

Introduction to ROP

chiliz

Download Workshop-VM: <https://rop.chiliz.tech/download-vm>

Register CTFd: <https://rop.chiliz.tech>

whoami



- Lisa / chiliz
- Student in Automation and Mechatronics
- Bachelor Thesis about Automated Security Testing & Fuzzing
- Blackhoodie Attendee and Trainer
- CTF Player

Agenda

- Recap Buffer Overflow
- What is ROP and why do we want it?
- Demo 64 Bit – simple ROP-chain
- 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:
    ;rax holds pointer
    ;to argv[1]
    mov rdi, rax
    call vuln(char*)
    ...
```

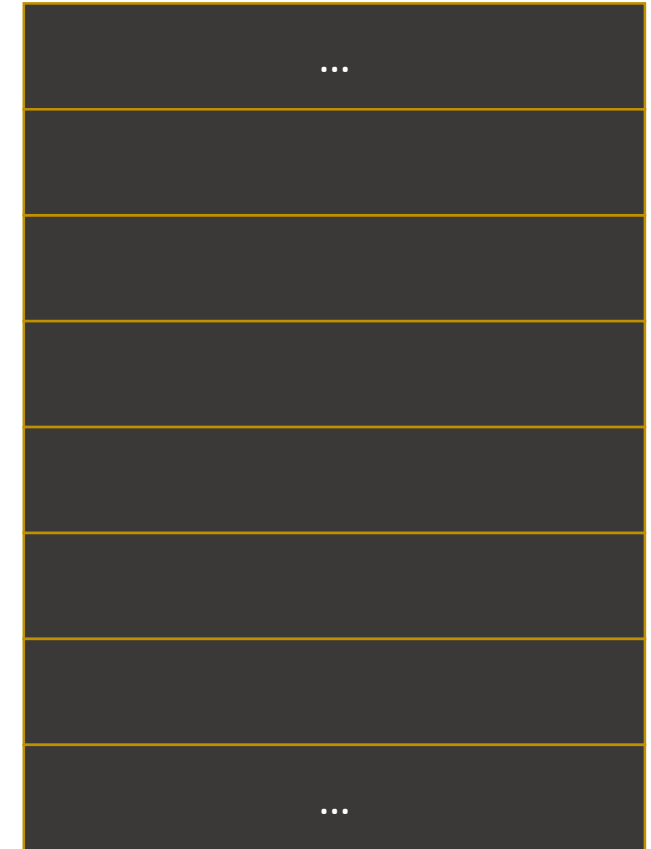
RDI



RSI



0x0000...



0x7FFFFFFF...



64 Bit – Calling convention Linux

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

Important registers:

- RIP: Instruction Pointer
- RSP: Current Top of the Stack



64 Bit – Calling convention Linux

- Move 2nd function argument in **RSI**
- Move 1st function argument in **RDI**
- Call to function
 - save **return address** on the stack to return to it later
 - Function gets executed
 - return to the address that is saved to the stack
- Execution continues, return value of function in **RAX**



Recap Buffer Overflow

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

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

```
main:
    ;rax holds pointer
    ;to argv[1]
    mov rdi, rax
    call vuln(char*)
    ...
```

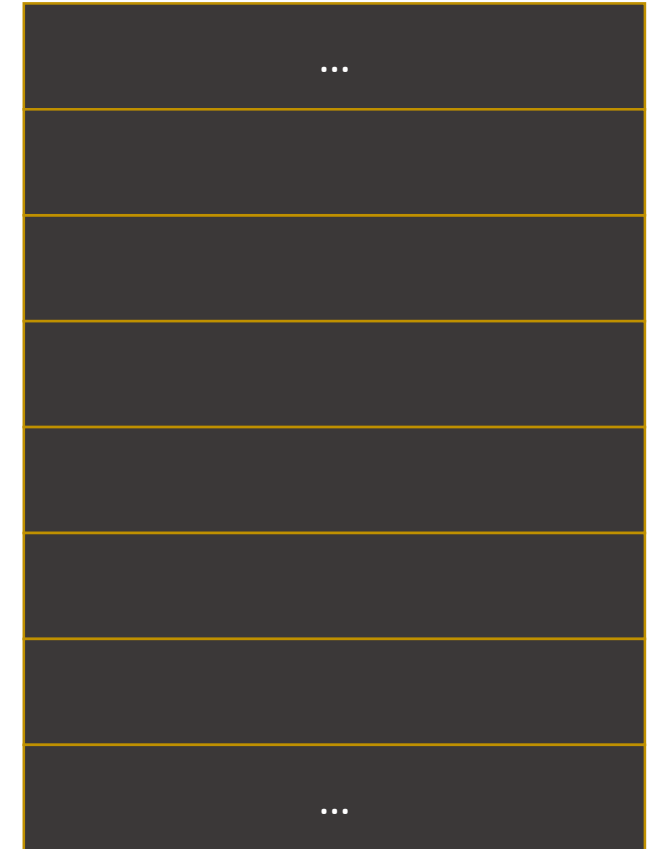
RDI



RSI



0x0000...



0x7FFFFFFF...

Recap Buffer Overflow

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

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

```
main:
    ;rax holds pointer
    ;to argv[1]
    mov rdi, rax
    call vuln(char*)
    ...
```

← RIP

RDI

ptr to argv[1]

RSI

0x0000...

RSP
(Top of Stack)

0x7FFFFFFF...



Recap Buffer Overflow

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

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

```
main:
    ;rax holds pointer
    ;to argv[1]
    mov rdi, rax
    call vuln(char*)
    ...
```

← RIP

0x0000...

RSP
(Top of Stack)

Saved RIP

RDI

ptr to argv[1]

RSI

0x7FFFFFFF...



Recap Buffer Overflow

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

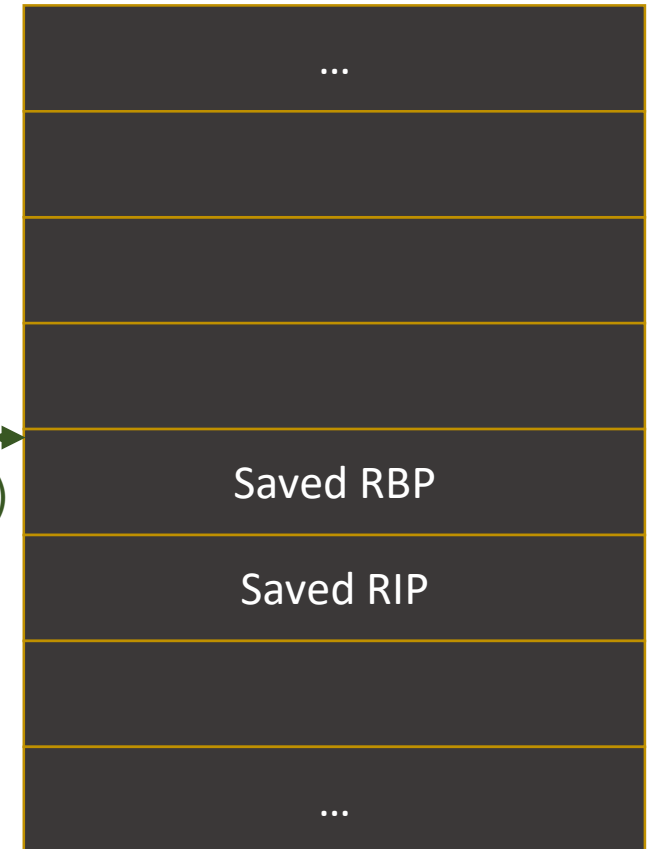
```
int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

←RIP

RSP →
(Top of Stack)

0x0000...



RDI

ptr to input (argv[1])

RSI

Recap Buffer Overflow

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

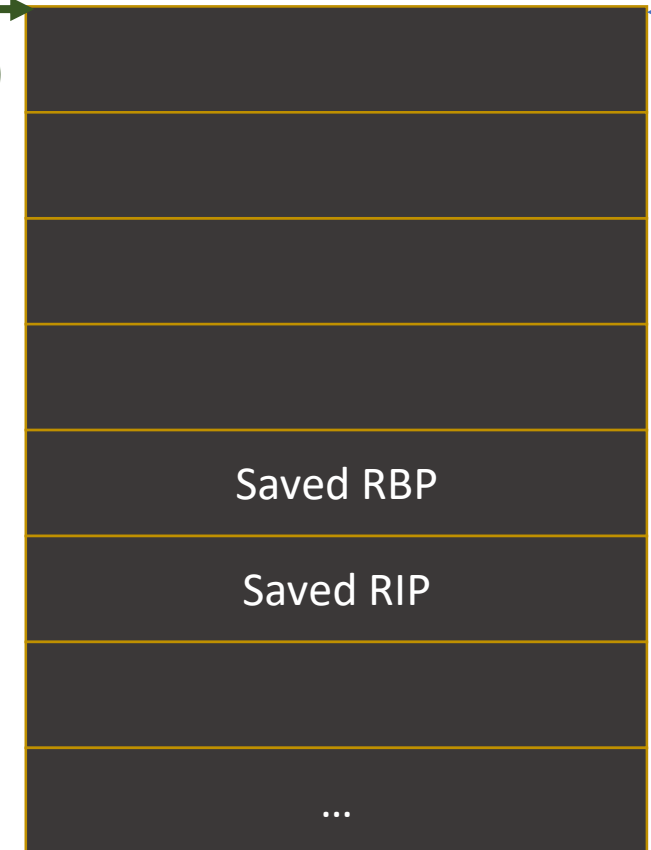
```
int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

0x0000...
RSP
(Top of Stack)

← RIP

← buffer



RDI

ptr to input (argv[1])

RSI

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 rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input ← RIP
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

RDI

ptr to input (argv[1])

RSI

ptr to input

0x0000...
RSP
(Top of Stack)

← buffer

Saved RBP

Saved RIP

...

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 rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer ← RIP
    call strcpy
    leave
    ret
```

RDI

ptr to buffer

RSI

ptr to input

0x0000...
RSP
(Top of Stack)

← buffer

Saved RBP

Saved RIP

...

0x7FFFFFFF...



Recap Buffer Overflow

Program call:

> ./myprogram AAA... (31*A)

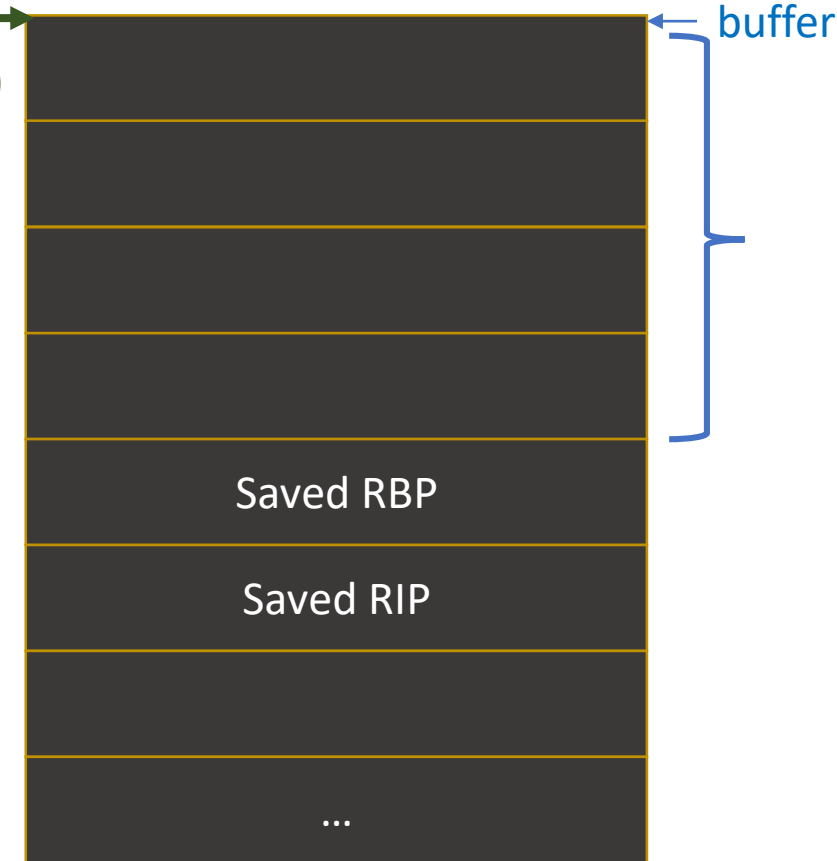
```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}
```

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

← RIP

0x0000...
RSP
(Top of Stack)



RDI

ptr to buffer

RSI

ptr to input



Recap Buffer Overflow

Program call:

> ./myprogram AAA... (31*A)

No overflow, we have 32 Bytes and write 32 Bytes

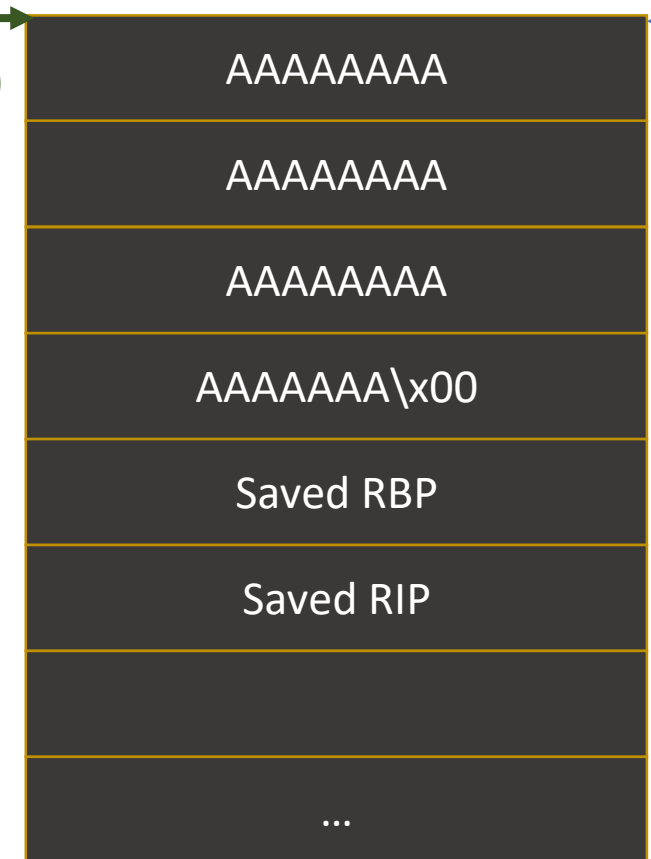
```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}
```

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

← RIP

0x0000...
RSP
(Top of Stack)



← buffer

RDI

ptr to buffer

RSI

ptr to input

0x7FFFFFFF...



