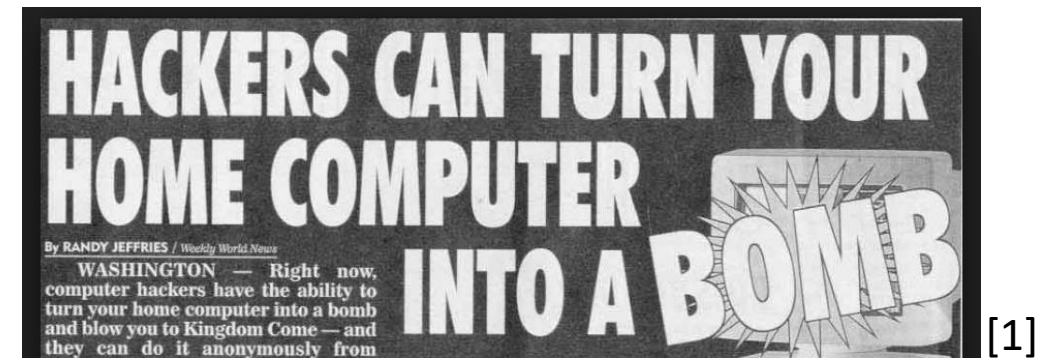
`system("/bin/sh")`

```
.binsh:  
.string "/bin/sh"  
  
main :  
    mov rdi, OFFSET.binsh  
    call system
```

← RIP

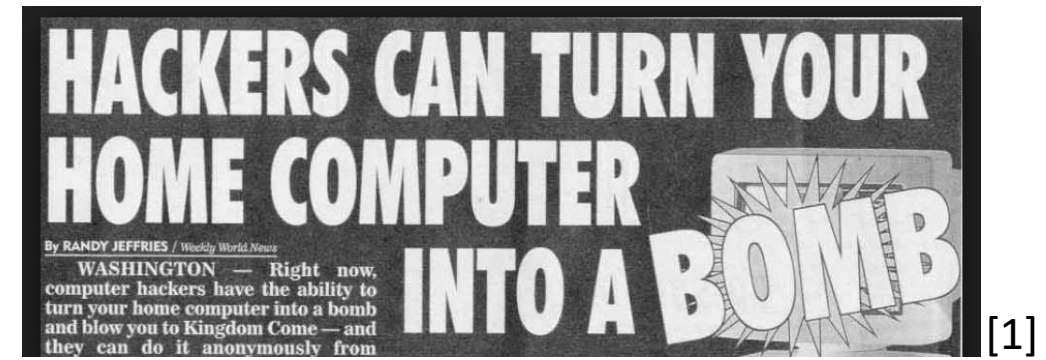


# Building ROP chains...



[1] <https://www.wired.com/2007/07/weekly-world-ne/>

# Building ROP chains...

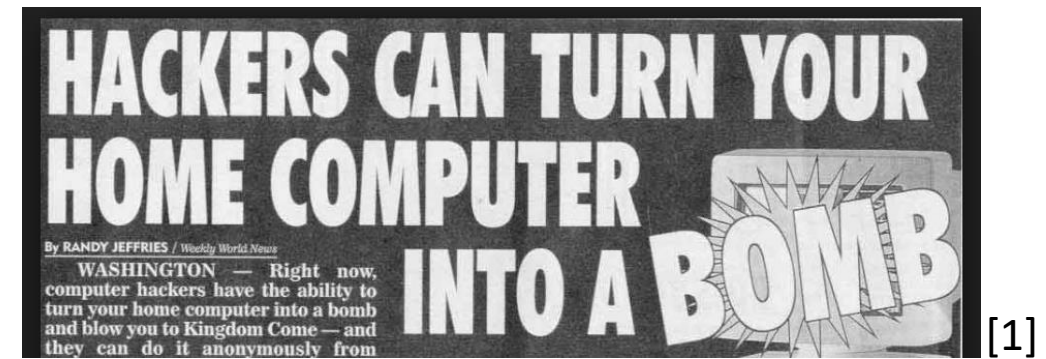


[1]



- Take snippets from the binary
- glue them together
- get the wanted code

# Building ROP chains...



[1]



- Take snippets from the binary
- glue them together
- get the wanted code

```
mov rax, 1  
ret
```

+

```
mov rbx, 2  
ret
```

+

```
pop rdi  
ret
```

vuln\_binary

```
; Attributes: bp-based frame  
; int __cdecl main(int argc, const char **argv, const char **envp)  
public main  
main proc near  
    buf_4 = 0  
    ptr_4 = 0  
    var_4 = dword ptr -4  
    push rbp  
    mov rbp, rsp  
    sub rsp, 20h  
    mov [rbp+var_4], 0  
    call _read  
    mov rax, [rbp+buf]  
    mov edx, 50h  
    mov rsi, rax  
    mov edi, 0  
    call _read  
    mov [rbp+var_4], eax  
    lea rax, [rbp+buf]  
    mov rsi, rax  
    lea rdi, format  
    mov eax, 0  
    call _printf  
    mov eax, 0  
    leave  
    retn  
main endp
```

[1] <https://www.wired.com/2007/07/weekly-world-ne/>

# Building ROP-chains ...

( „ret“ = pop RIP)

0x400111

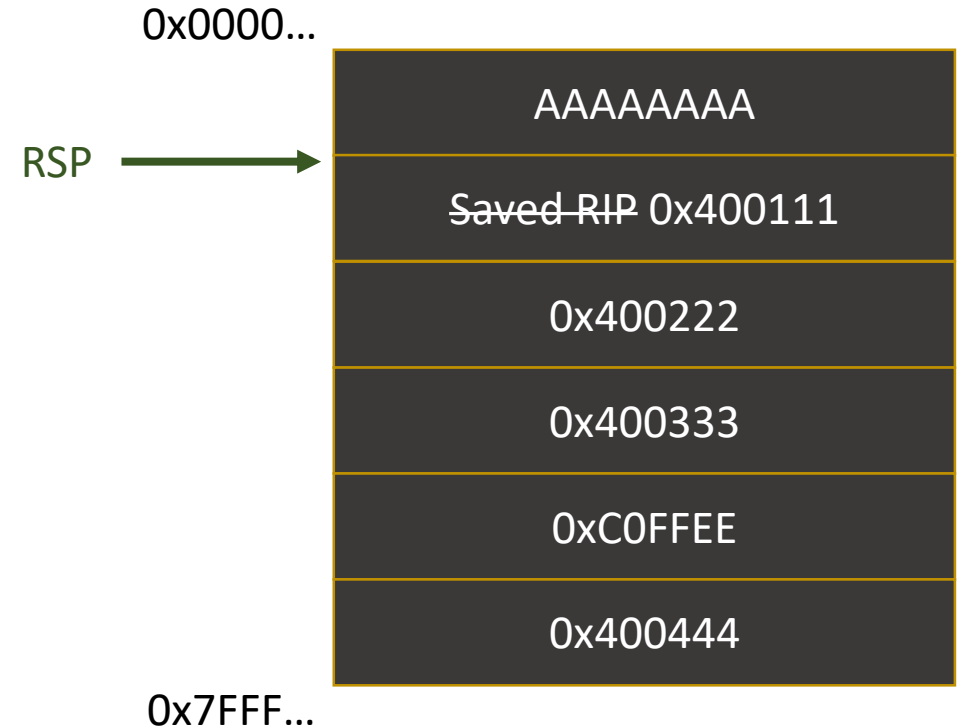
```
mov rax, 1
ret
```

0x400222

```
mov rbx, 2
ret
```

0x400333

```
pop rdi
ret
```



RAX



RBX



RDI



# Building ROP-chains ...

0x400111

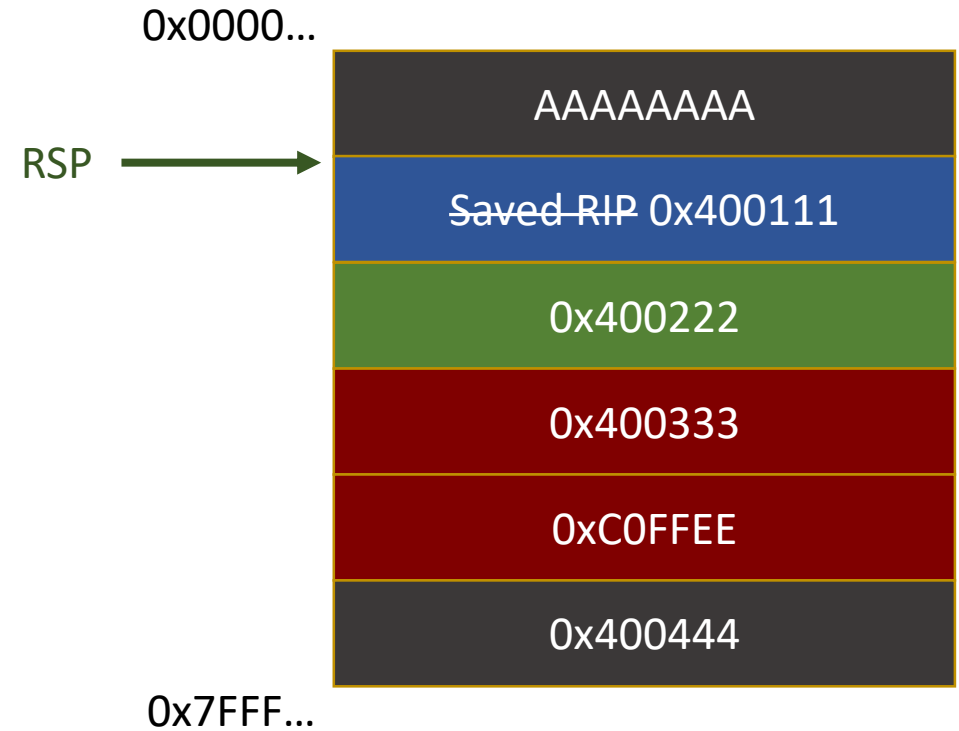
```
mov rax, 1  
ret
```

0x400222

```
mov rbx, 2  
ret
```

0x400333

```
pop rdi  
ret
```



RAX

RBX

RDI

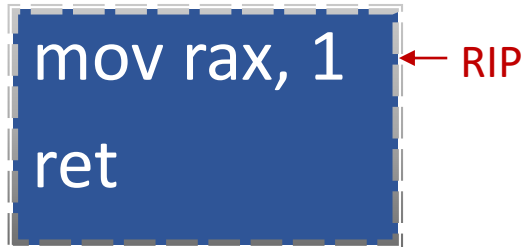




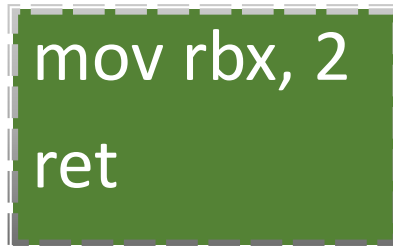
# Building ROP-chains ...