Recap Buffer Overflow

Program call:

> ./myprogram AAA... (56*A)

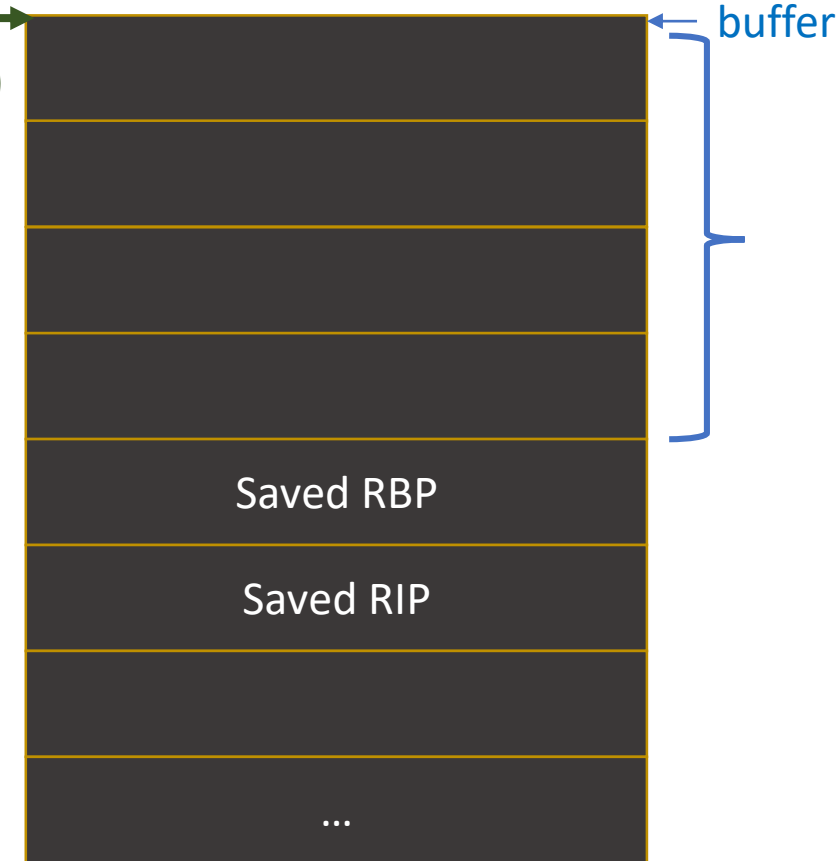
```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}
```

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

← RIP

0x0000...
RSP
(Top of Stack)



RDI

ptr to buffer

RSI

ptr to input

0x7FFFFFFF...



Recap Buffer Overflow

Program call:

> ./myprogram AAA... (56*A)

! Buffer Overflow, we have 32 bytes and write 56 bytes

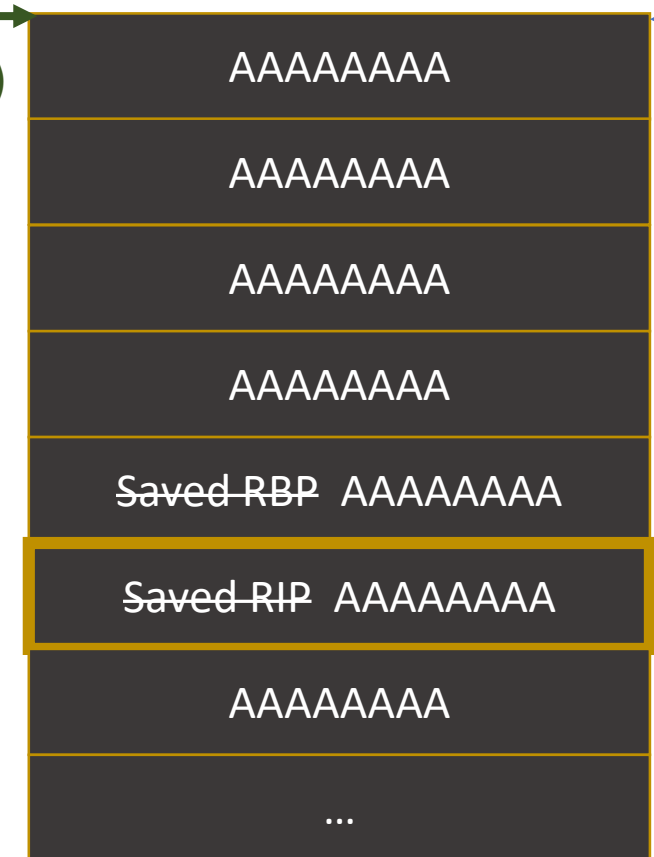
```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}
```

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

← RIP

0x0000...
RSP
(Top of Stack)



← buffer

RDI

ptr to buffer

RSI

ptr to input

0x7FFFFFFF...



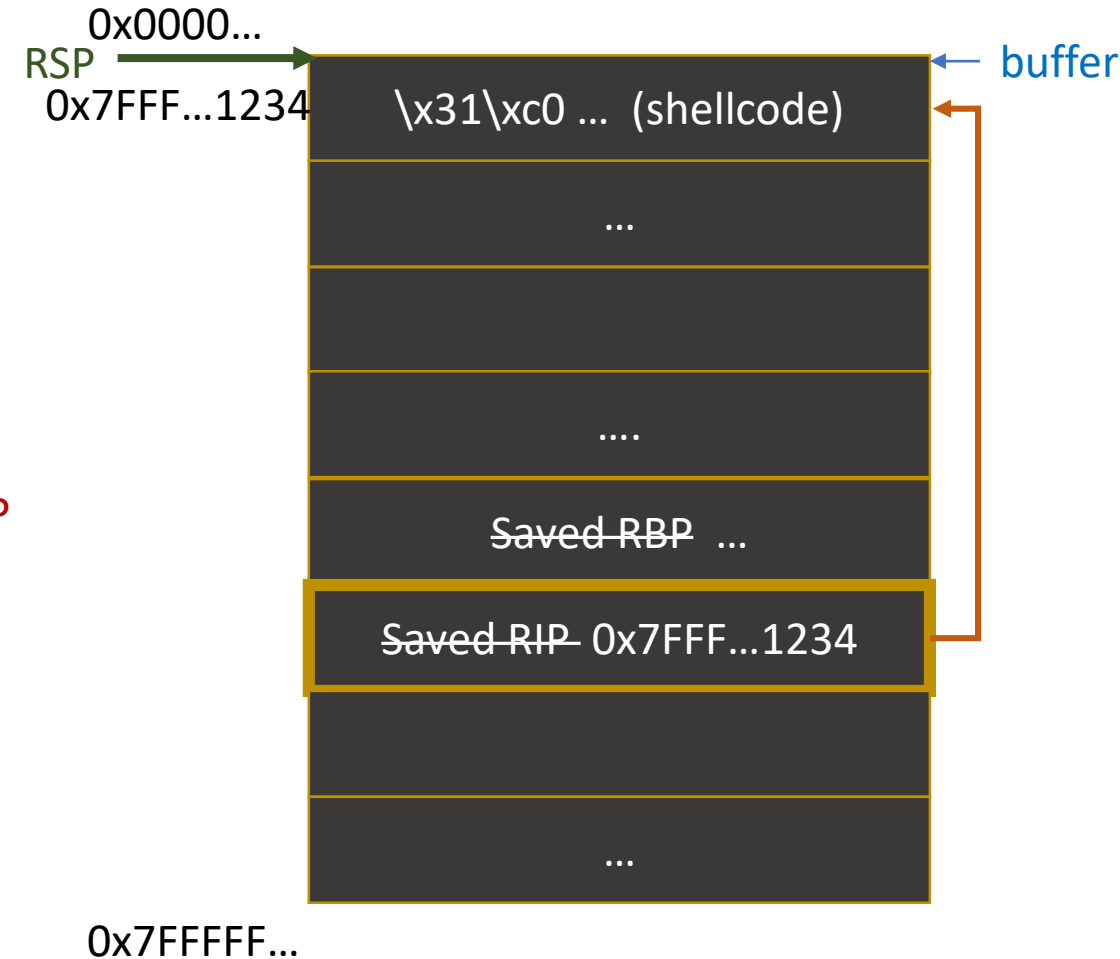
Recap Buffer Overflow

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

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

← RIP



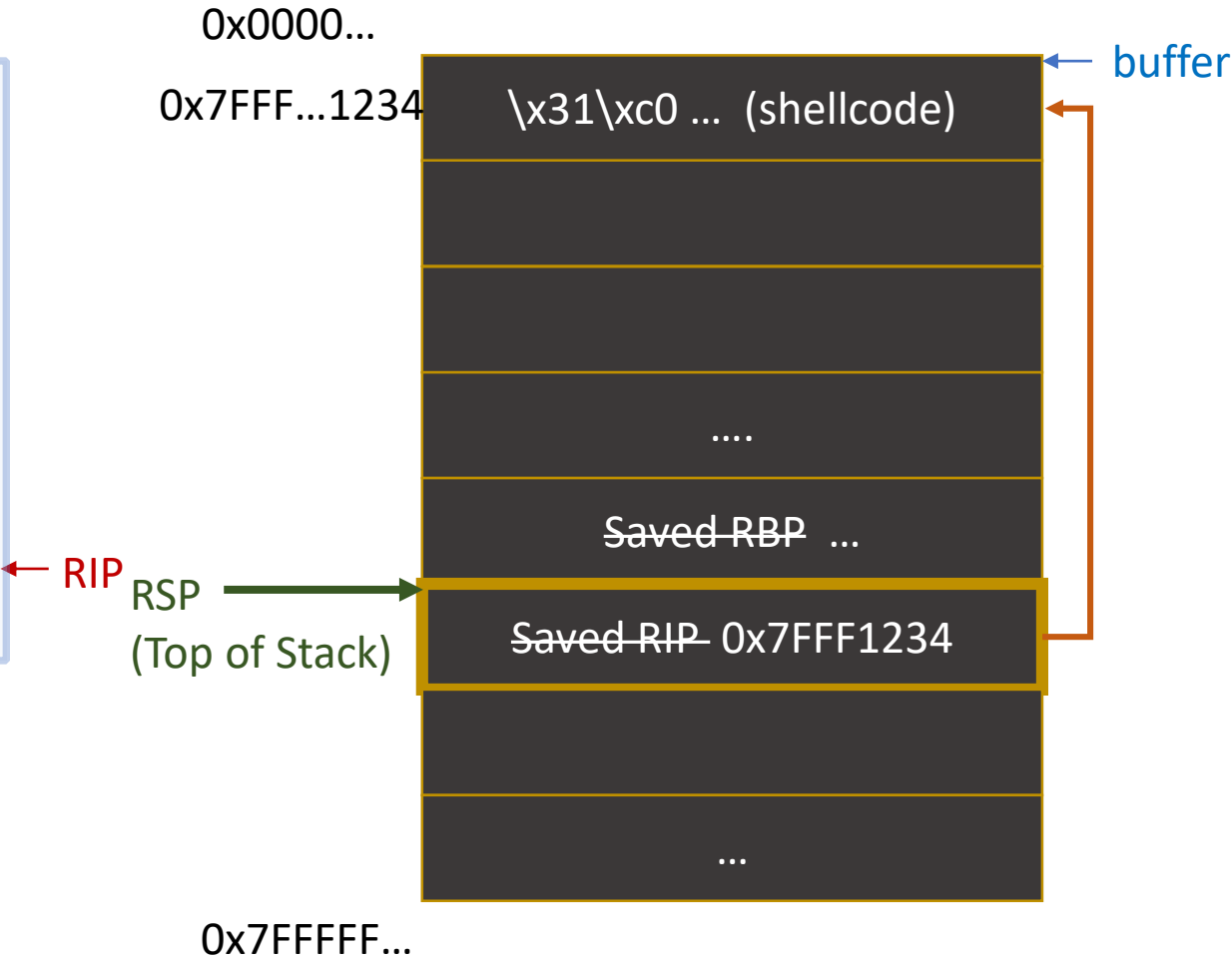
Recap Buffer Overflow

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

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

Leave:	Ret:
mov rsp, rbp	"pop rip"
pop rbp	



Recap Buffer Overflow

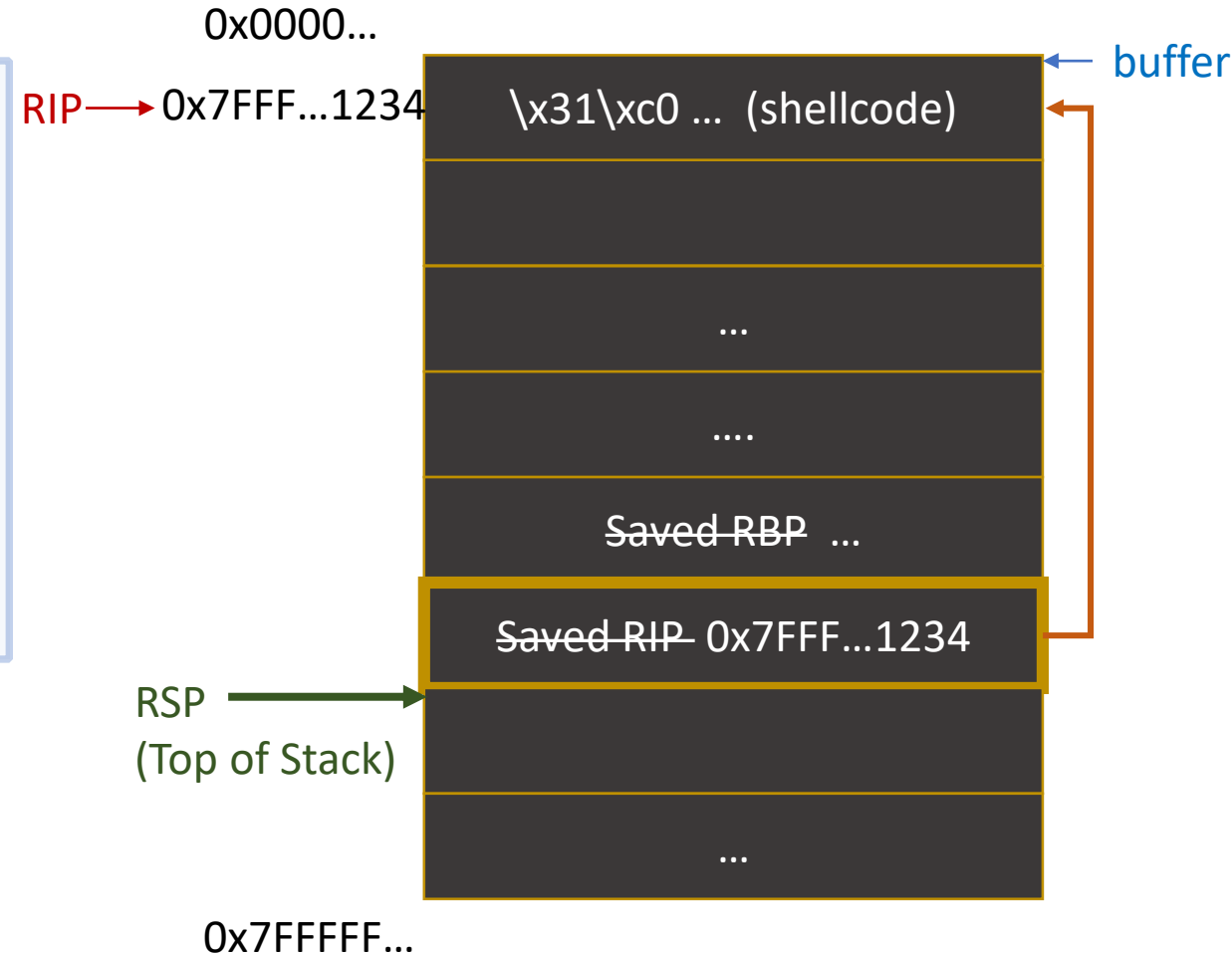
```
void vuln(char *input)
{
    char buffer[32];
    strcpy(buffer, input);
}
```

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

Leave:
mov rsp, rbp
pop rbp

Ret:
"pop rip"



Patterns - RIP control

How to set RIP to an **exact** value?

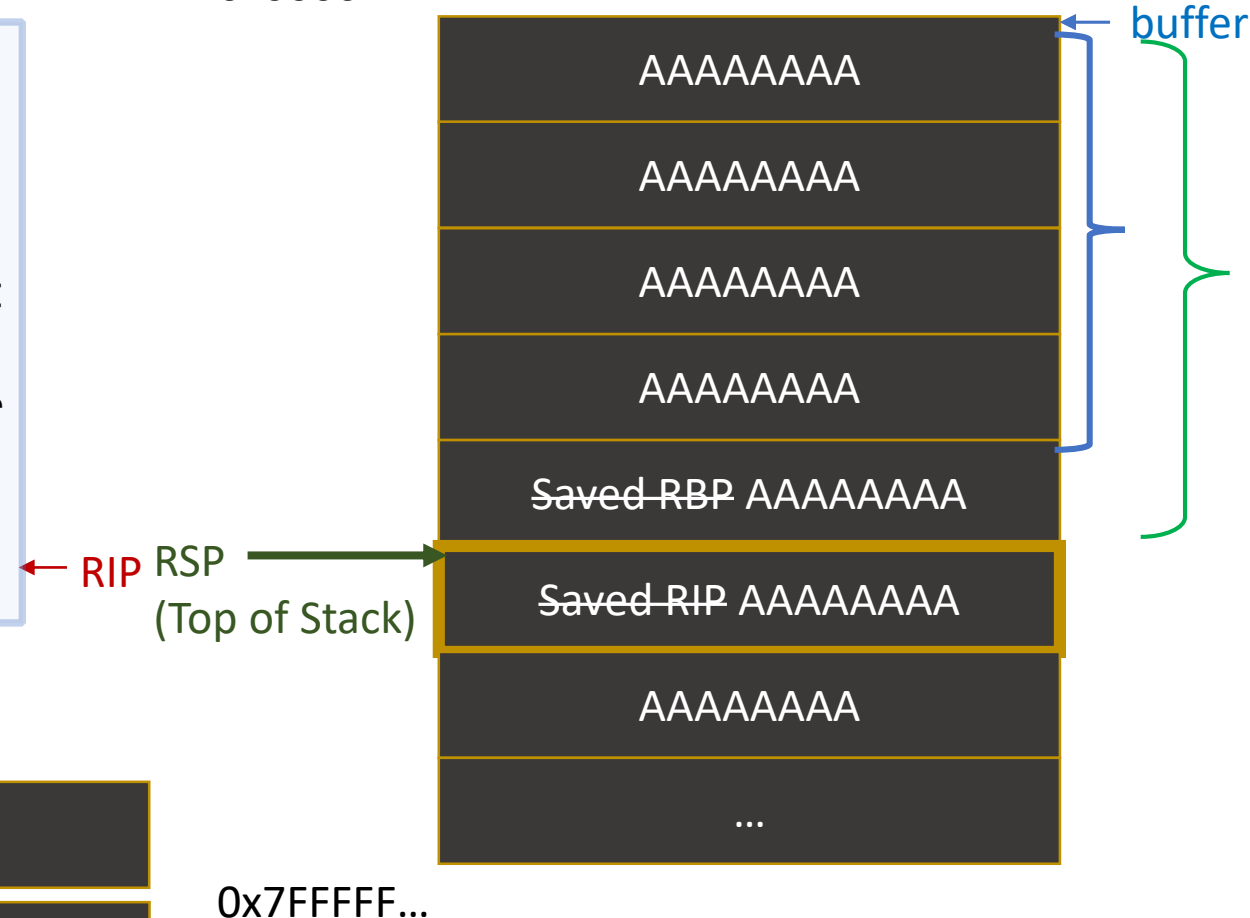
How many A's until we reach the saved RIP?

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

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

0x0000...



RDI

ptr to buffer

RSI

ptr to input



Patterns - RIP control

Program call:

> ./myprogram AAAAAAAAAABBBBBBBBBBCCCCCCCC...

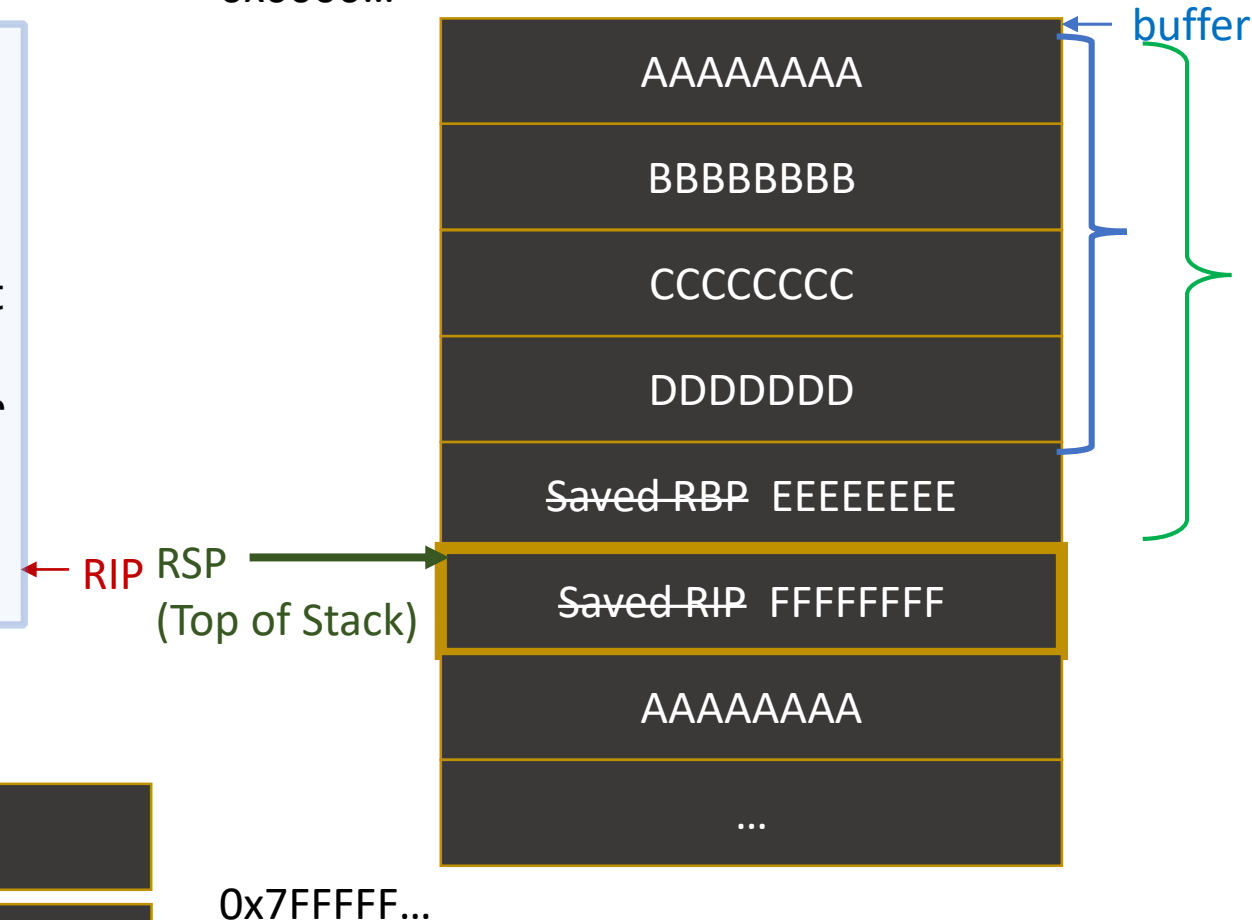
! Buffer Overflow, RIP will be FFFFFFFF

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

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

0x0000...



RDI

ptr to buffer

RSI

ptr to input

0x7FFFFFFF...



Patterns - RIP control

Program call:

> ./myprogram AAAAAAAAAABBBBBBBBCCCCCCCC...

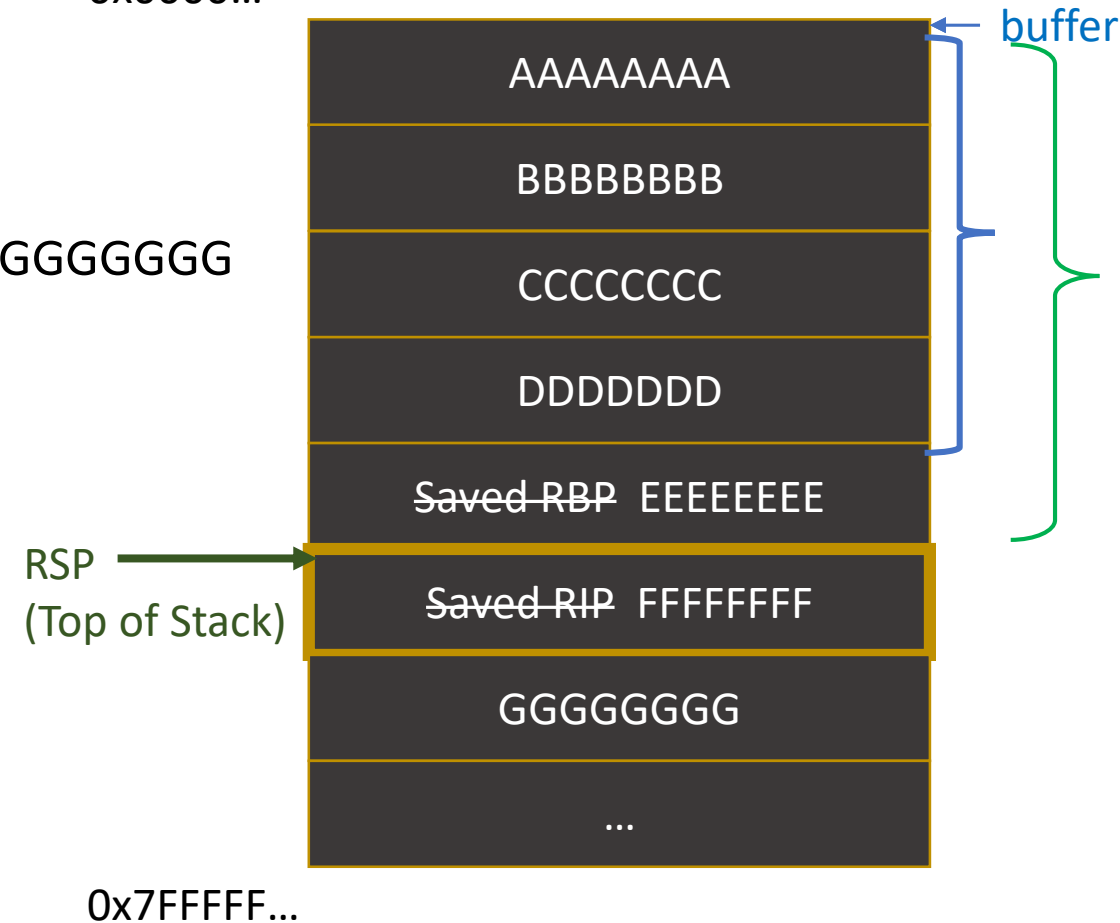
! Buffer Overflow, RIP will be FFFFFFFF

0x0000...

How many bytes do we have to write until we reach RIP?

Input – String:

AAAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDDEEEEEEEEFFFFFFFFFGGGGGGGG



Patterns - RIP control

Program call:

```
> ./myprogram AAAAAAAAAABBBBBBBBCCCCCCCC...
```

! Buffer Overflow, RIP is now FFFFFFFF

How many bytes do we have to write until we reach RIP?

Input – String:

AAAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDDEEEEEEEFFFFFFFGGGGGGG

Fill-buff

RIP

$\text{Fill-buff} = 8 * "A" + 8 * "B" + 8 * "C" + 8 * "D" + 8 * "E"$
= 40 Bytes

RSP
(Top of Stack)

0x0000...



0x7FFFFF...



Patterns - RIP control

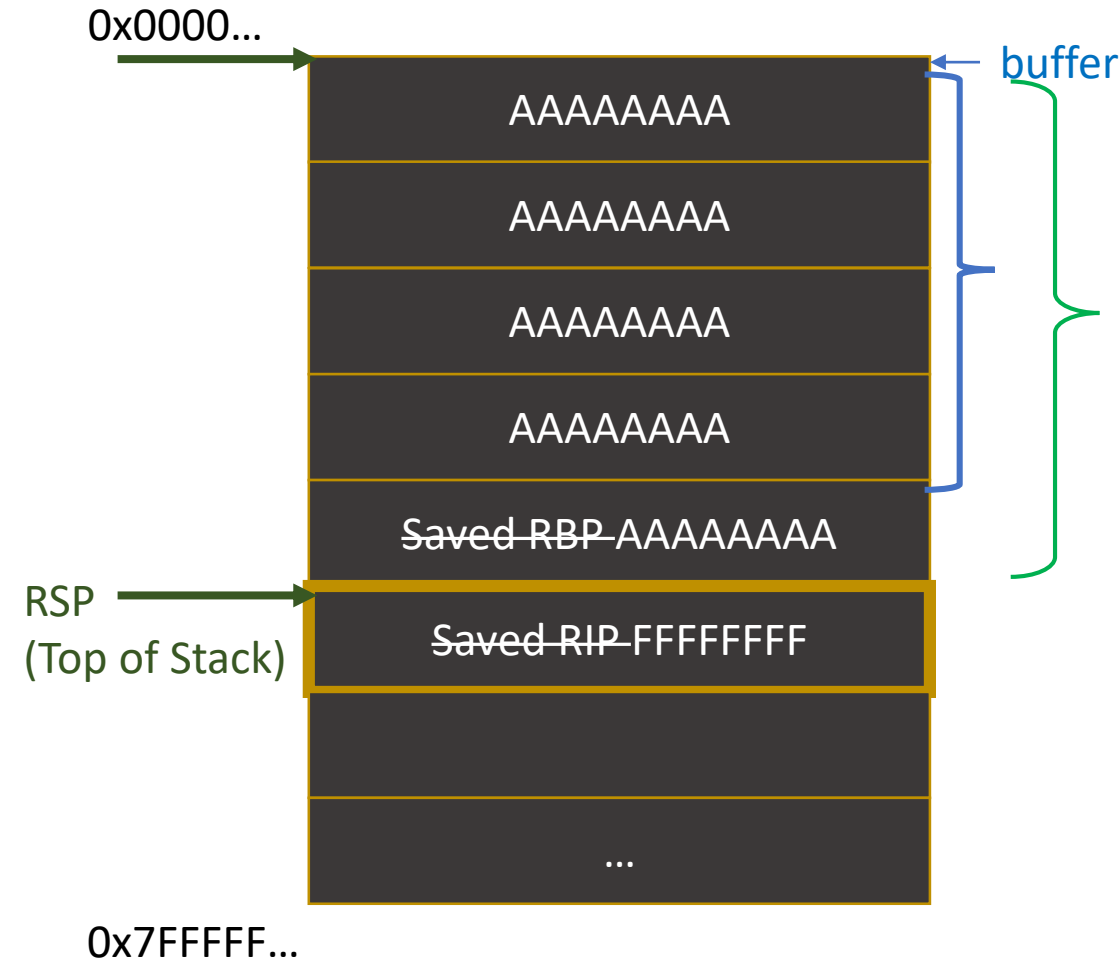
How many bytes do we have to write until we reach RIP?