( „ret“ = pop RIP)

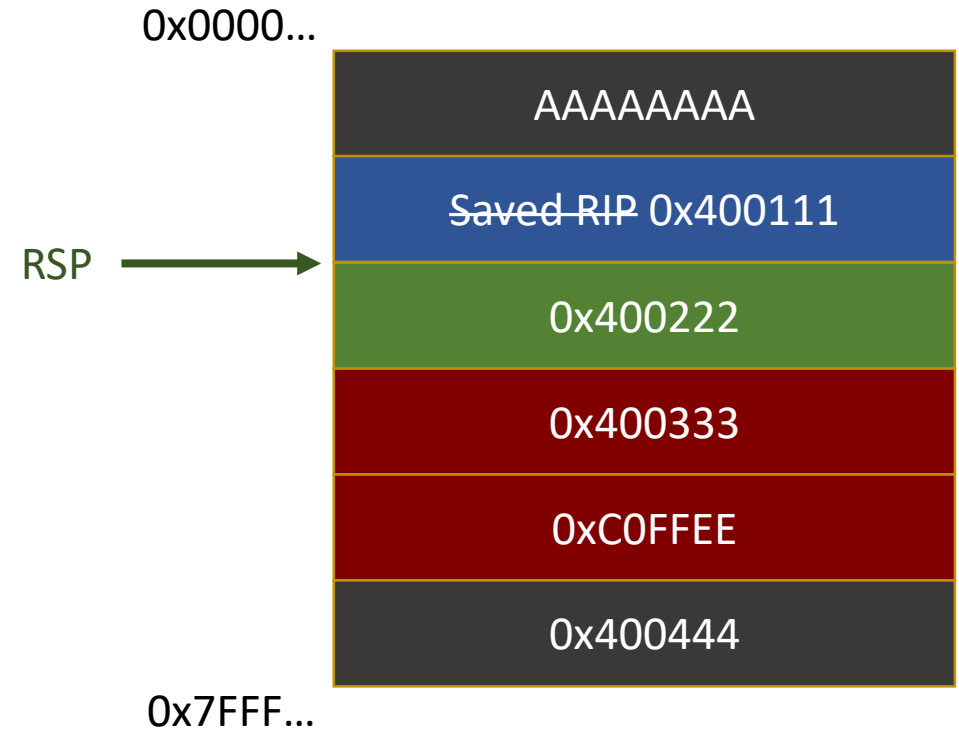
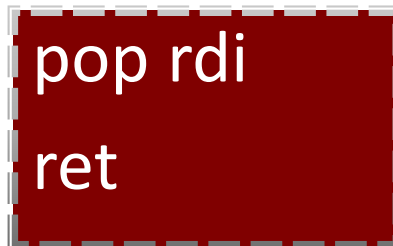
0x400111



0x400222



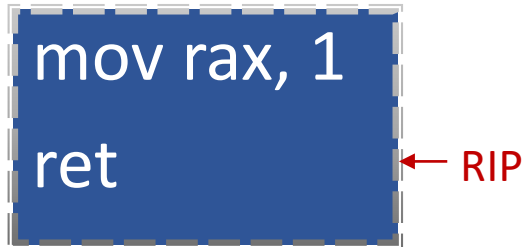
0x400333



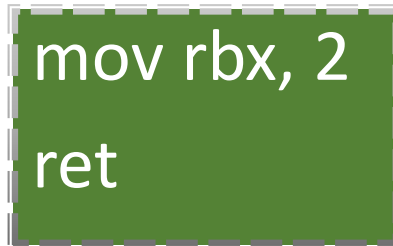
# Building ROP-chains ...

( „ret“ = pop RIP)

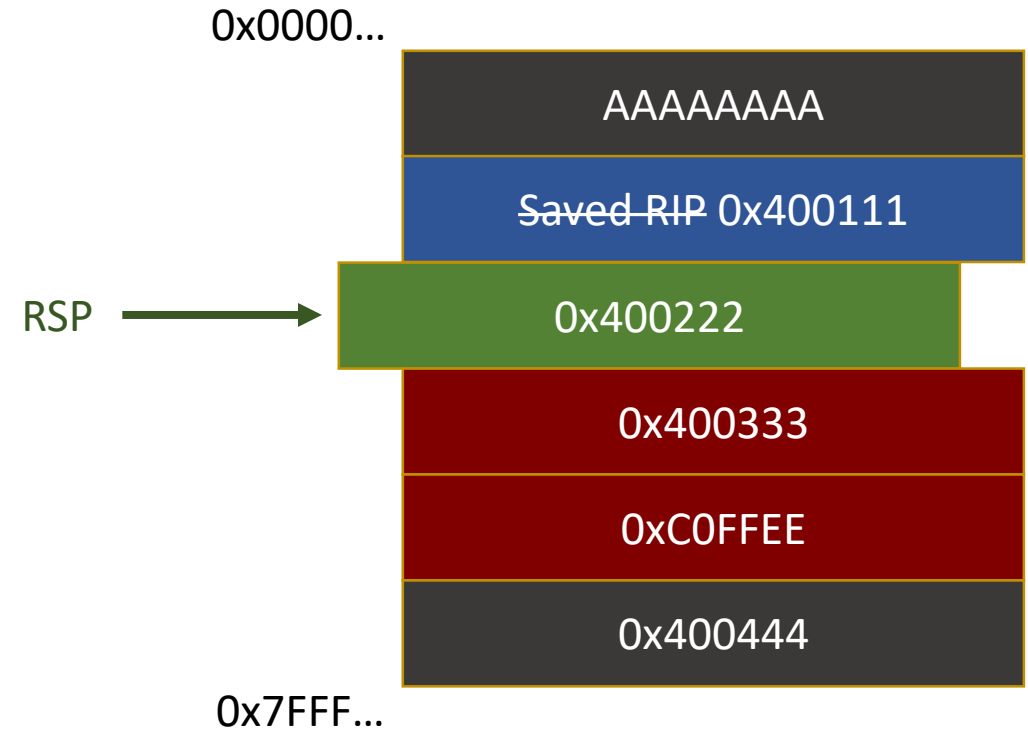
0x400111



0x400222



0x400333



# Building ROP-chains ...

0x400111

```
mov rax, 1
ret
```

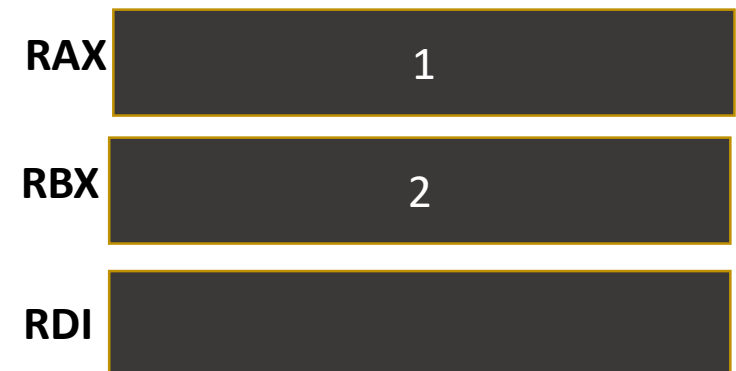
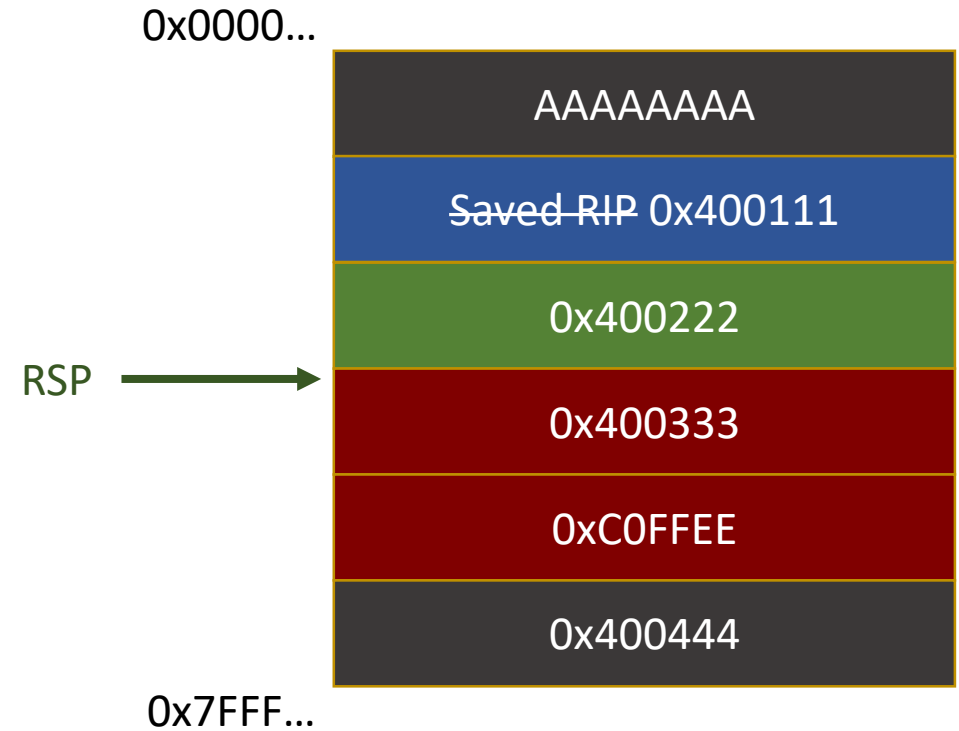
0x400222

```
mov rbx, 2
ret
```

← RIP

0x400333

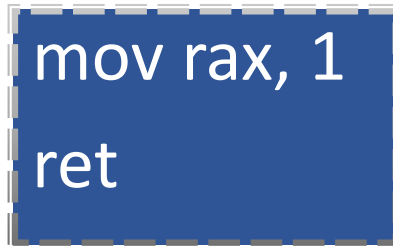
```
pop rdi
ret
```



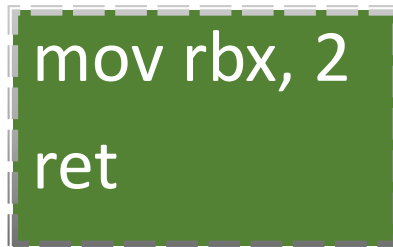
# Building ROP-chains ...

( „ret“ = pop RIP)

0x400111

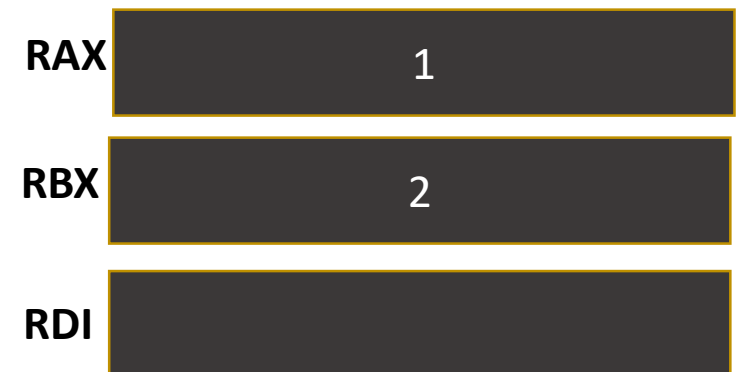
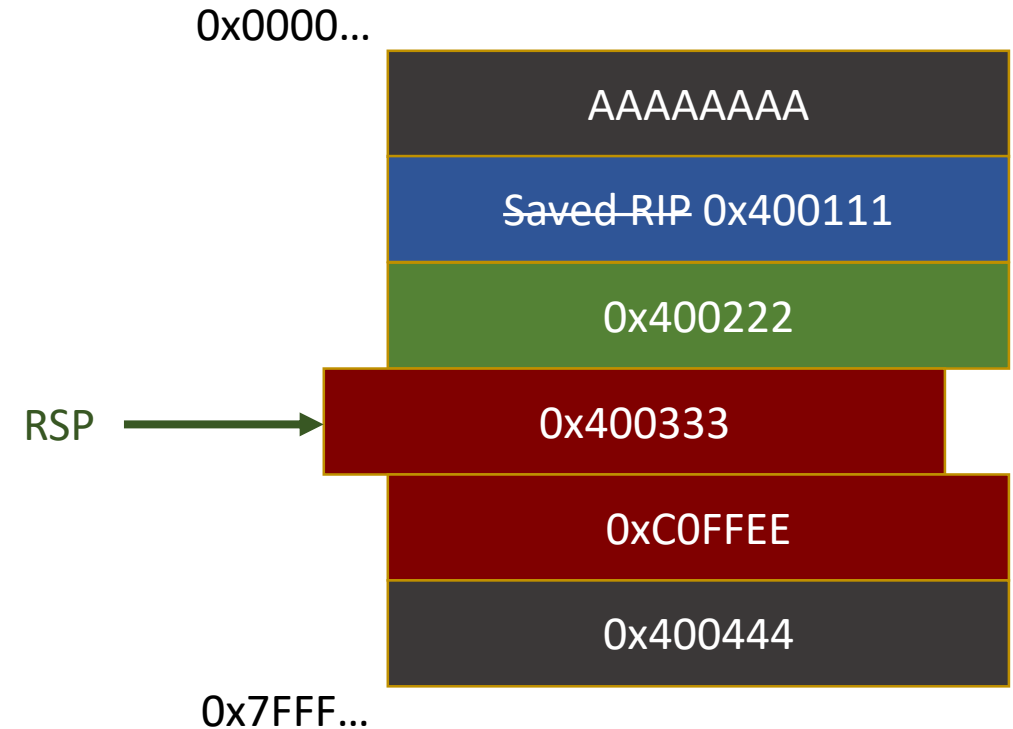


0x400222



← RIP

0x400333



# Building ROP-chains ...

0x400111

```
mov rax, 1  
ret
```

0x400222

```
mov rbx, 2  
ret
```

0x400333

```
pop rdi  
ret
```

← RIP

0x0000...

AAAAAAAA

~~Saved RIP~~ 0x400111

0x400222

0x400333

0xC0FFEE

0x400444

RSP →

0x7FFF...

RAX

1

RBX

2

RDI

0xC0FFEE

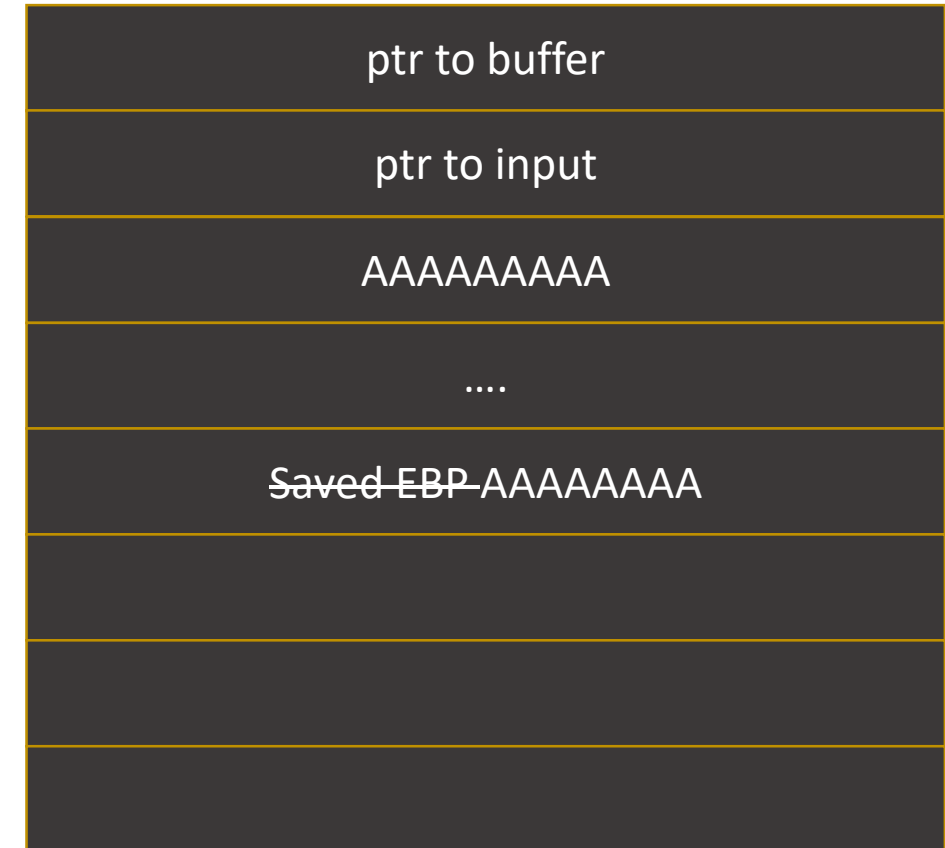


# 64 Bit – simple ROP-chain

Payload = "A" \* 32

+ "AAAAAAAA" (~~saved EBP~~)

0x0000...



0x7FFF...



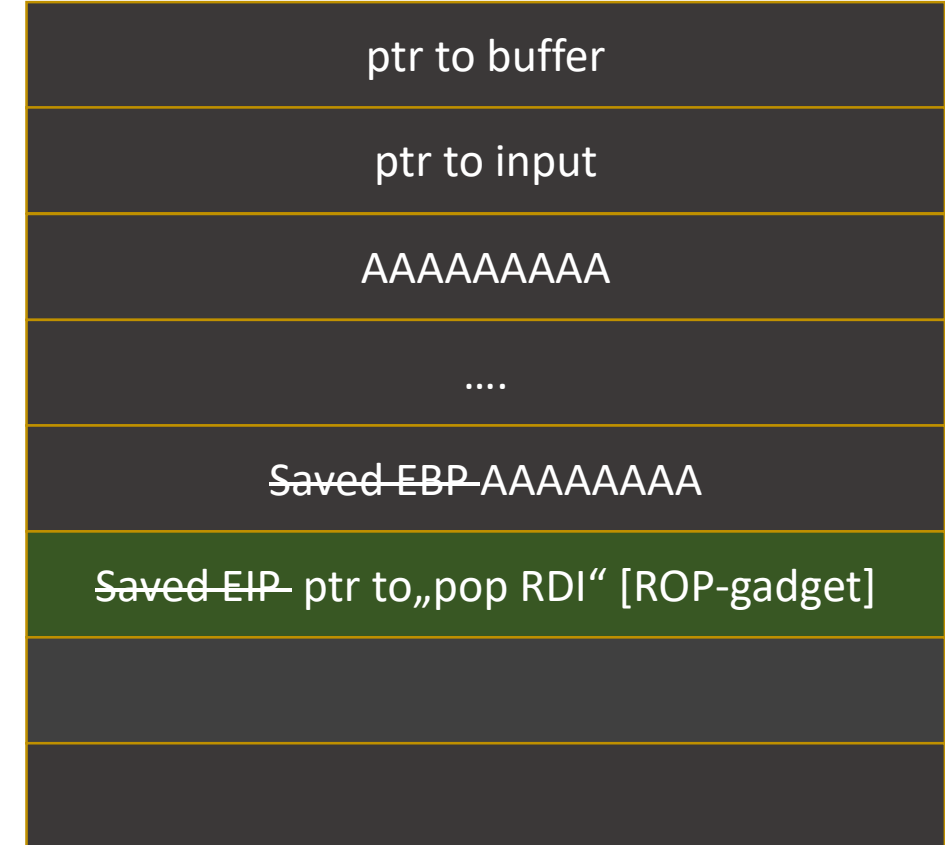
# 64 Bit – simple ROP-chain

Payload = "A" \* 32

+ "AAAAAAAA" (~~saved EBP~~)

+ address "pop RDI; ret" [ROP-gadget] (~~saved EIP~~)

0x0000...



0x7FFF...



# 64 Bit – simple ROP-chain

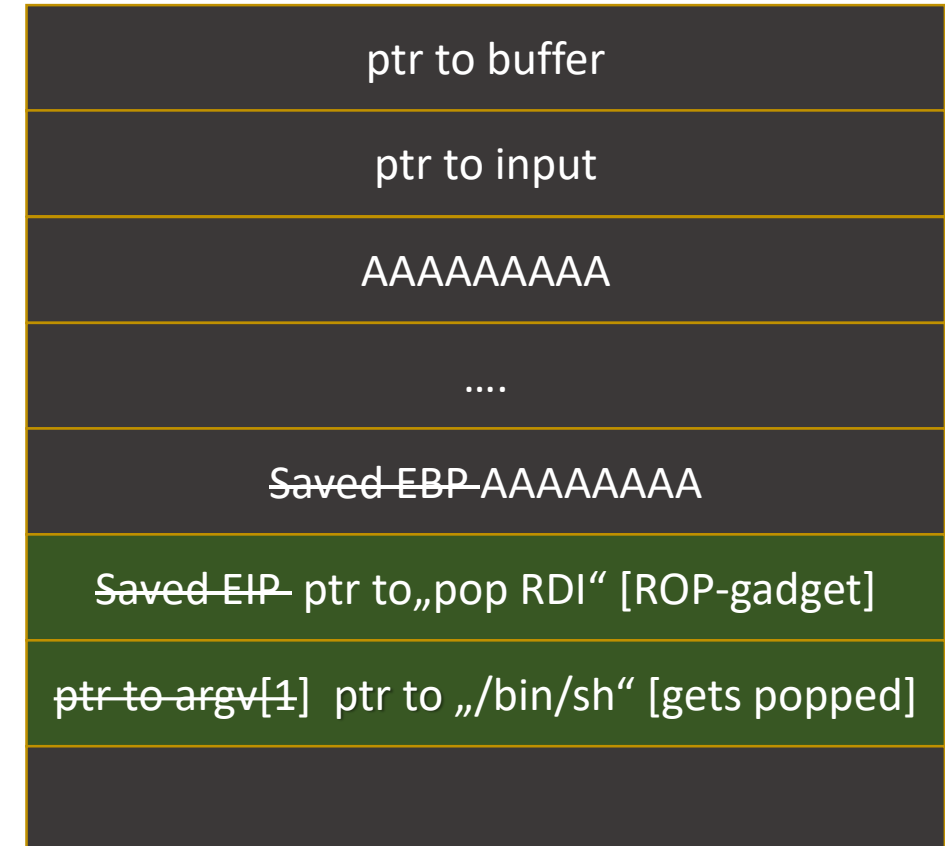
Payload = "A" \* 32

+ "AAAAAAAA" (~~saved EBP~~)

+ address "pop RDI; ret" [ROP-gadget] (~~saved EIP~~)

+ address "/bin/sh" [value that gets popped in RDI]

0x0000...



0x7FFF...





# 64 Bit – simple ROP-chain

Payload = "A" \* 32

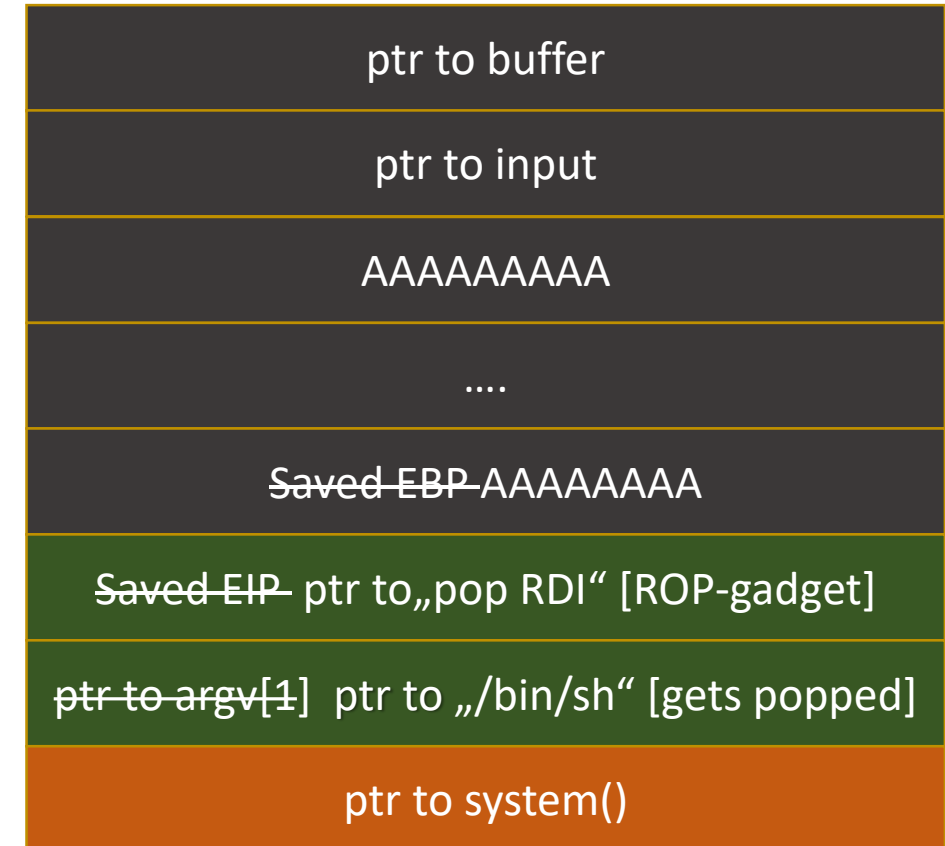
+ "AAAAAAAA" (~~saved EBP~~)

+ address "pop RDI; ret" [ROP-gadget] (~~saved EIP~~)

+ address "/bin/sh" [value that gets popped in RDI]

+ address of system

0x0000...