Input – String = 40*"A" + 8*"F"

Program call:

```
> ./myprogram AA...(A*40)FFFFFFFF
```

! Buffer Overflow, RIP is now FFFFFFFF



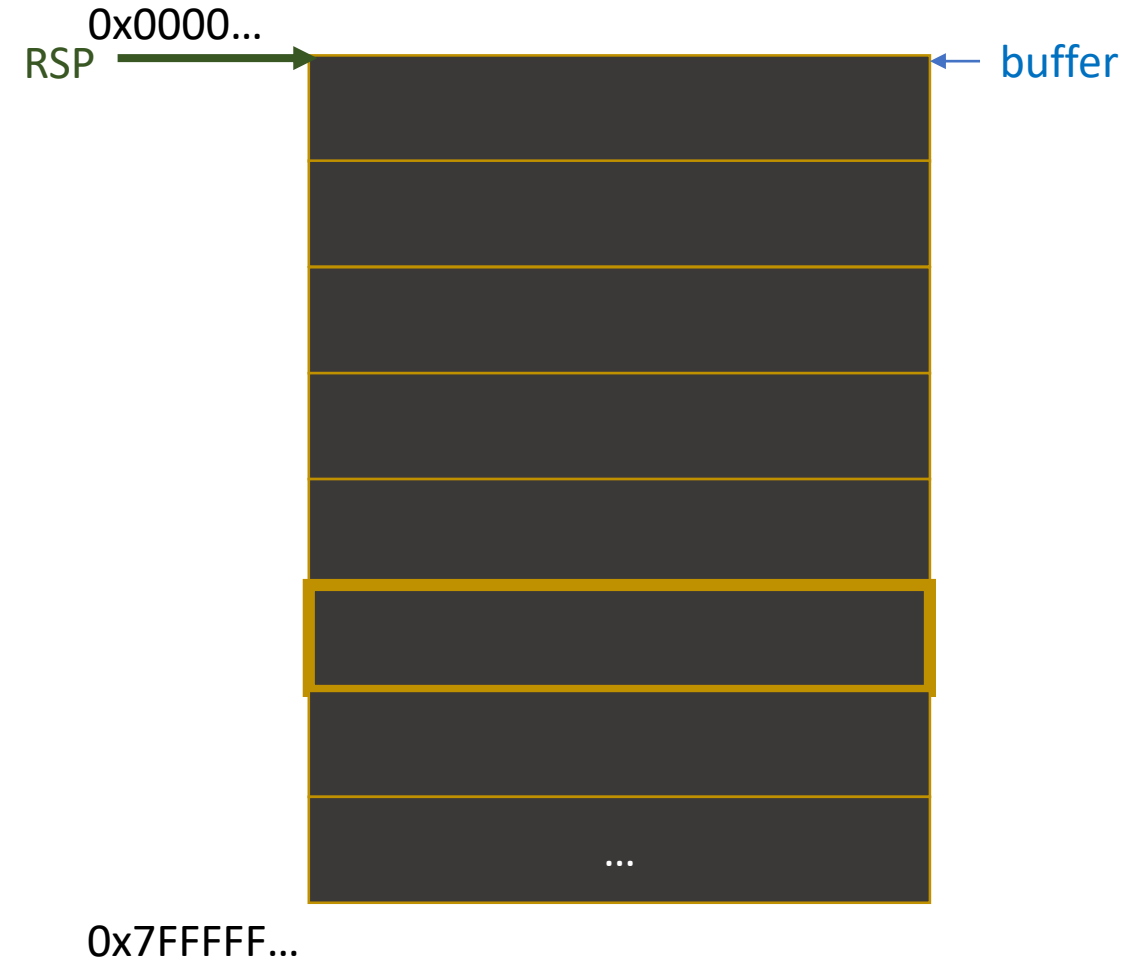
Cyclic Patterns

1. Generate Pattern

`gdb-peda$ pattern create 60`

`AAA%AAsAABAA$AAAnAACAA-AA..`

- pwntools: `cyclic(60)`
- Gdb-peda: `pattern create 60`



Cyclic Patterns

1. Generate Pattern

`gdb-peda$ pattern create 60`

`AAA%AAsAABAA$AAAnAACAA-AA`

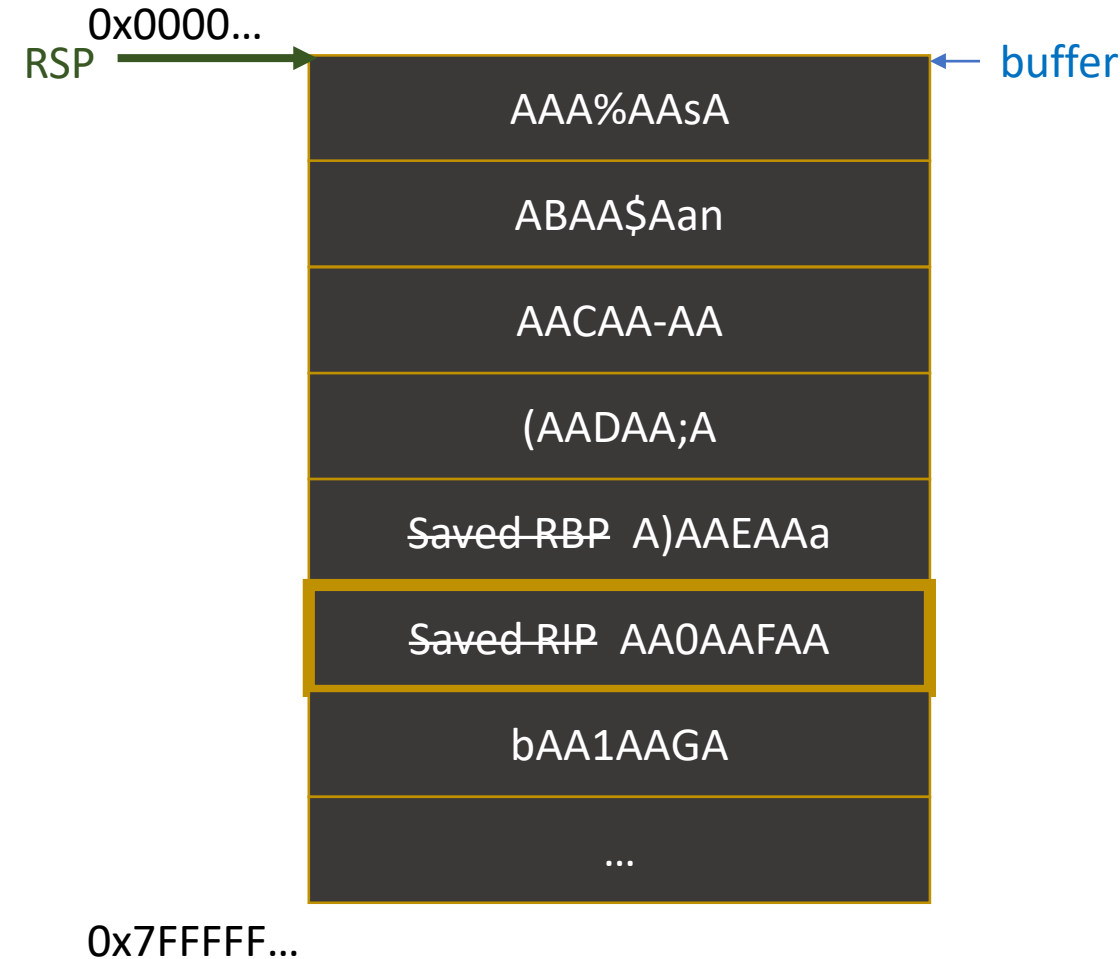
2. Pattern as input for the program

`gdb-peda$ run`

Give me the Code:

`AAA%AAsAABAA$AAAnAACAA-AA`

- pwntools: (metasploit cyclic pattern): `cyclic(60)`
- Gdb-peda: `pattern create 60`



Cyclic Patterns

1. Generate Pattern

```
gdb-peda$ pattern create 60
```

```
AAA%AAsAABAA$AAAnAACAA-AA
```

2. Pattern as input for the program

```
gdb-peda$ run
```

Give me the Code:

```
AAA%AAsAABAA$AAAnAACAA-AA
```

3. Find the part in the pattern that overwrote RIP

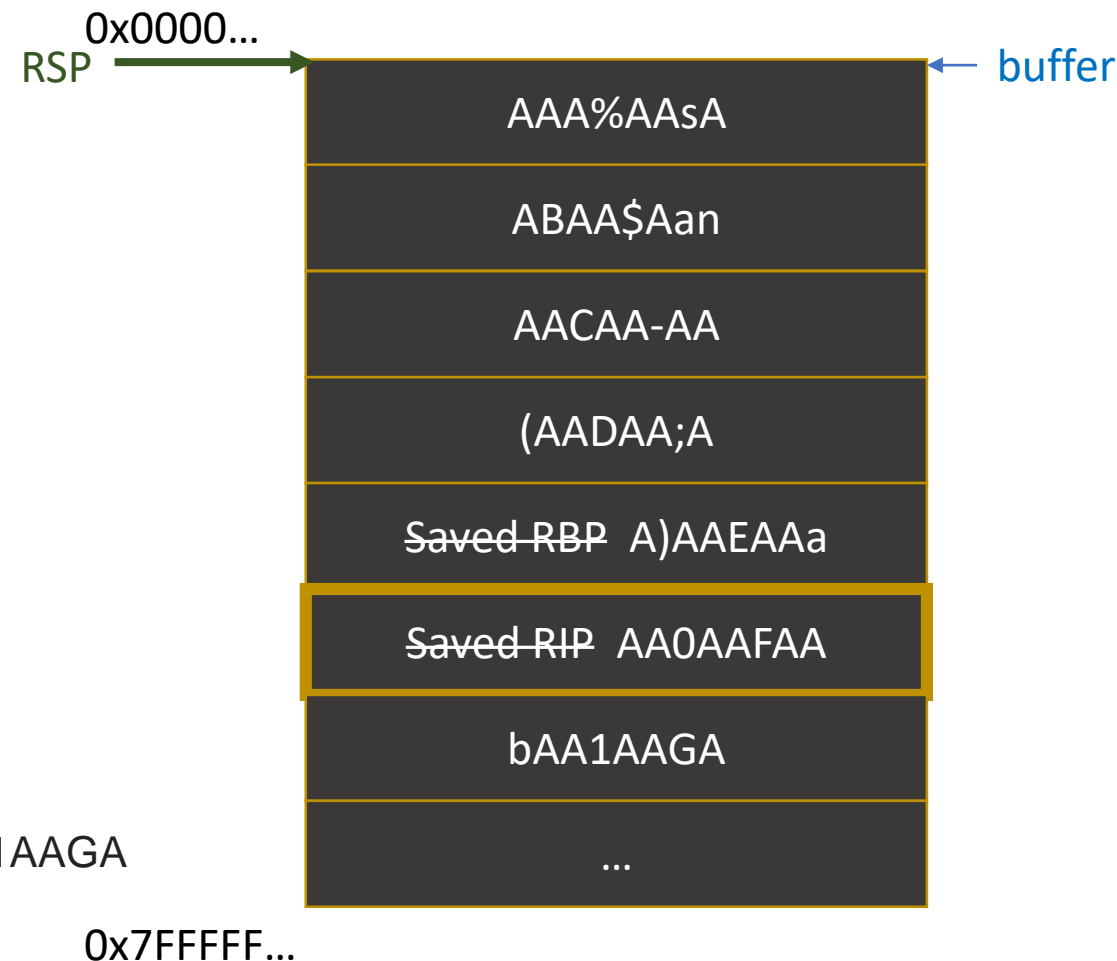
```
gdb-peda$ pattern offset AA0AAFAA
```

Gdb-peda internally uses pattern matching for that:

```
AAA%AAsAABAA$AAAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGA
```

```
AA0AAFAA
```

- pwntools: (metasploit cyclic pattern): cyclic(60)
- Gdb-peda: pattern create 60



Cyclic Patterns

1. Generate Pattern

```
gdb-peda$ pattern create 60
```

```
AAA%AAsAABAA$AAAnAACAA-AA
```

2. Pattern as input for the program

```
gdb-peda$ run
```

Give me the Code:

```
AAA%AAsAABAA$AAAnAACAA-AA
```

3. Find the part in the pattern that overwrote RIP

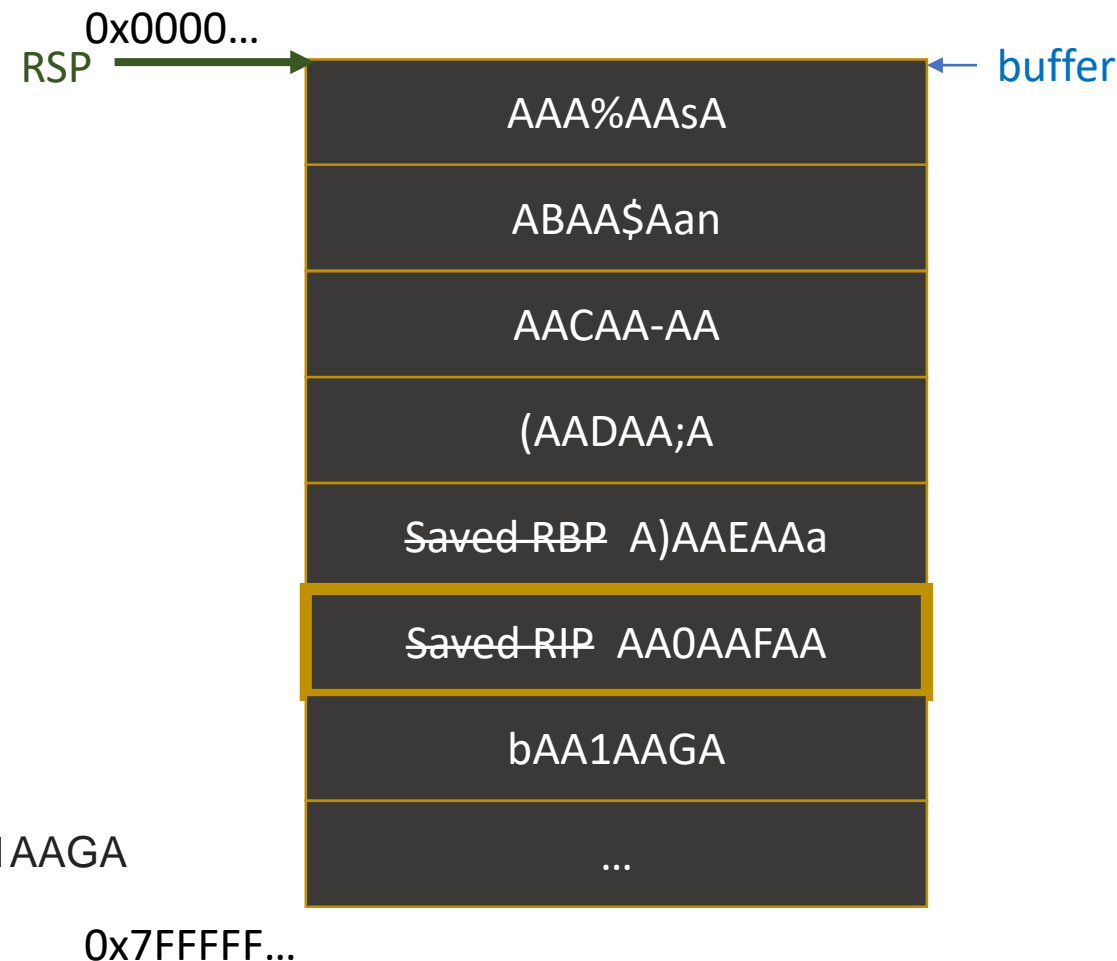
```
gdb-peda$ pattern offset AA0AAFAA
```