0x7FFF...



# x86 – ROPgadget

Why is there code we don't see while disassembling?

# x86 – ROPgadget

Why is there code we don't see while disassembling?

push 0x11c35faa

RIP → 0x68 0xaa 0x5f 0xc3 0x11



# x86 – ROPgadget

Why is there code we don't see while disassembling?

push 0x11c35faa

RIP → 0x68 0xaa 0x5f 0xc3 0x11  
RIP ↗

# x86 – ROPgadget

Why is there code we don't see while disassembling?

```
push 0x11c35faa
```

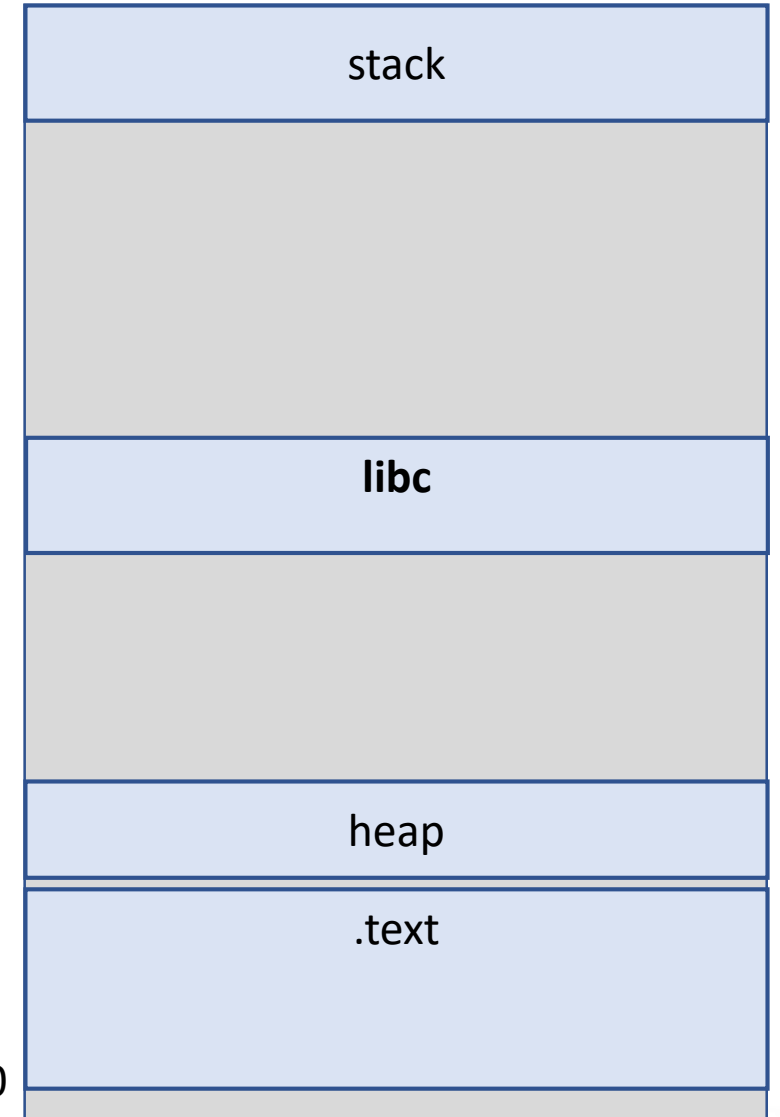
```
0x68 0xaa 0x5f 0xc3 0x11
```

```
RIP → pop rdi; ret
```

# 64 Bit – ASLR enabled

- ASLR: Address Space Layout Randomization
- System wide security mechanism

0x7FFF...

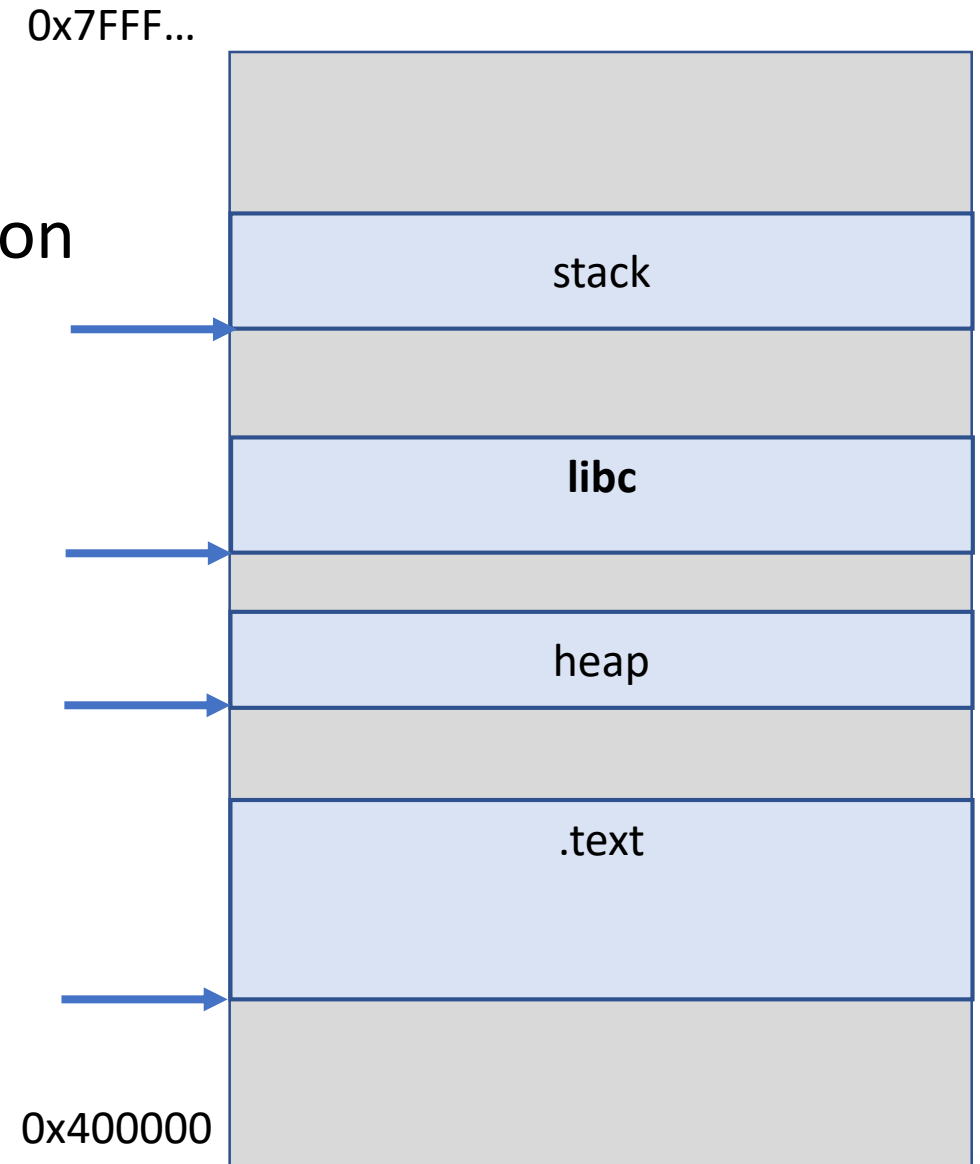


0x400000



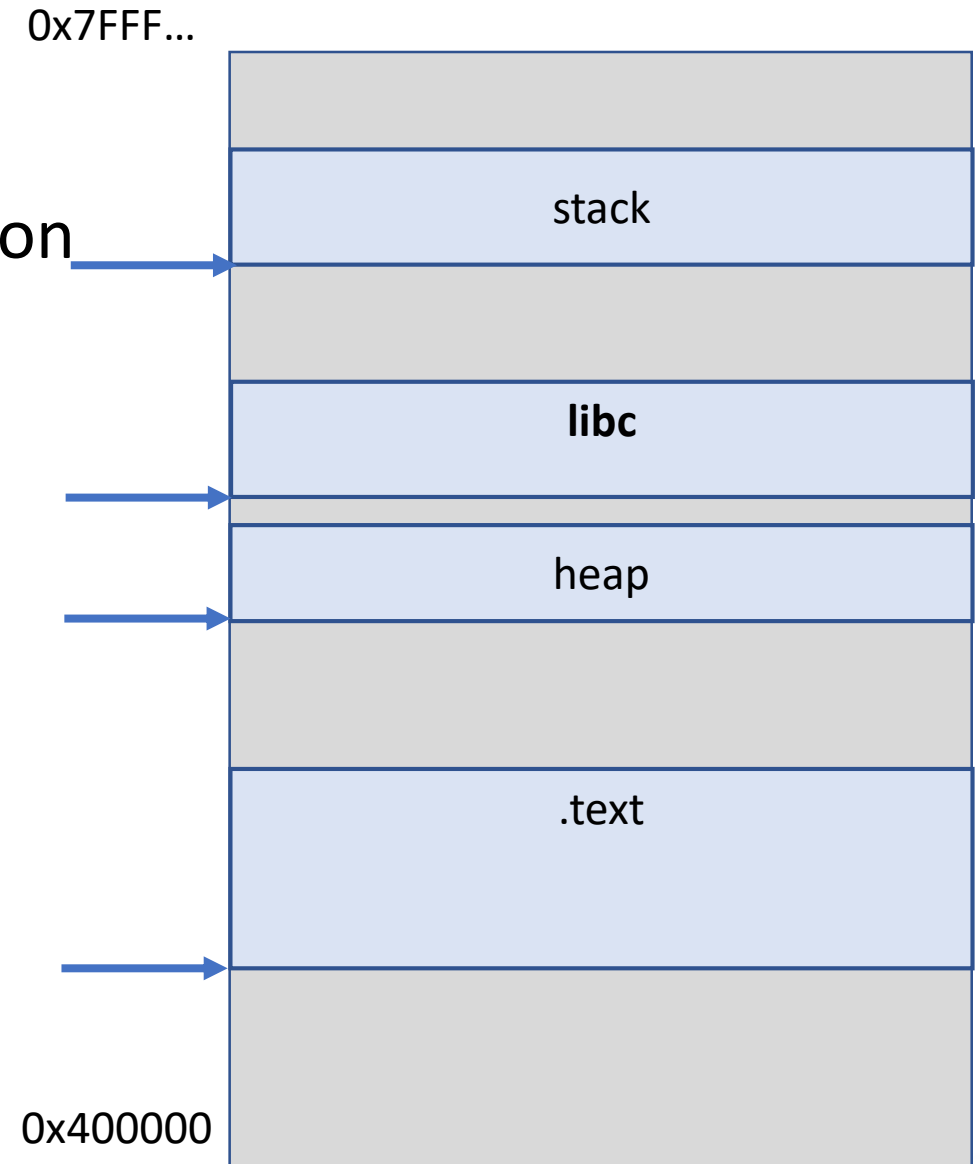
# 64 Bit – ASLR enabled

- ASLR: Address Space Layout Randomization
- System wide security mechanism
- Base addresses of each section are randomized
- With each execution of the program addresses change unpredictable for an attacker



# 64 Bit – ASLR enabled

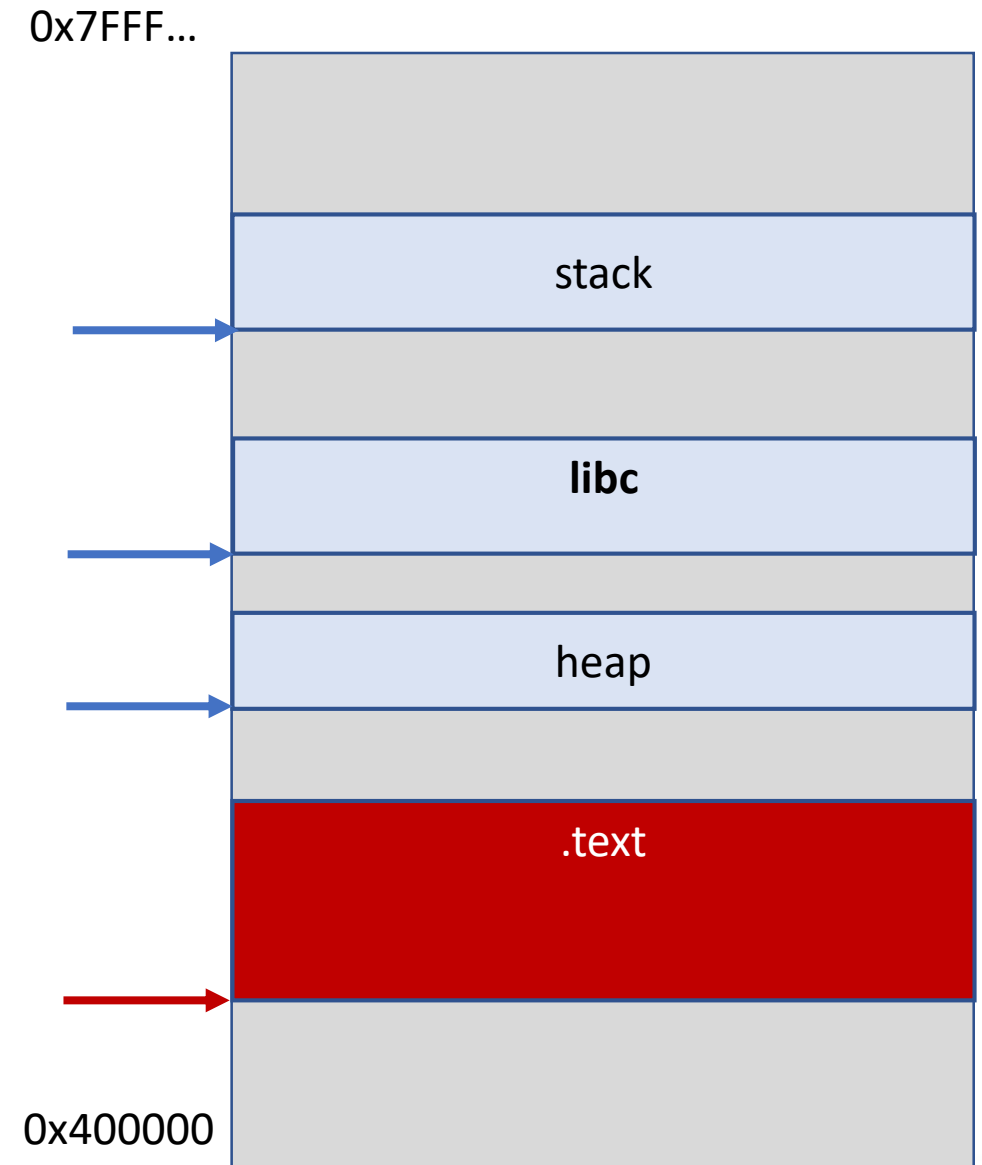
- ASLR: Address Space Layout Randomization
- System wide security mechanism
- Base addresses of each section are randomized
- With each execution of the program addresses change unpredictable for an attacker





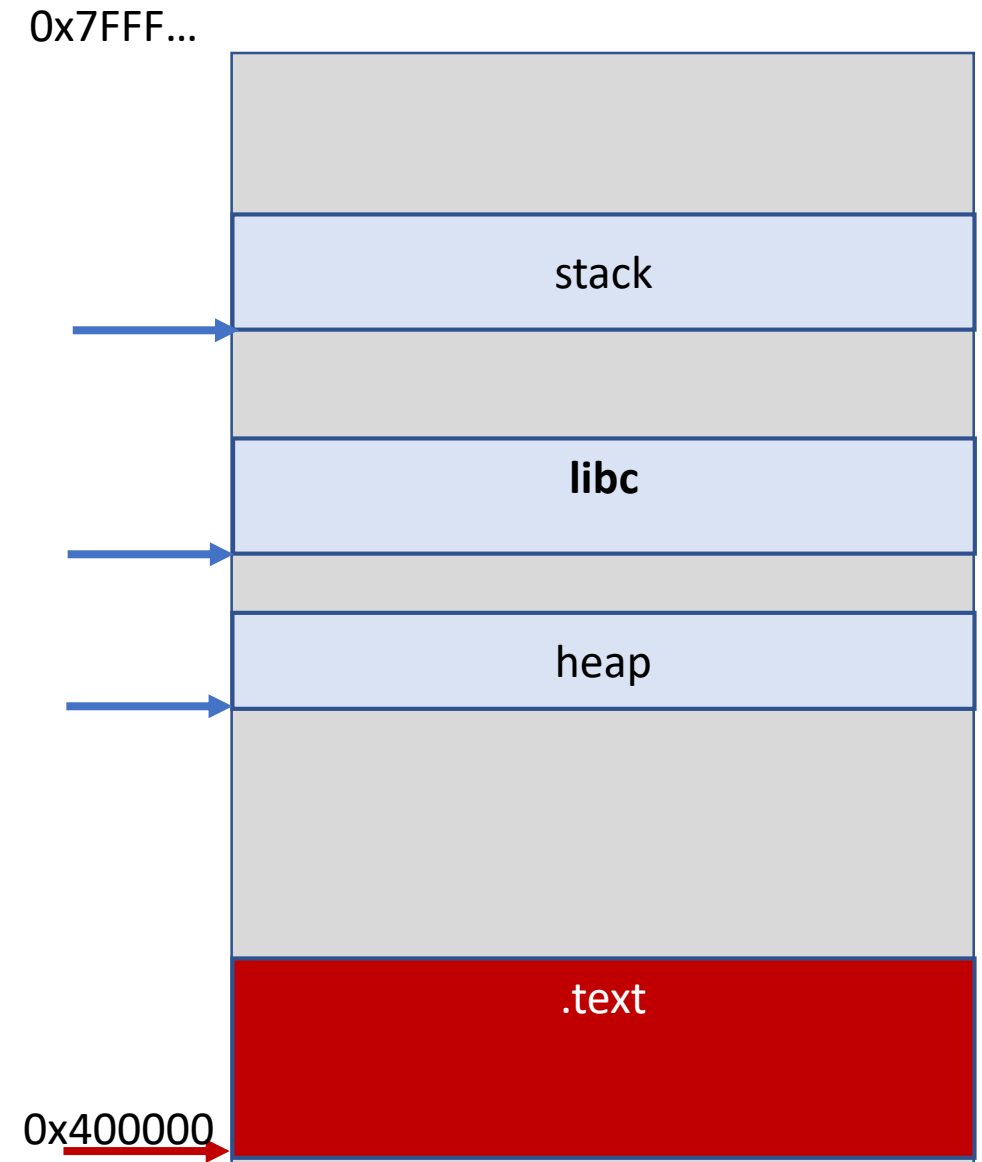
# 64 Bit – ASLR enabled

- **PIE (Position Independent Executable) ENABLED**



# 64 Bit – ASLR enabled

- **PIE (Position Independent Executable) DISABLED**



# 64 Bit – ASLR enabled - Strategy

1. Call printf/puts with our ROP-chain, and leak with this an address of the libc => calculate libc base address
2. Find a gadget in the binary to trigger the Buffer Overflow again
3. Perform the known exploit with the new calculated addresses of system and /bin/sh