Gdb-peda internally uses pattern matching for that:

```
AAA%AAsAABAA$AAAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGA
```

→ **AA0AAFAA**

- pwntools: (metasploit cyclic pattern): cyclic(60)
- Gdb-peda: pattern create 60



Cyclic Patterns

1. Generate Pattern

`gdb-peda$ pattern create 60`

`AAA%AAsAABAA$AAAnAACAA-AA`

2. Pattern as input for the program

`gdb-peda$ run`

Give me the Code:

`AAA%AAsAABAA$AAAnAACAA-AA`

3. Find the part in the pattern that overwrote RIP

`gdb-peda$ pattern offset AA0AAF`

Gdb-peda internally uses pattern matching for that:

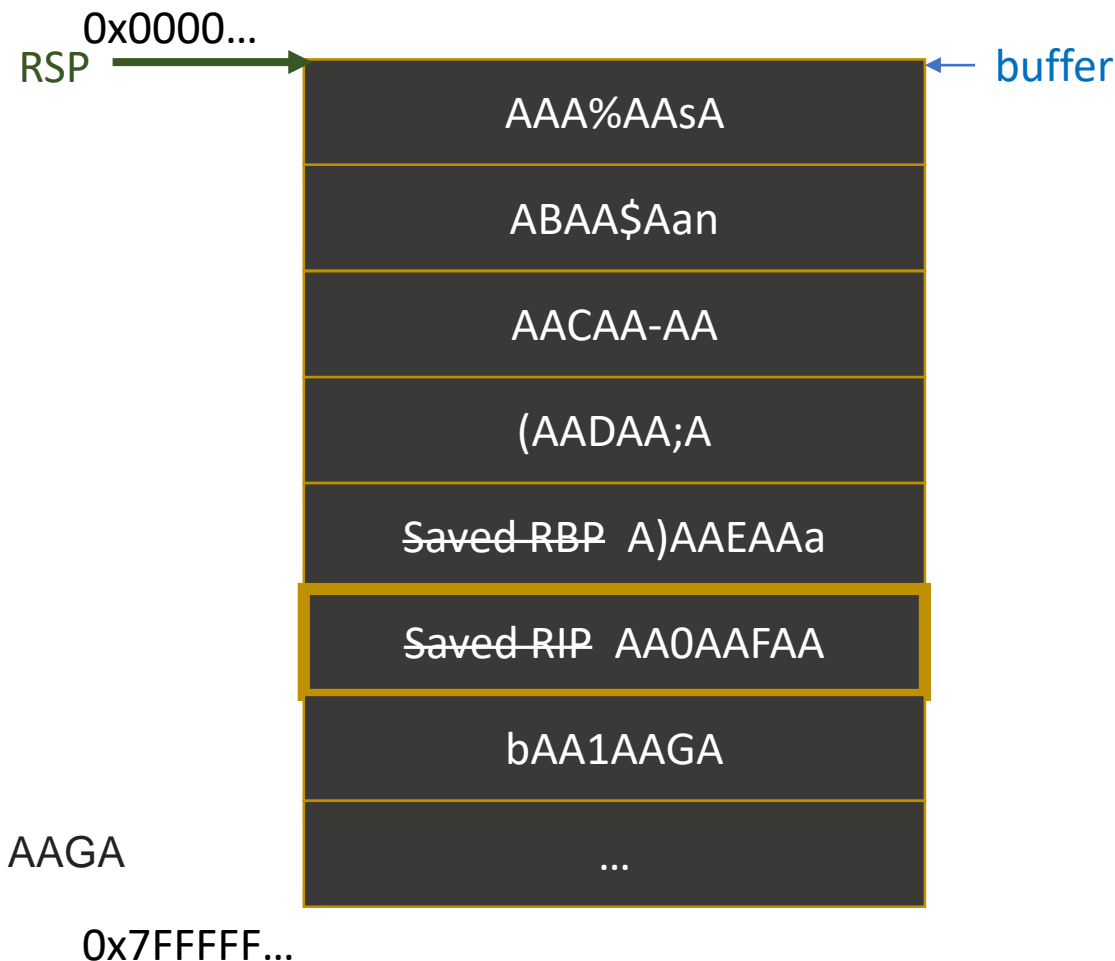
`AAA%AAsAABAA$AAAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFbAA1AAGA`

`AA0AAF`

Match!

Offset: 40

- pwntools: (metasploit cyclic pattern): `cyclic(60)`
- Gdb-peda: `pattern create 60`



Cyclic Patterns

- ## 1. Generate Pattern

```
gdb-peda$ pattern create 60
```

AAA%AAsAABAA\$AA nAACAA-AA

- ## 2. Pattern as input for the program

```
gdb-peda$ run
```

Give me the Code:

AAA%AAsAABAA\$AA nAACAA-AA

- ### 3. Find the part in the pattern that overwrote RIP

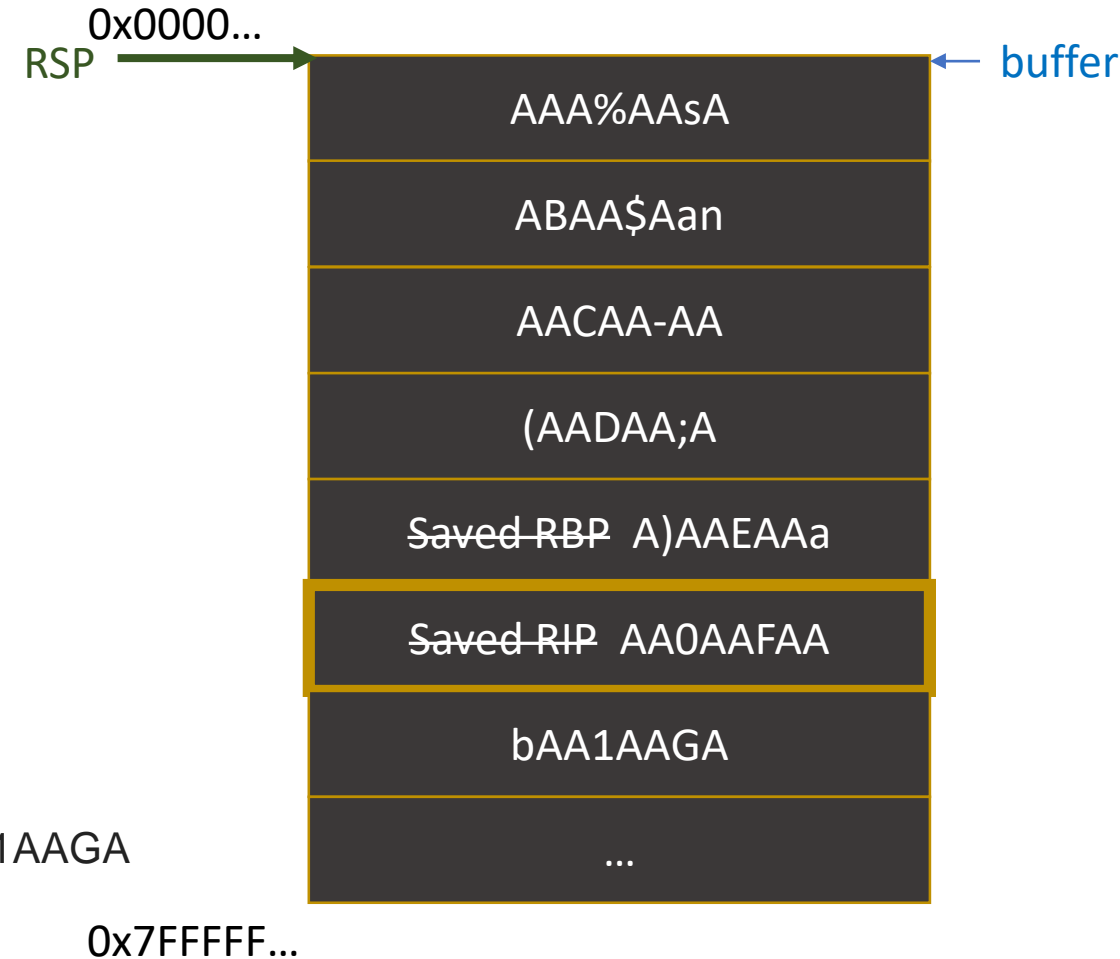
gdb-peda\$ pattern offset **AA0AAFAA**

Gdb-peda internally uses pattern matching for that:

AAA%AAsAABAA\$AA nAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGA

Match!
Offset: 40

- pwntools: (metasploit cyclic pattern): cyclic(60)
- Gdb-peda: pattern create 60



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)

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
```

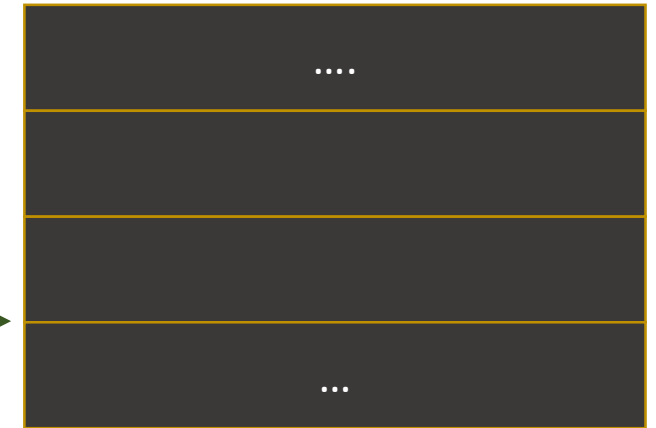
← RIP

0x0000...

RSP



0x7FFF...



RDI

ptr to „/bin/sh“

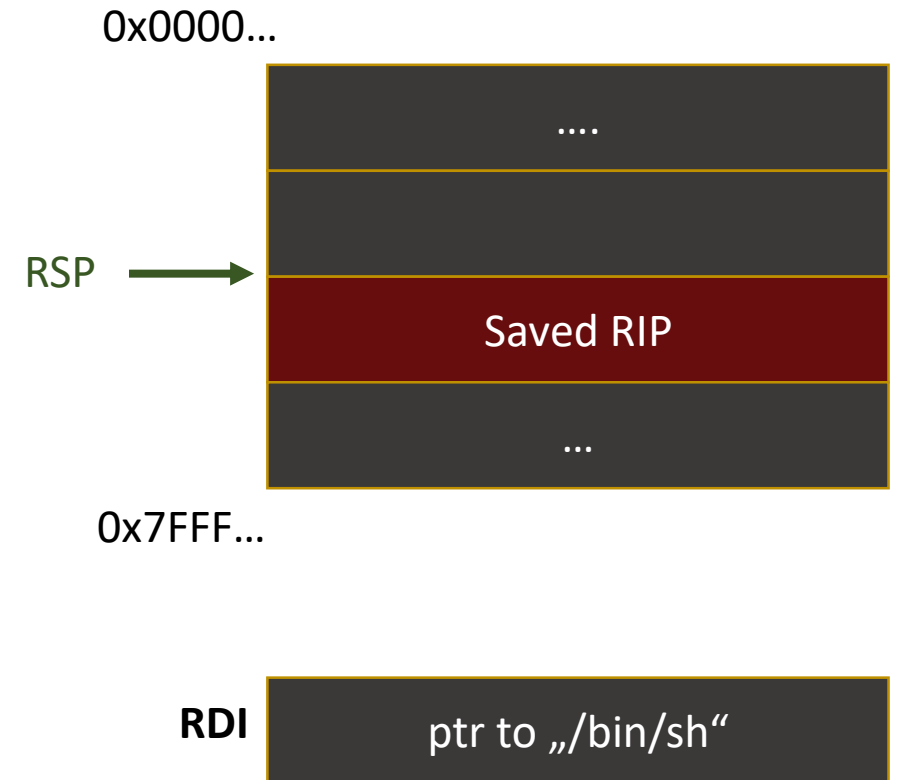


64 Bit – Calling convention

`system("/bin/sh")`

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

← RIP



Payload strategy

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

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

Leave:
mov rsp, rbp
pop rbp

Ret:
"pop rip"

RDI

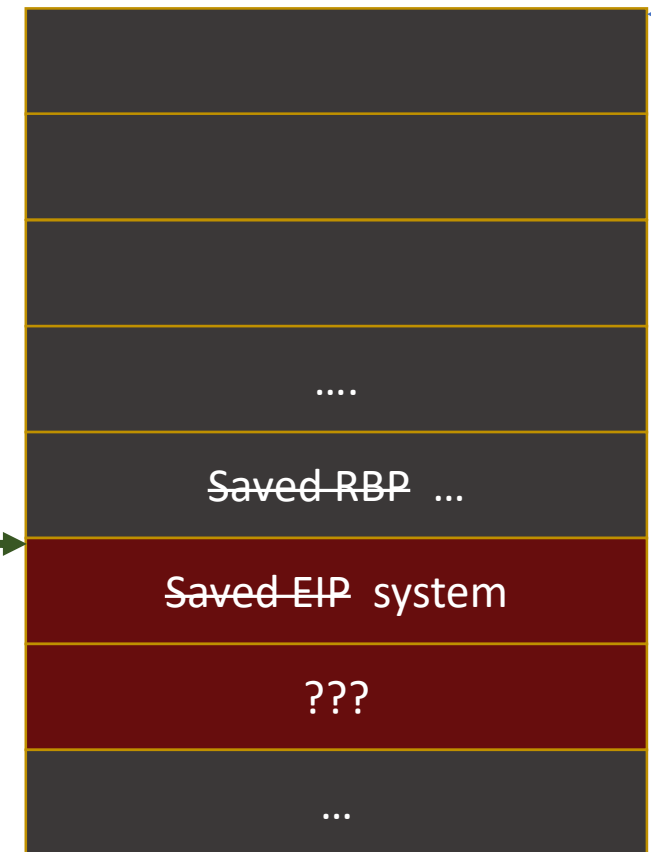
???

0x7FFFFFFF...

0x0000...

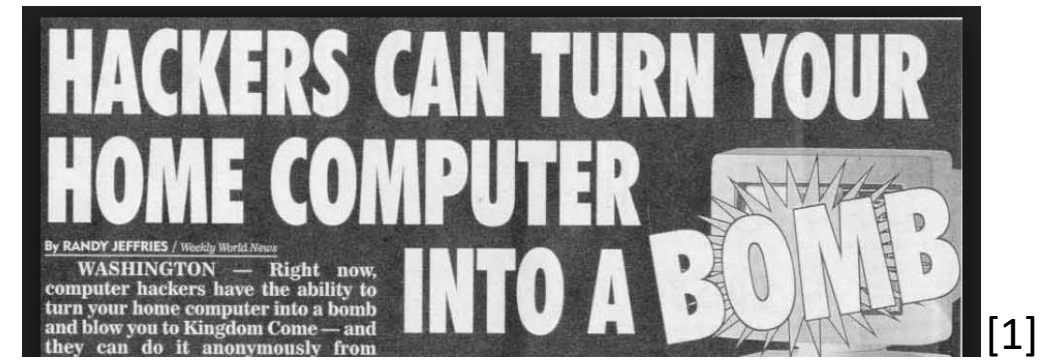
← RIP

RSP
(Top of Stack)



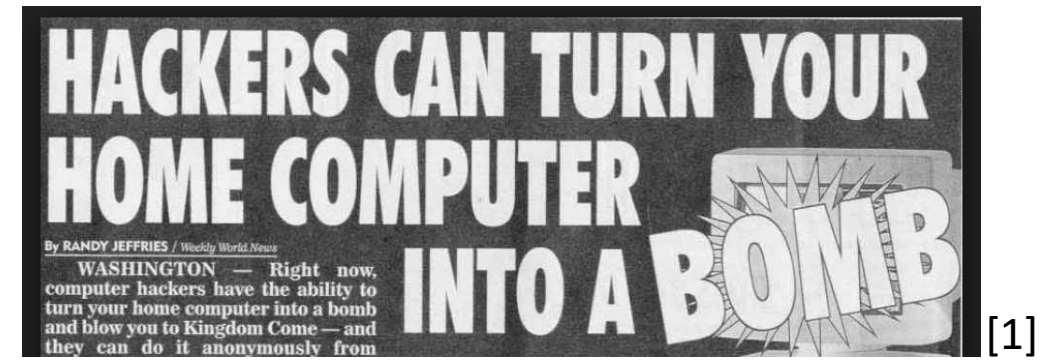
← buffer

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

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

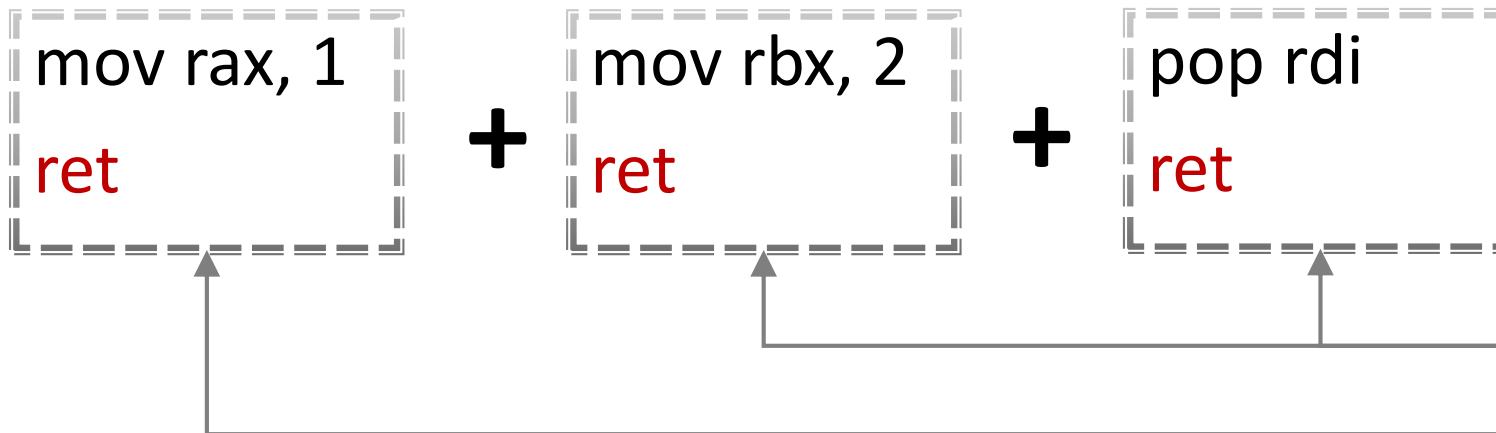
Building ROP chains...



[1]



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



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

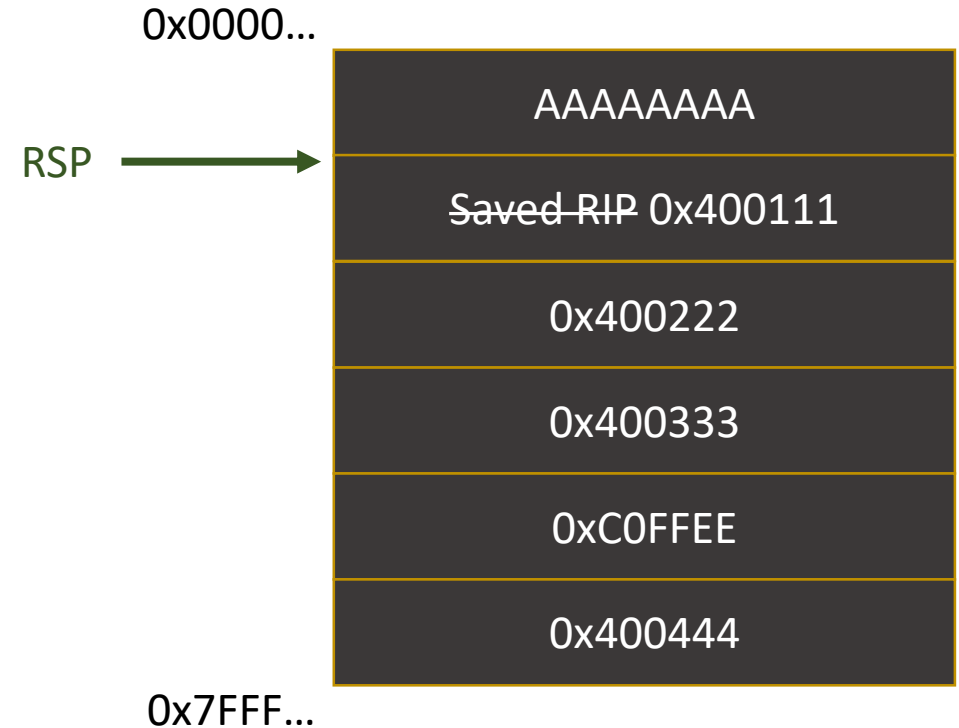
```
mov rax, 1
ret
```

0x400222

```
mov rbx, 2
ret
```

0x400333

```
pop rdi
ret
```



RAX



RBX



RDI



Building ROP-chains ...

0x400111

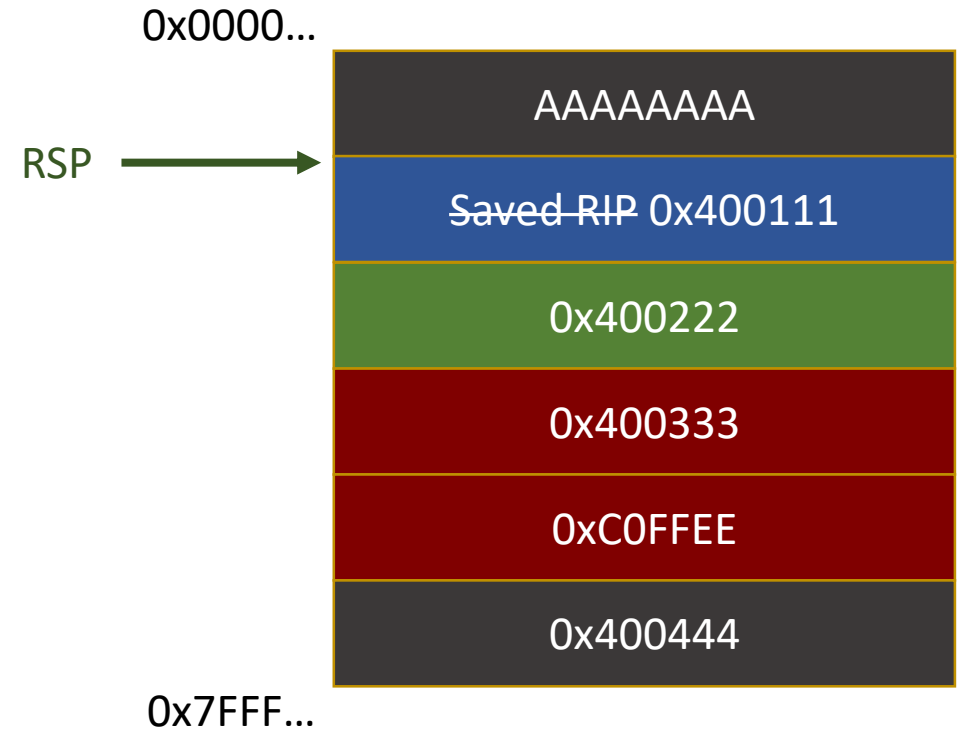
```
mov rax, 1
ret
```

0x400222

```
mov rbx, 2
ret
```

0x400333

```
pop rdi
ret
```



RAX

RBX

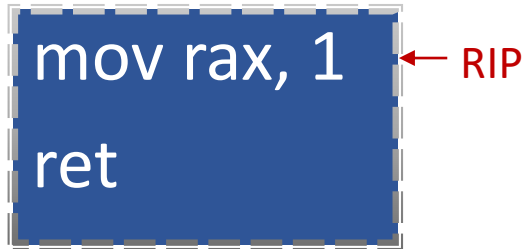
RDI



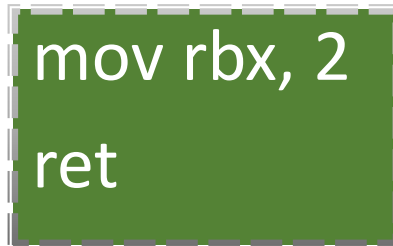
Building ROP-chains ...

(„ret“ = pop RIP)

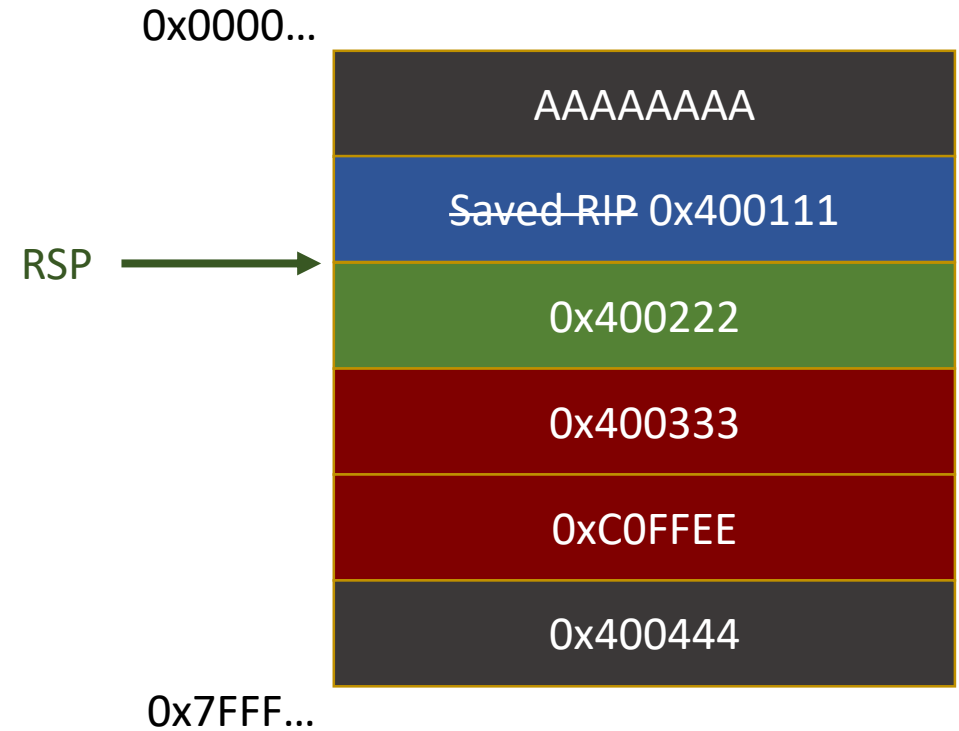
0x400111



0x400222



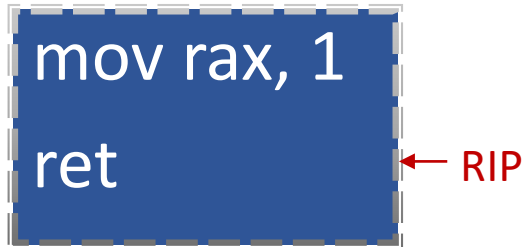
0x400333



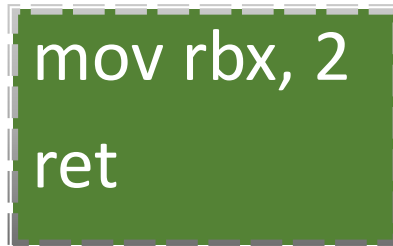
Building ROP-chains ...

(„ret“ = pop RIP)

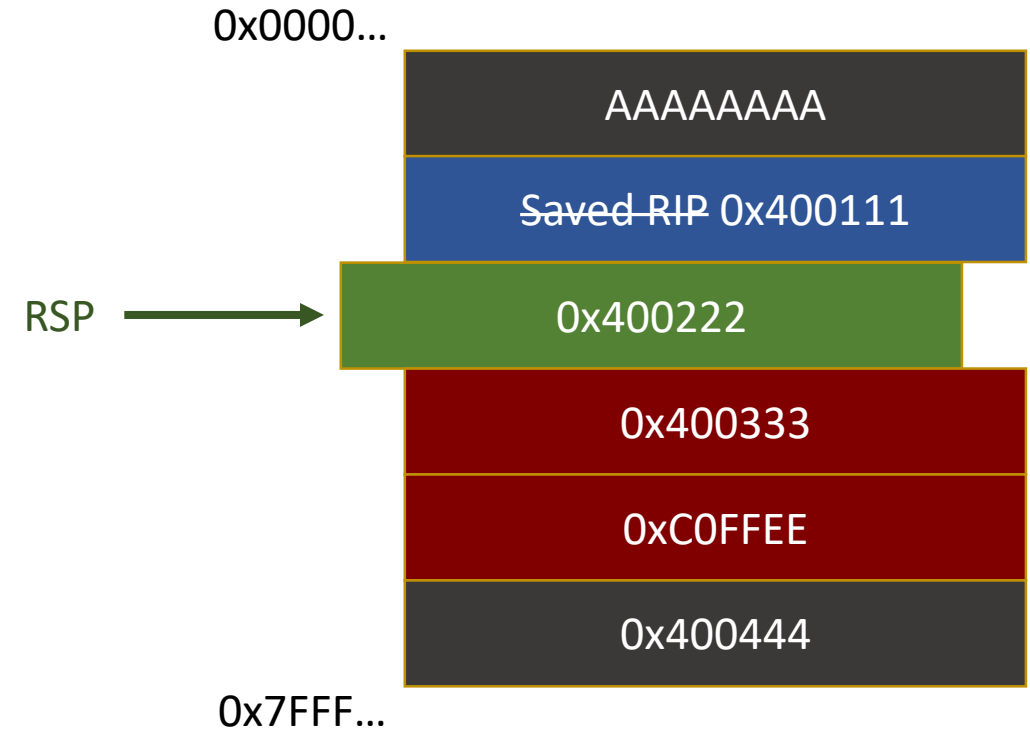
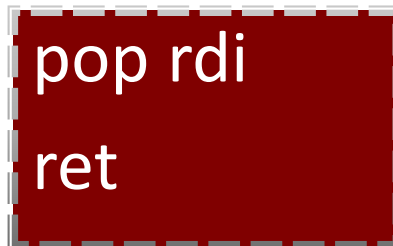
0x400111



0x400222



0x400333



Building ROP-chains ...

0x400111