# GOT and PLT

**GOT: Global Offset Table**

**PLT: Procedure Linkage Table**

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

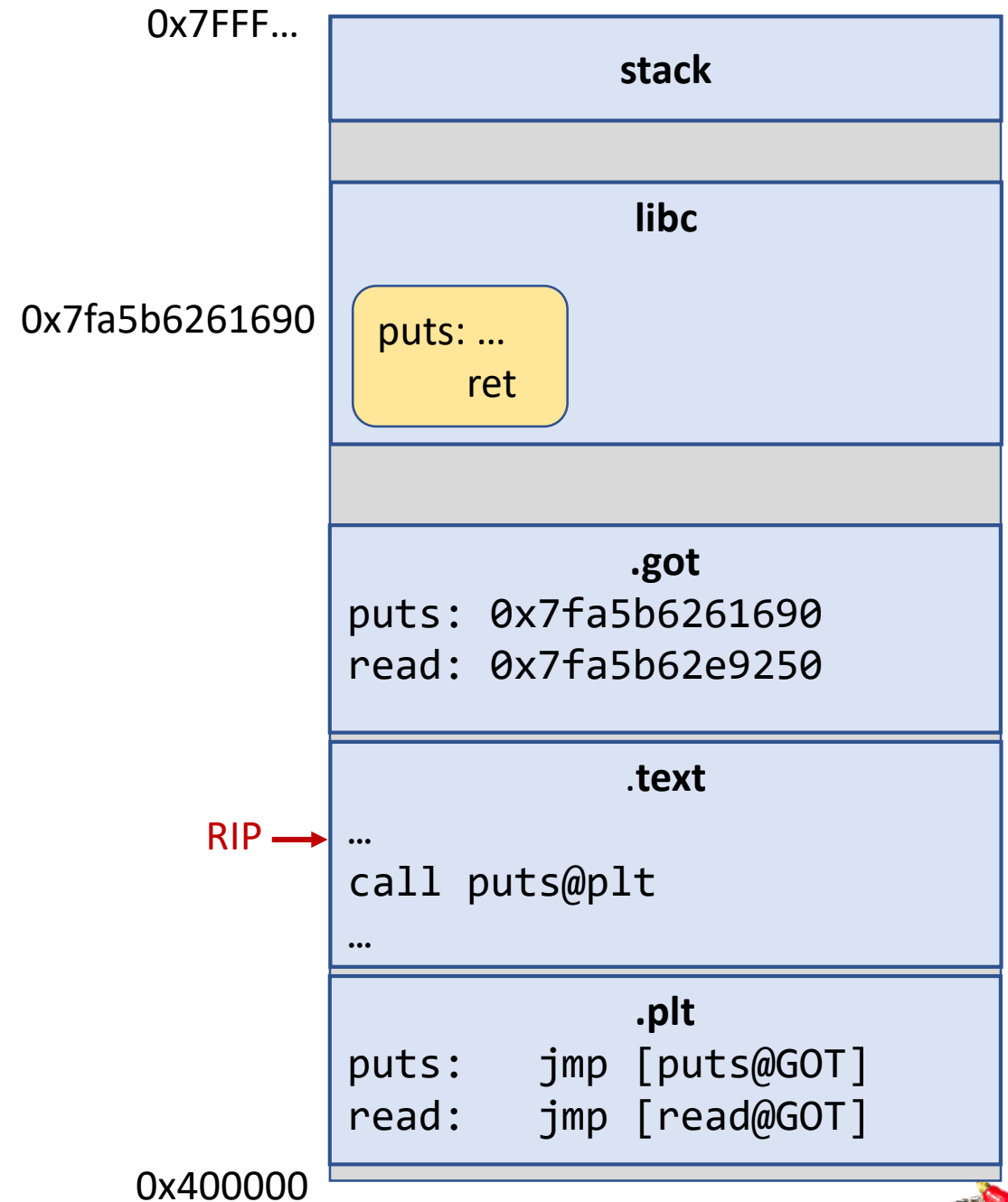


# GOT and PLT

**GOT: Global Offset Table**

**PLT: Procedure Linkage Table**

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

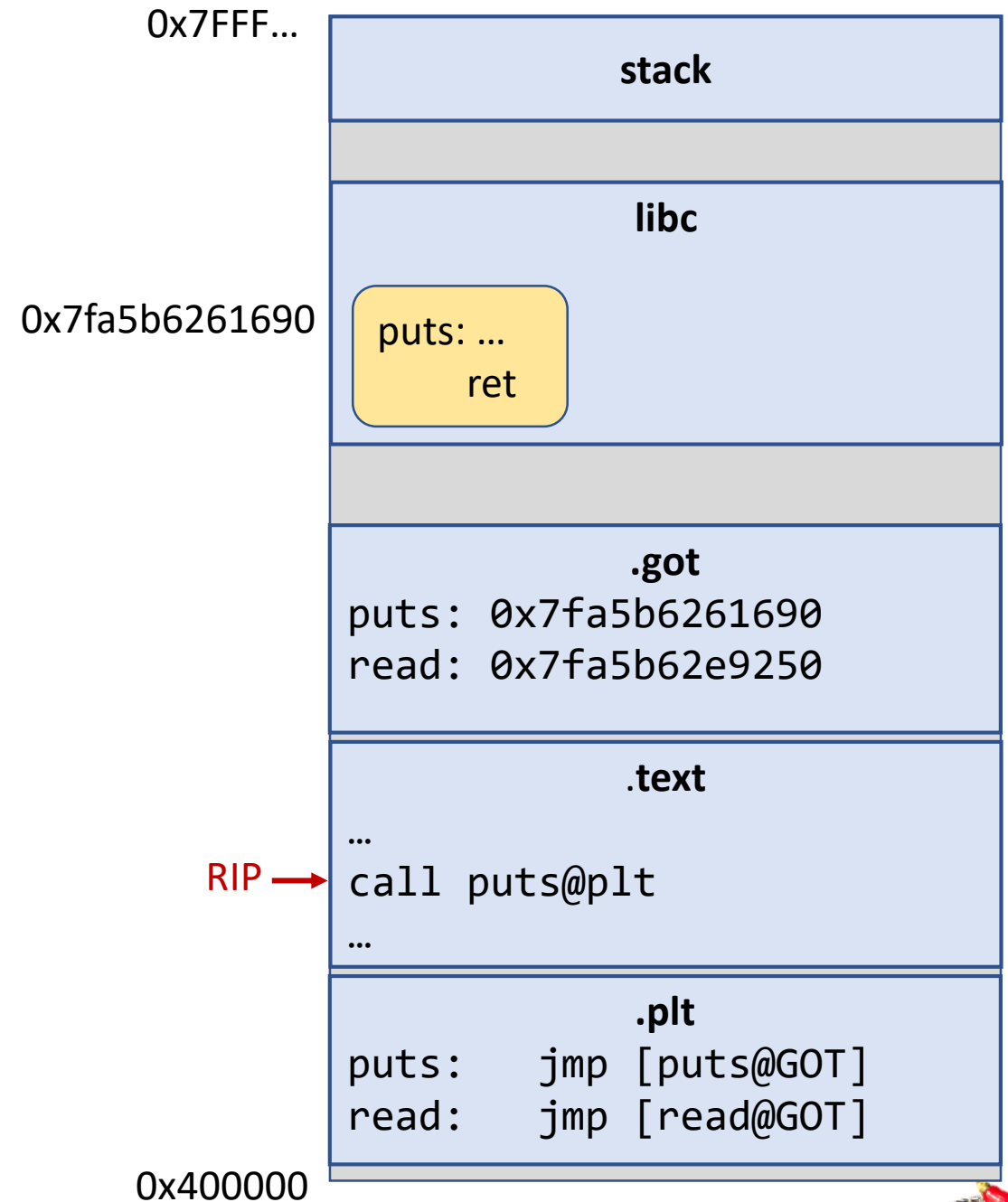


# GOT and PLT

**GOT: Global Offset Table**

**PLT: Procedure Linkage Table**

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

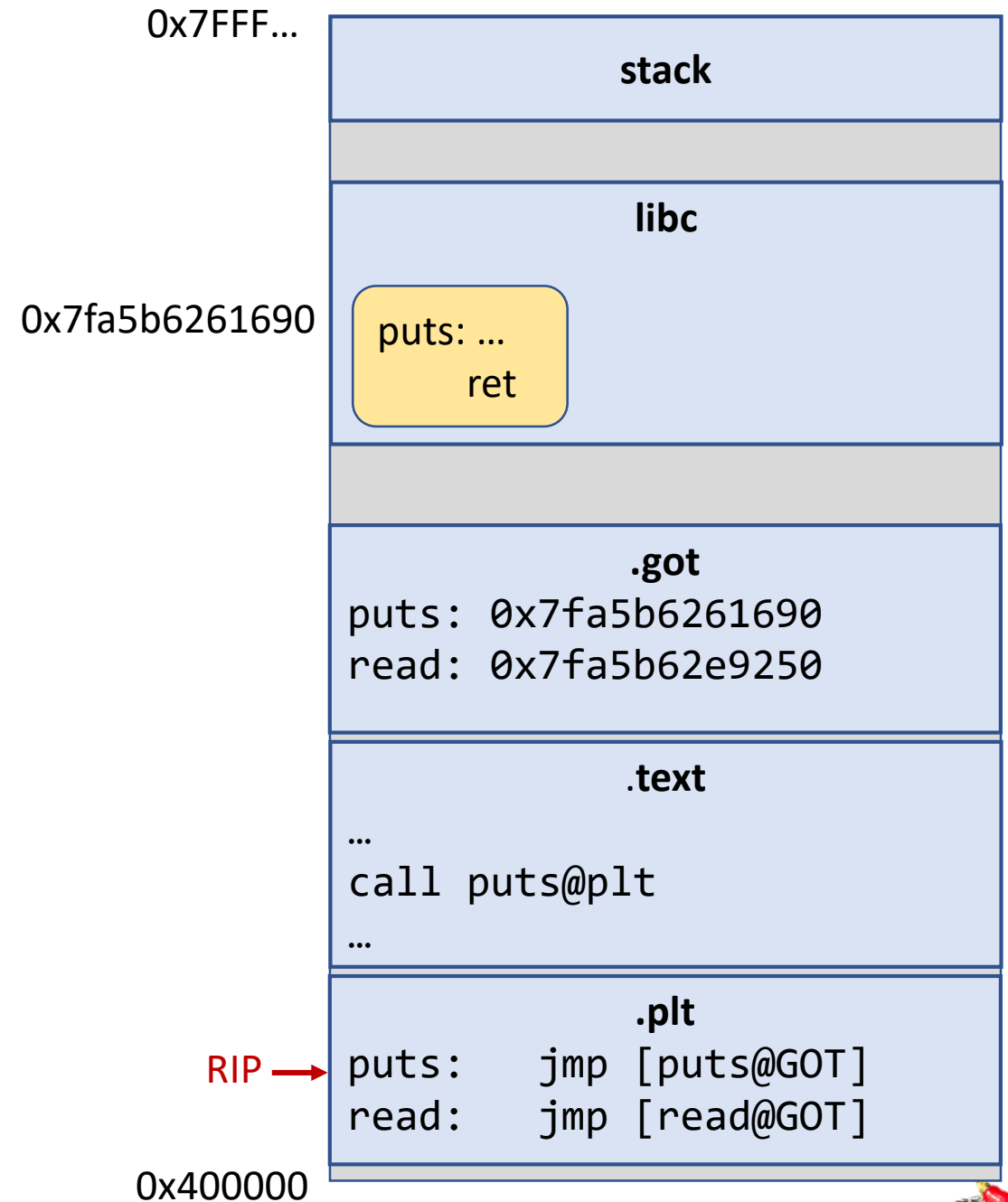


# GOT and PLT

**GOT: Global Offset Table**

**PLT: Procedure Linkage Table**

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

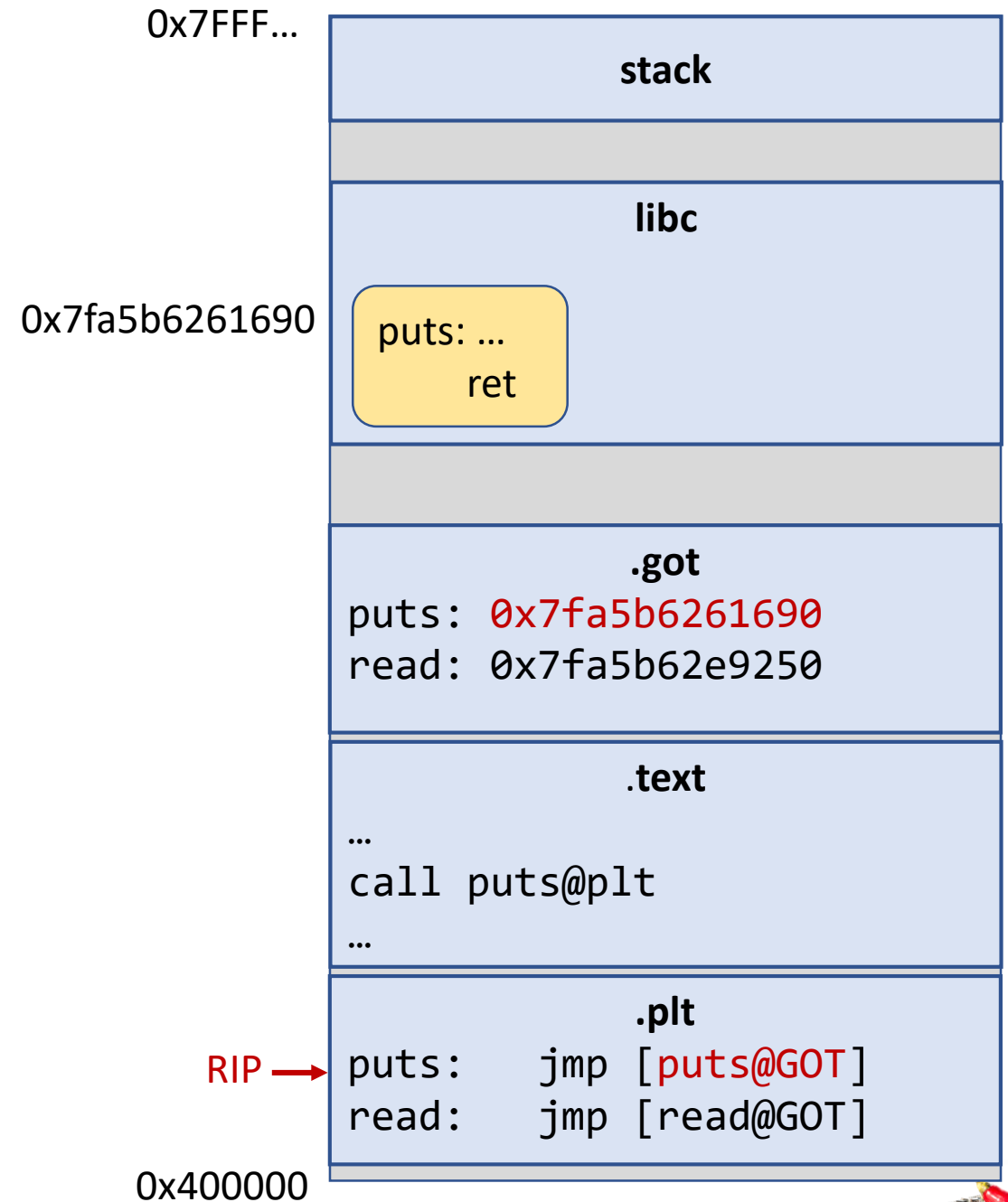


# GOT and PLT

**GOT: Global Offset Table**

**PLT: Procedure Linkage Table**

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables





# GOT and PLT

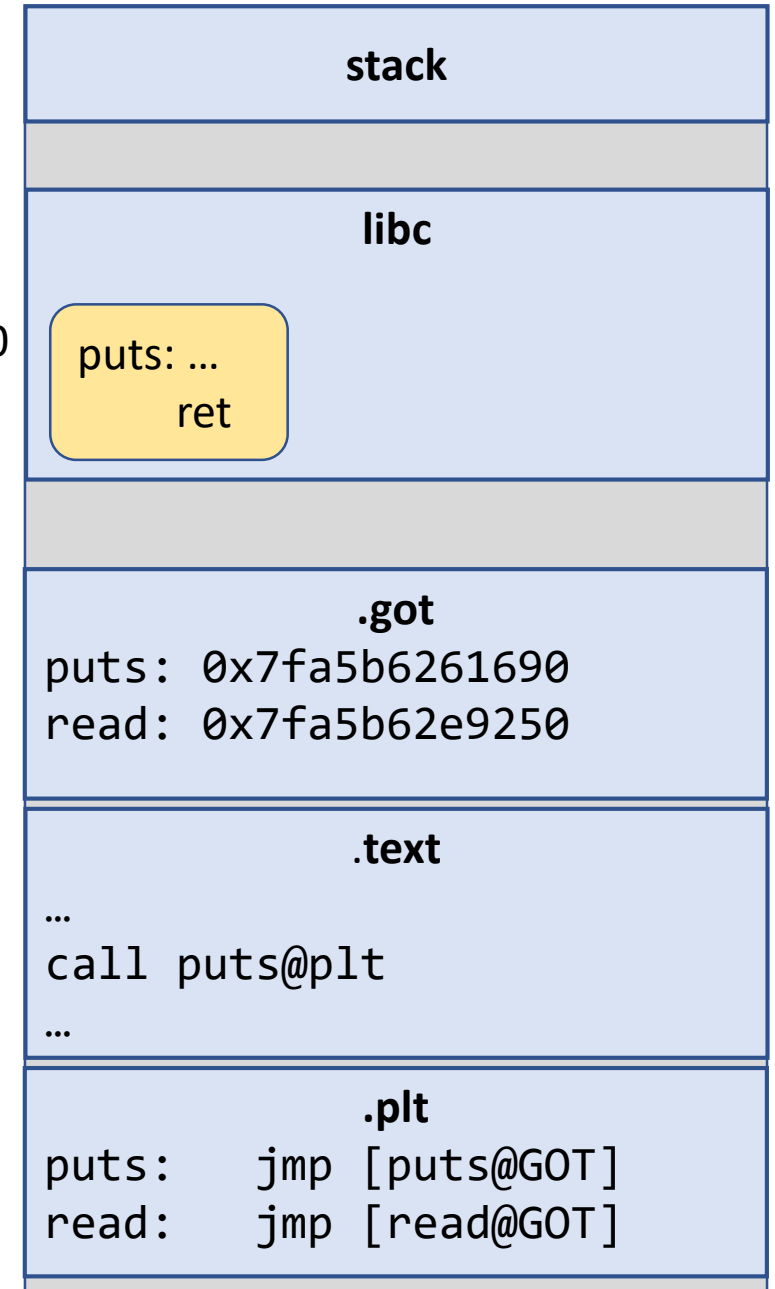
**GOT: Global Offset Table**

**PLT: Procedure Linkage Table**

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

RIP → 0x7fa5b6261690

0x7FFF...



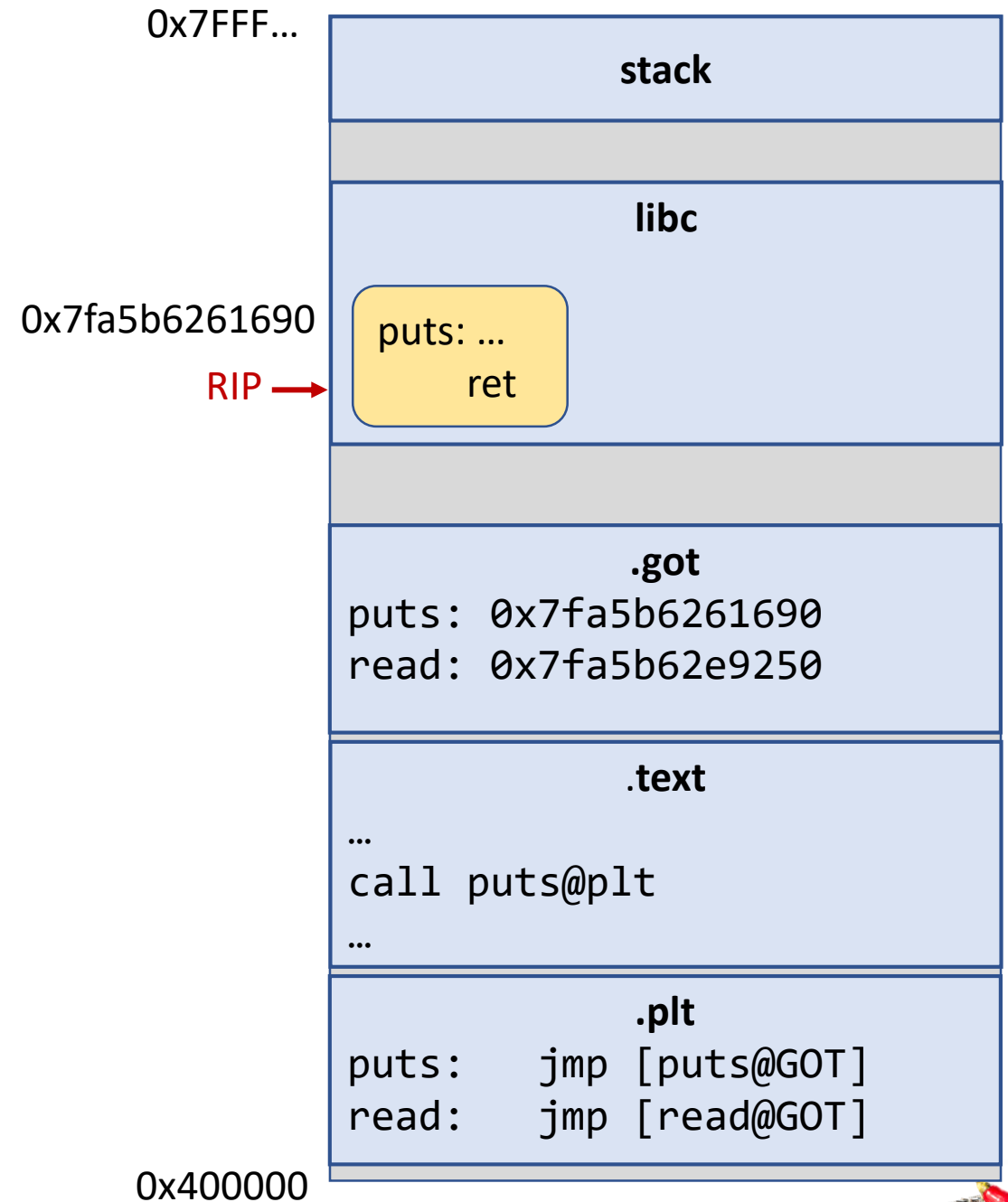
0x400000

# GOT and PLT

**GOT: Global Offset Table**

**PLT: Procedure Linkage Table**

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

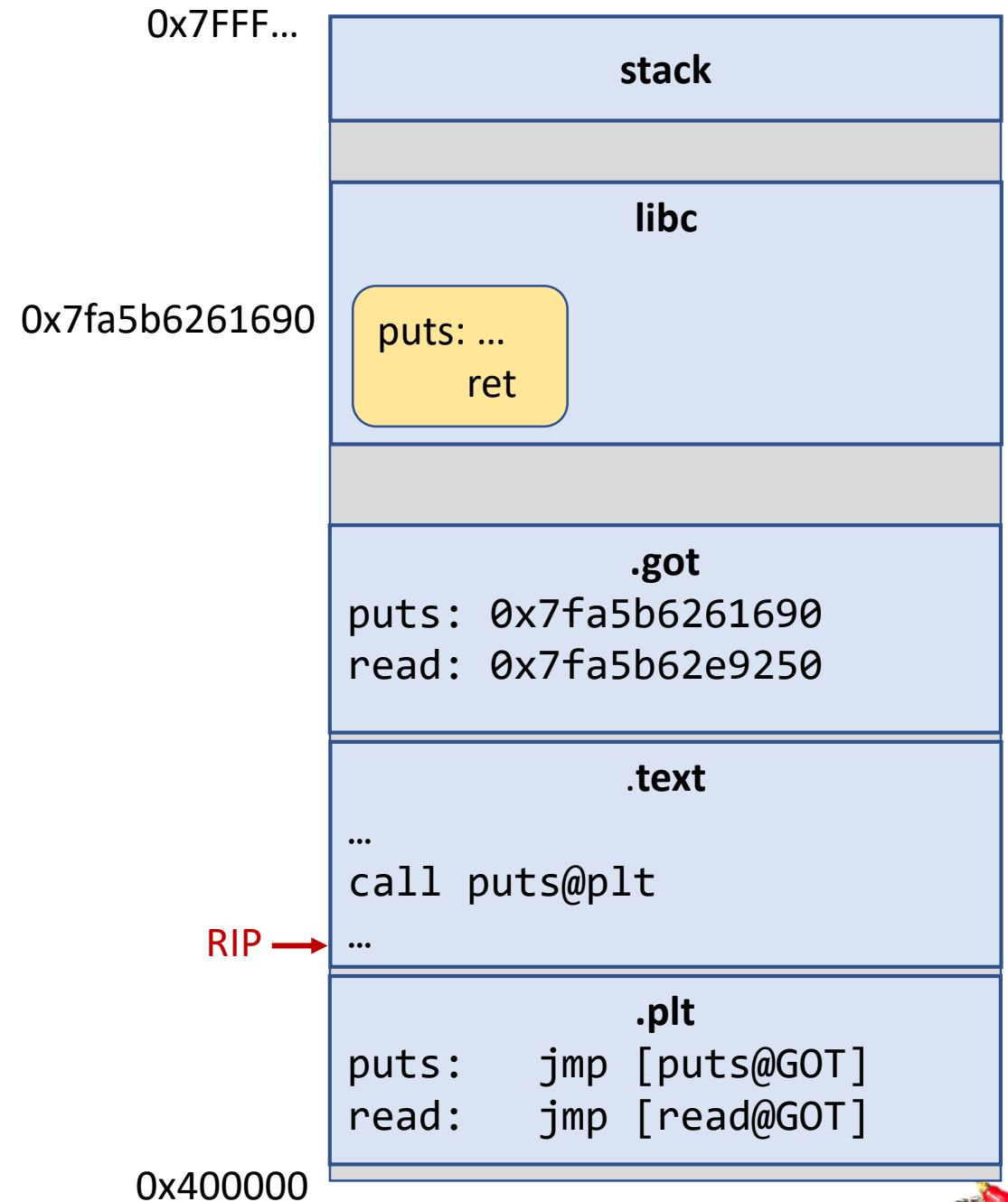


# GOT and PLT

**GOT: Global Offset Table**

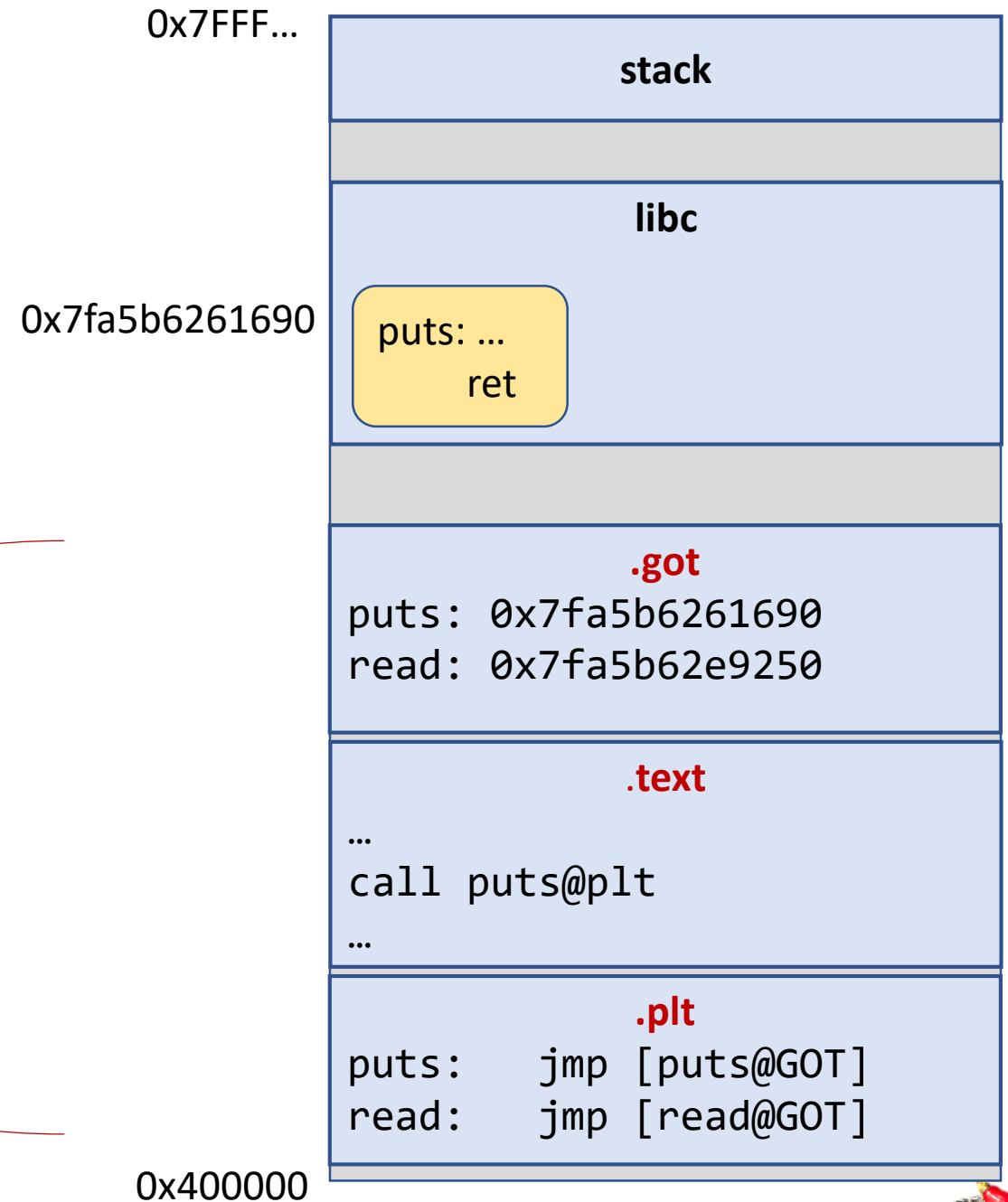
**PLT: Procedure Linkage Table**

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

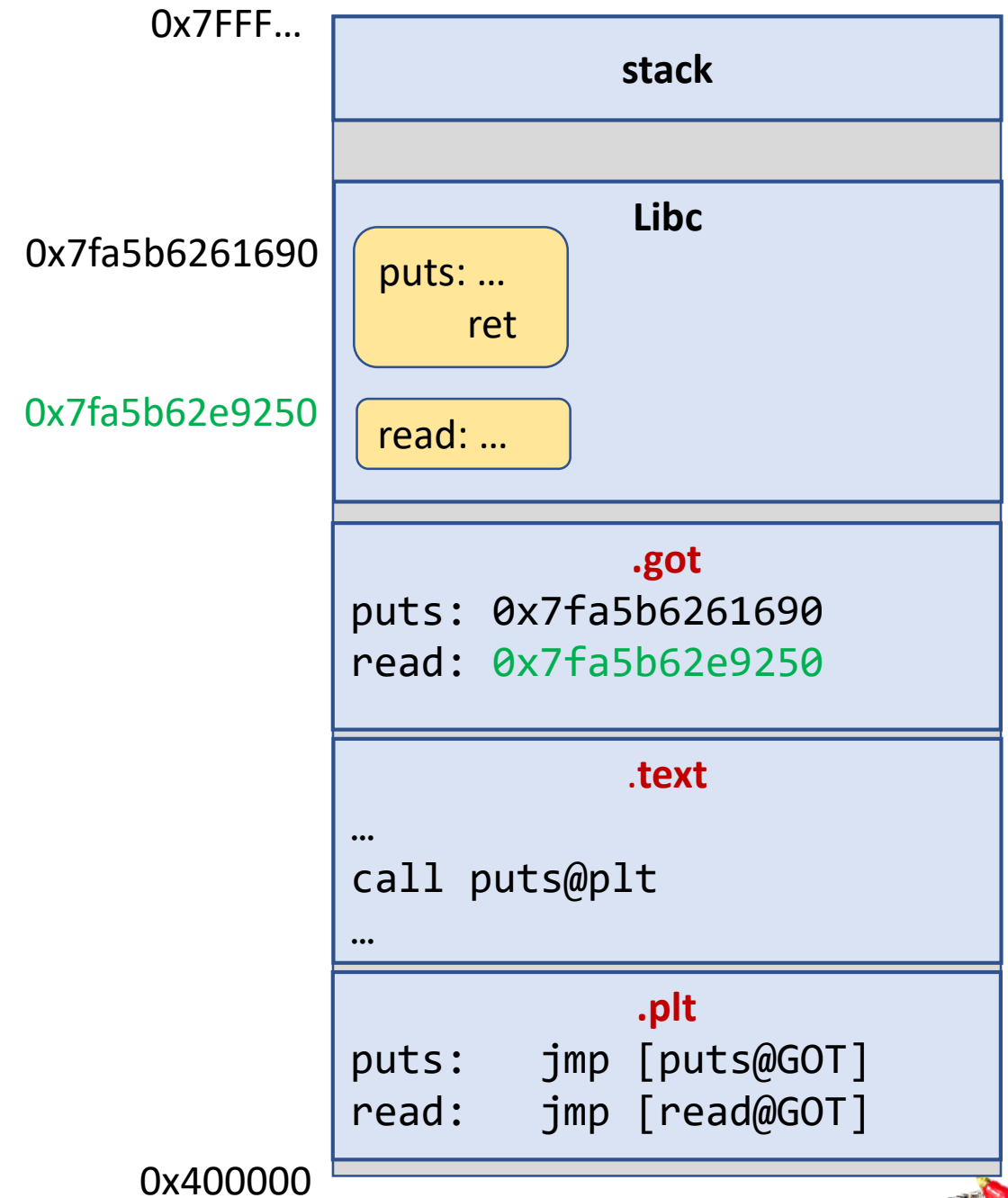


# GOT and PLT

Not randomized if not compiled  
as PIE



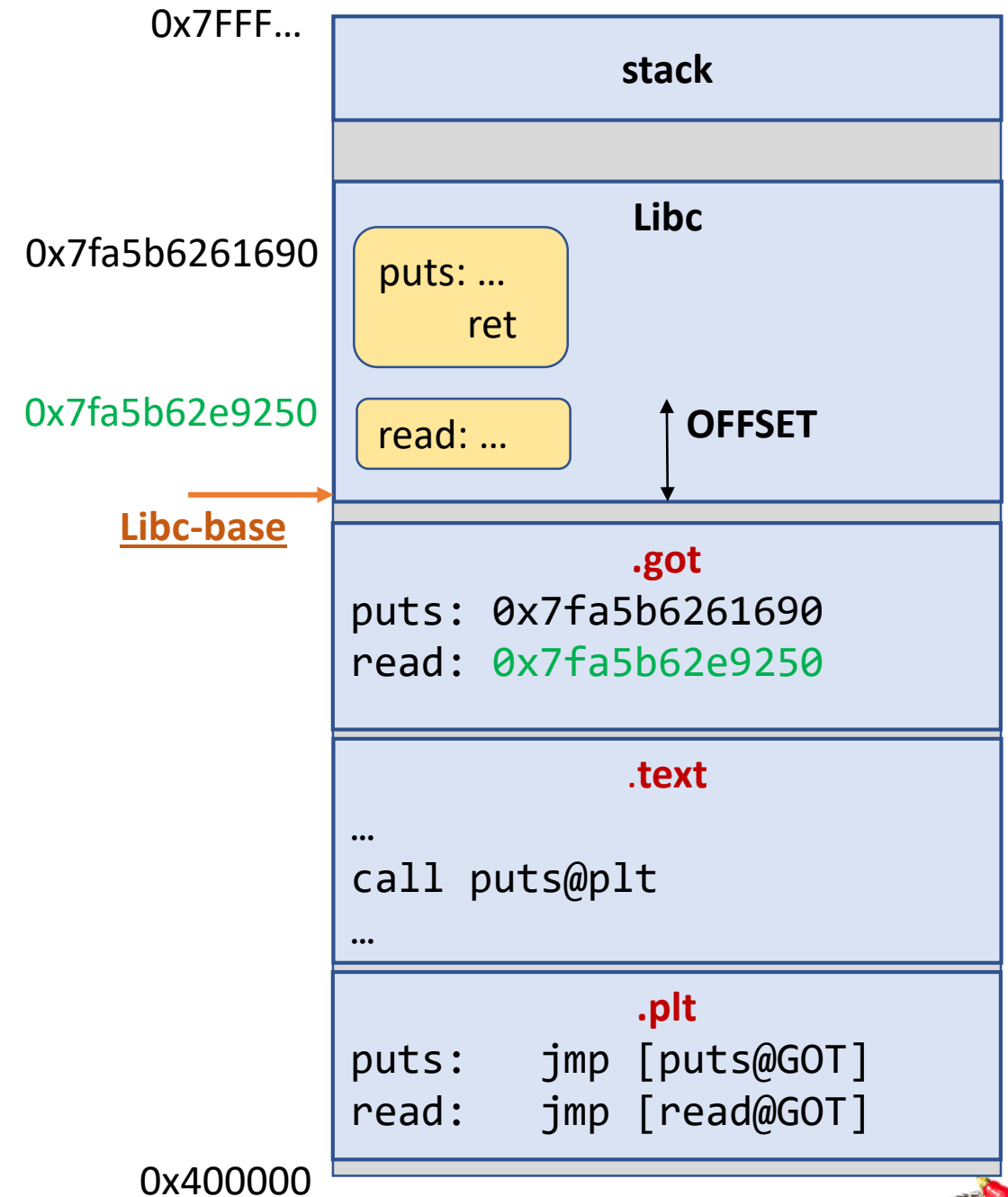
# GOT and PLT



# GOT and PLT

$\text{libcbase} = [\text{leaked address}] - \text{OFFSET}$

$\text{libcbase} = 0x7fa5b62e9250 - \text{OFFSET}$



# Leak and jump back to main

- Goal:

`puts([read@got])` → prints the address of `read@got` → leak to libc!

RDI: `[read@got]`

RIP: `puts@plt`

# ... where can I get more ROP?

Channels:

LiveOverflow Youtube Channel – Binary series

GynvaelEN: Hacking Livestream #20: Return-oriented Programming

Training:

<https://picoctf.com/> (binaries in higher levels are a good exercise!)

<https://ringzer0ctf.com> (Linux pwnage – the important ones are online)

<https://github.com/RPISEC/MBE> (RPI-sec, lab 07)

overthewire

Every CTF is a good exercise ;)

(to train that specific, junior variants are also a good option – e.g. 34C3 junior ctf)

...

These channels and trainings were both my practice and source of knowledge.  
They serve as reference and recommendation by heart.



# Congratulations – you made it to the end!

I hope you also had a lot of fun popping shells!

If you have any questions you can reach me here:

E-Mail: [chiliz0x10@gmail.com](mailto:chiliz0x10@gmail.com)

Twitter: [@chiliz16](https://twitter.com/chiliz16)