```
mov rax, 1
ret
```

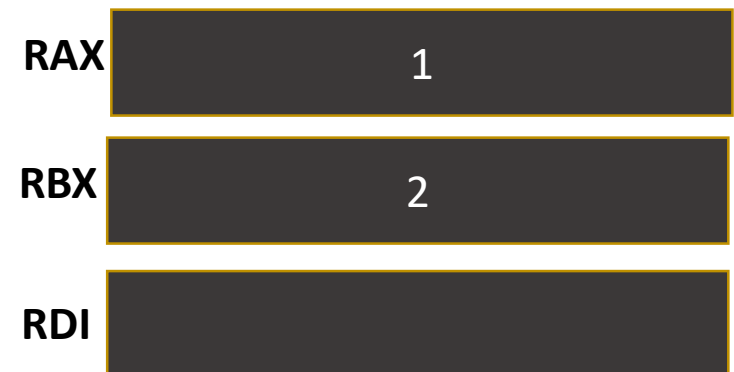
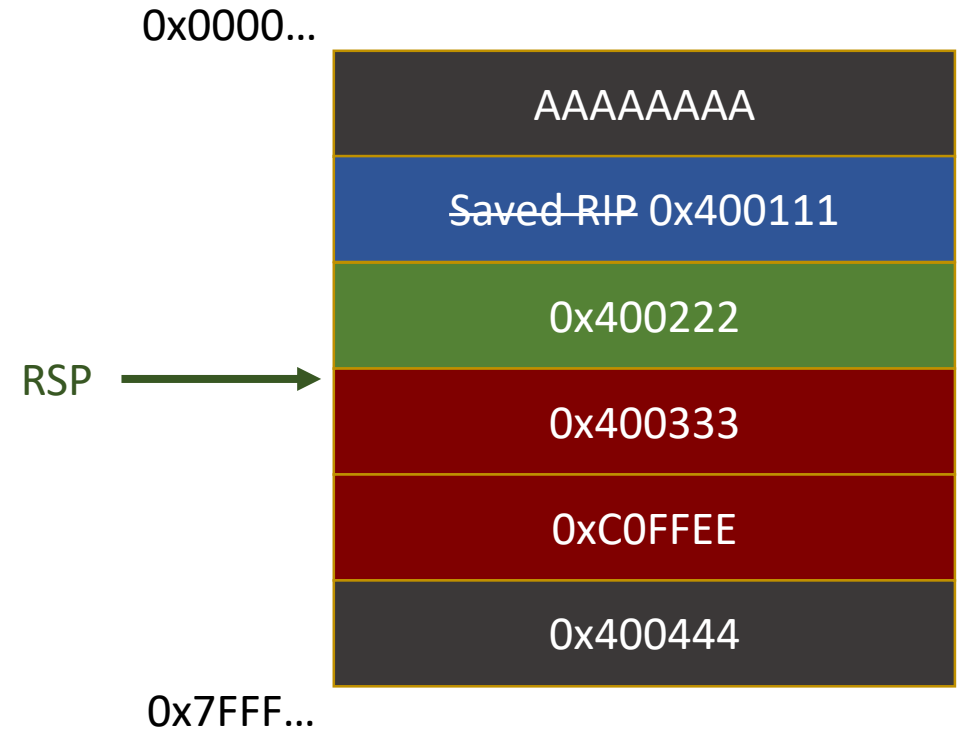
0x400222

```
mov rbx, 2
ret
```

← RIP

0x400333

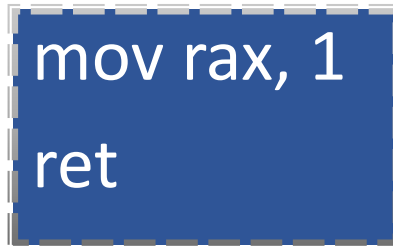
```
pop rdi
ret
```



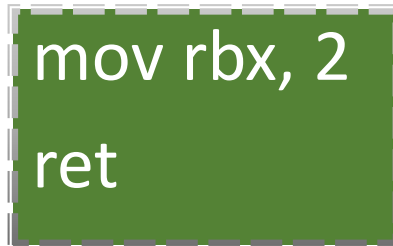
Building ROP-chains ...

(„ret“ = pop RIP)

0x400111

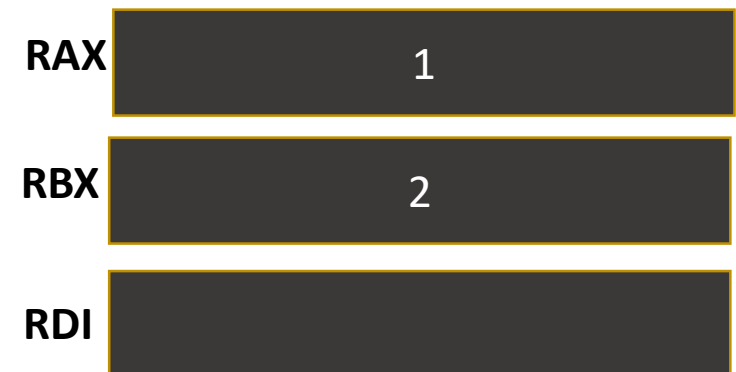
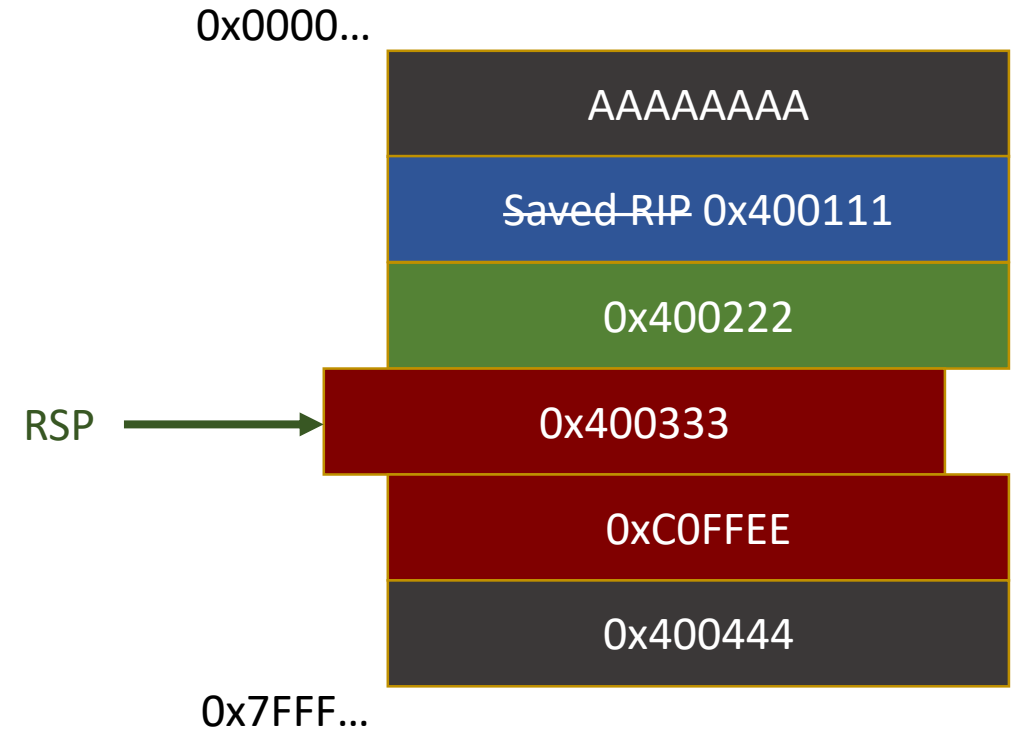


0x400222



← 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



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
```



Payload strategy

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

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

Leave:
mov rsp, rbp
pop rbp

Ret:
"pop rip"

RDI

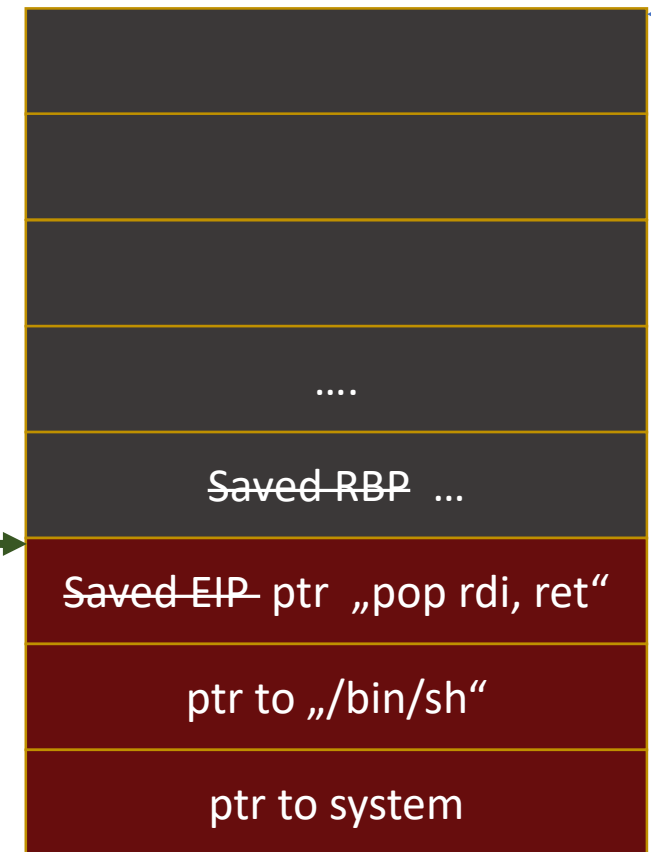
???

0x7FFFFFFF...

0x0000...

← RIP

RSP
(Top of Stack)



← buffer

Payload strategy

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

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

```
vuln(char*) :
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov rsi, rdi ; input
    lea rax, [rbp-32]
    mov rdi, rax ;buffer
    call strcpy
    leave
    ret
```

Leave: mov rsp, rbp pop rbp	Ret: "pop rip"
-----------------------------------	-------------------

RDI

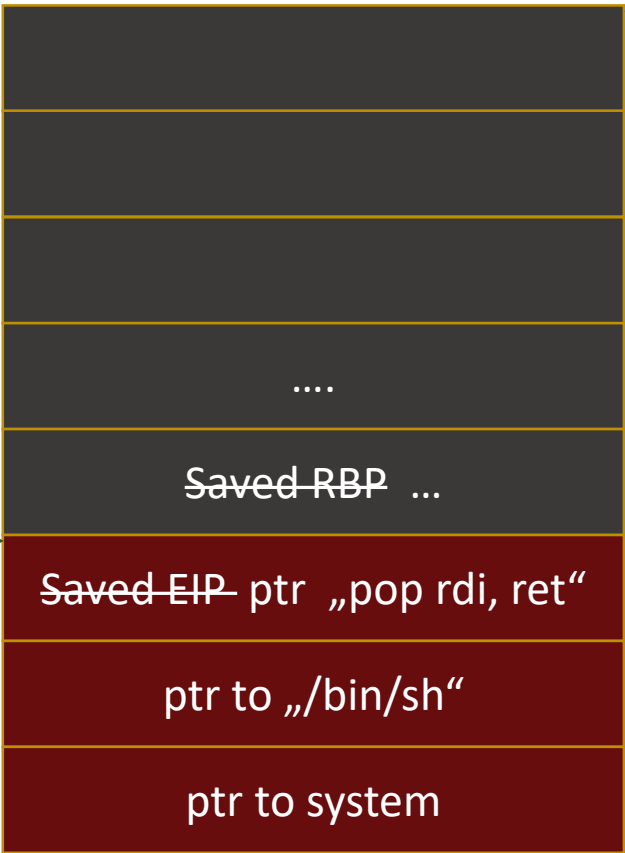


0x7FFFFFFF...

0x0000...

← RIP

RSP
(Top of Stack)



← buffer

Payload strategy

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

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

```
....:
...
pop rdi
ret
```

Leave:
mov rsp, rbp
pop rbp

Ret:
"pop rip"

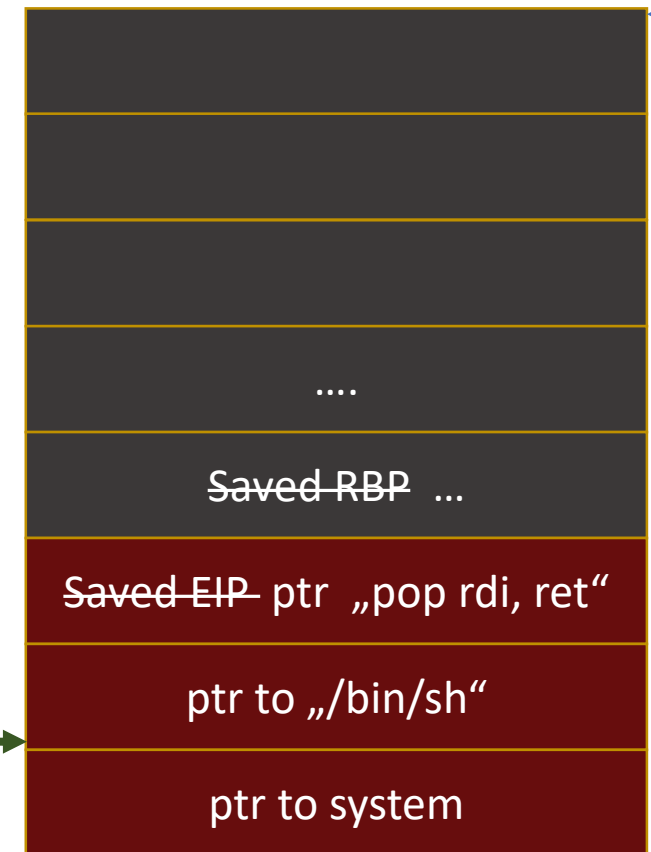
RDI

ptr to „/bin/sh“

RSP
(Top of Stack)

0x7FFFFFFF...

0x0000...



← buffer

← RIP

Payload strategy

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

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

```
....:
...
pop rdi
ret
```

← RIP

Leave:
mov rsp, rbp
pop rbp

Ret:
"pop rip"

RDI

ptr to „/bin/sh“

RSP
(Top of Stack)

0x7FFFFFFF...

0x0000...

← buffer



Payload strategy

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

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

system:
...

← RIP

Leave:
mov rsp, rbp
pop rbp

Ret:
"pop rip"

RDI

ptr to „/bin/sh“

RSP

0x7FFFFFFF...

0x0000...

← buffer

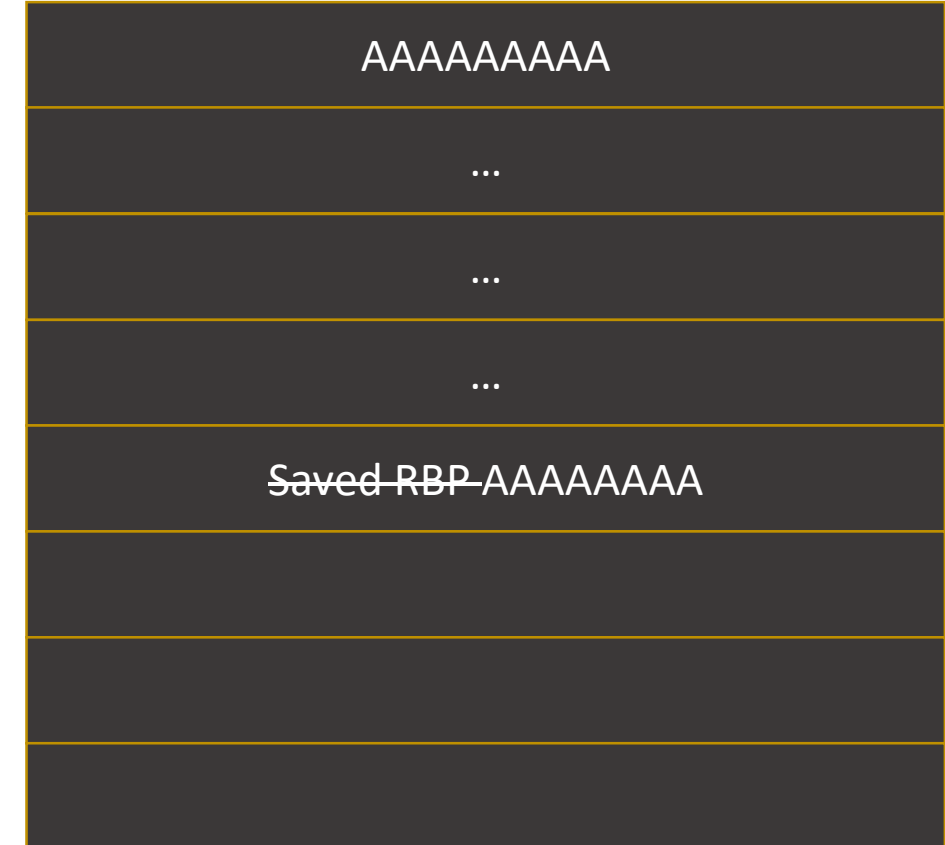


64 Bit – simple ROP-chain

Payload = "A" * 32

+ "AAAAAAAA" (~~saved RBP~~)

0x0000...



0x7FFF...

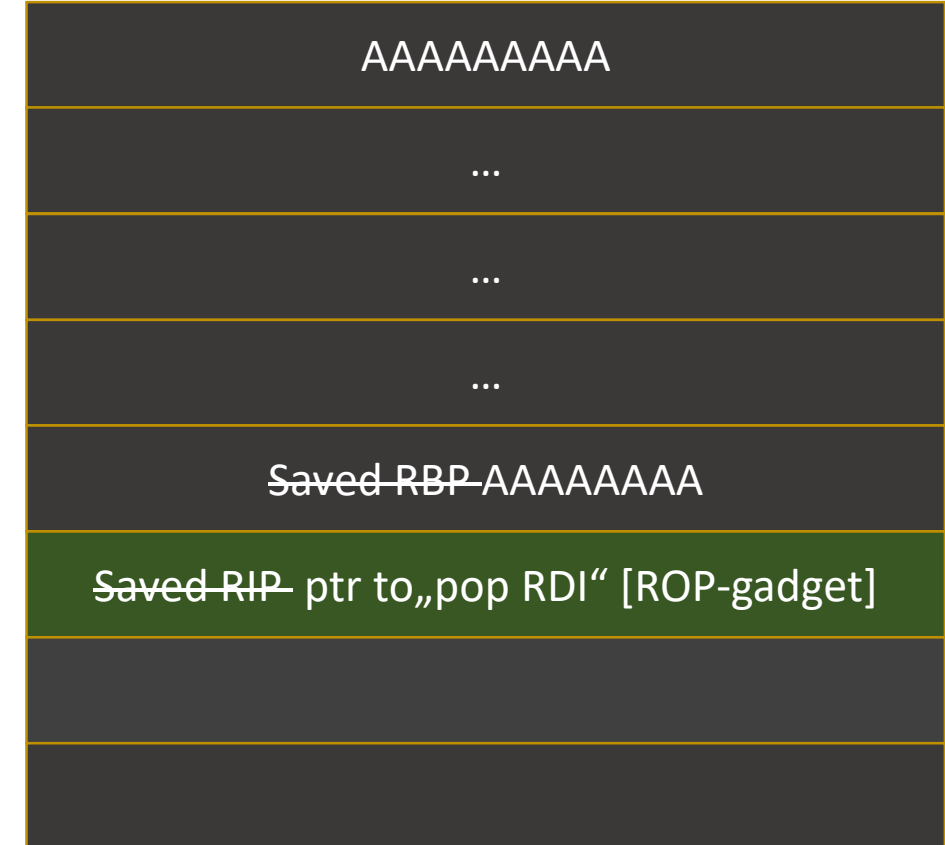
64 Bit – simple ROP-chain

Payload = "A" * 32

+ "AAAAAAAA" (~~saved RBP~~)

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

0x0000...



0x7FFF...



64 Bit – simple ROP-chain

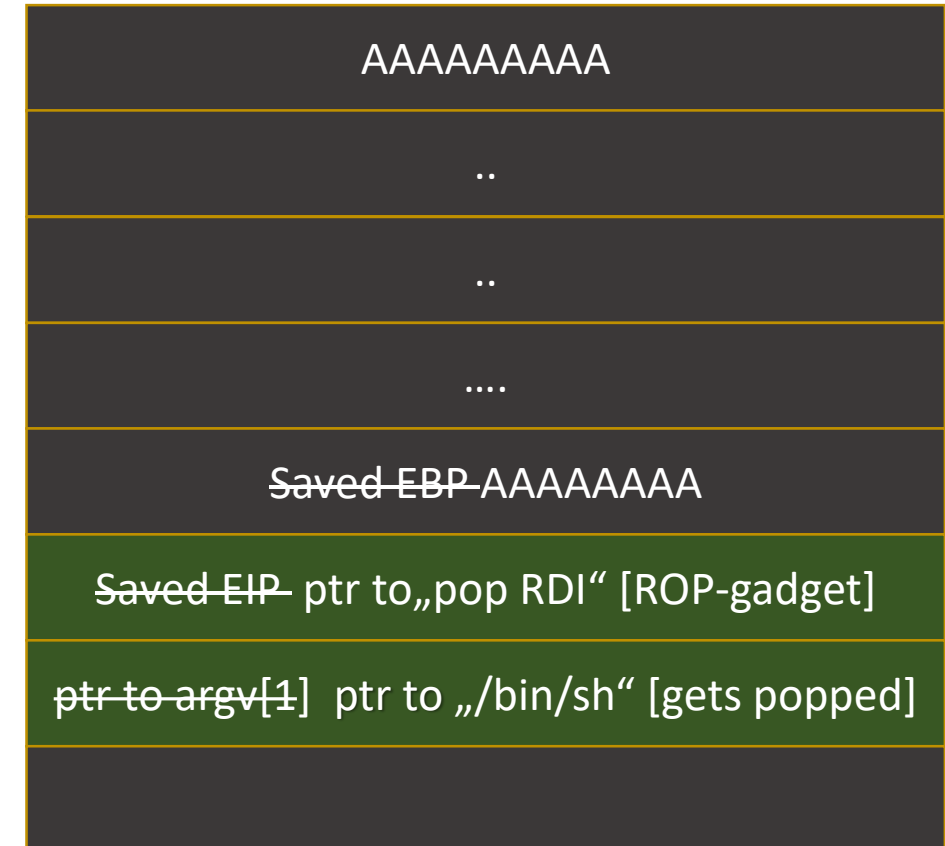
Payload = "A" * 32

+ "AAAAAAAA" (~~saved RBP~~)

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

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

0x0000...



0x7FFF...



64 Bit – simple ROP-chain

Payload = "A" * 32

+ "AAAAAAAA" (~~saved RBP~~)

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

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

+ address of system

0x0000...



0x7FFF...

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

0x7FFFFFFF

stack



libc

`"/bin/sh"`

`system(){ ...}`

Libc-base



heap

.text

0x00000000

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

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



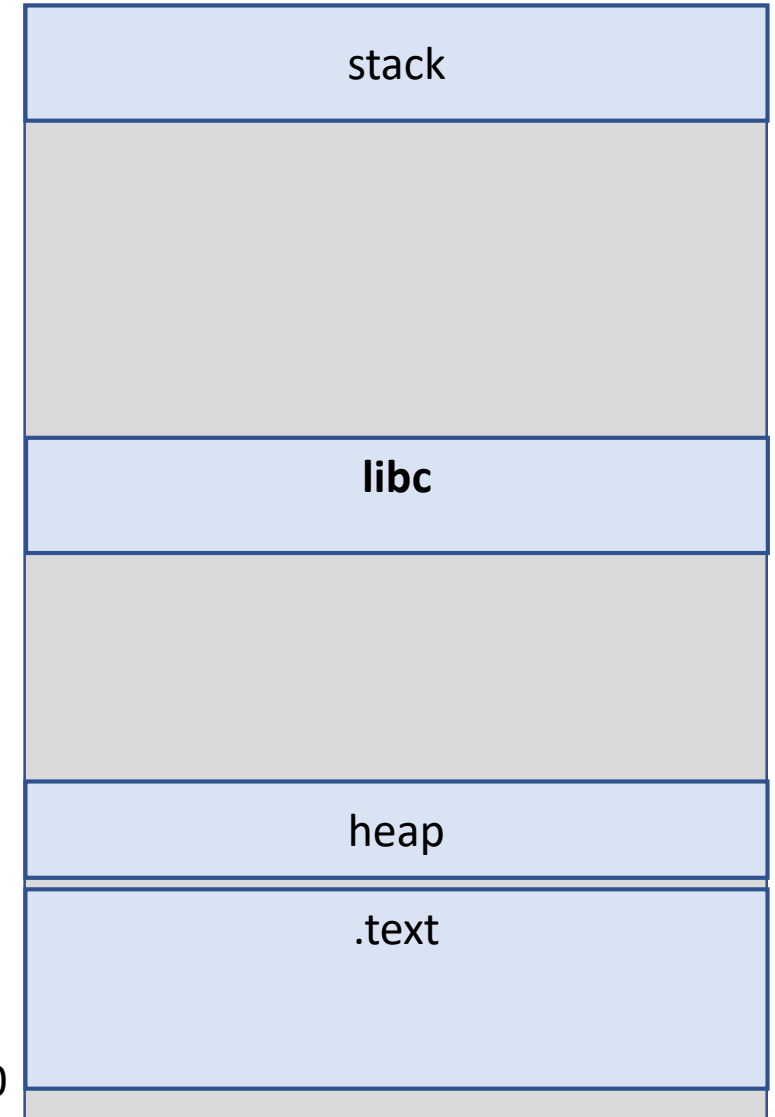
tools

Libc Base	gdb-peda	\Rightarrow run \Rightarrow Ctrl + c \Rightarrow vmmap
Offset system	Command line	readelf -s /path/to/libc grep system
Offset "/bin/sh"	Command line	strings -tx /path/to/libc grep /bin/sh

64 Bit – ASLR enabled

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

0x7FFF...

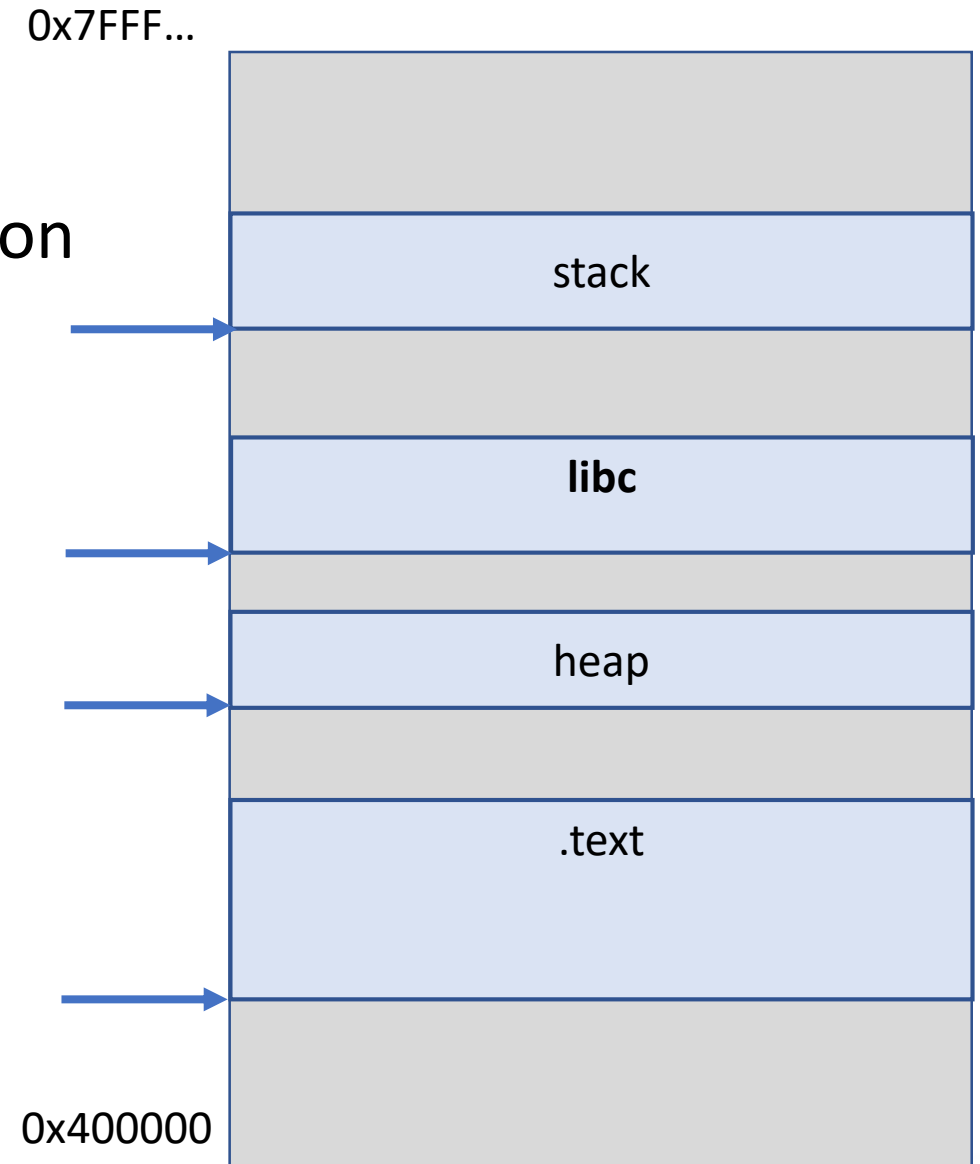


0x400000



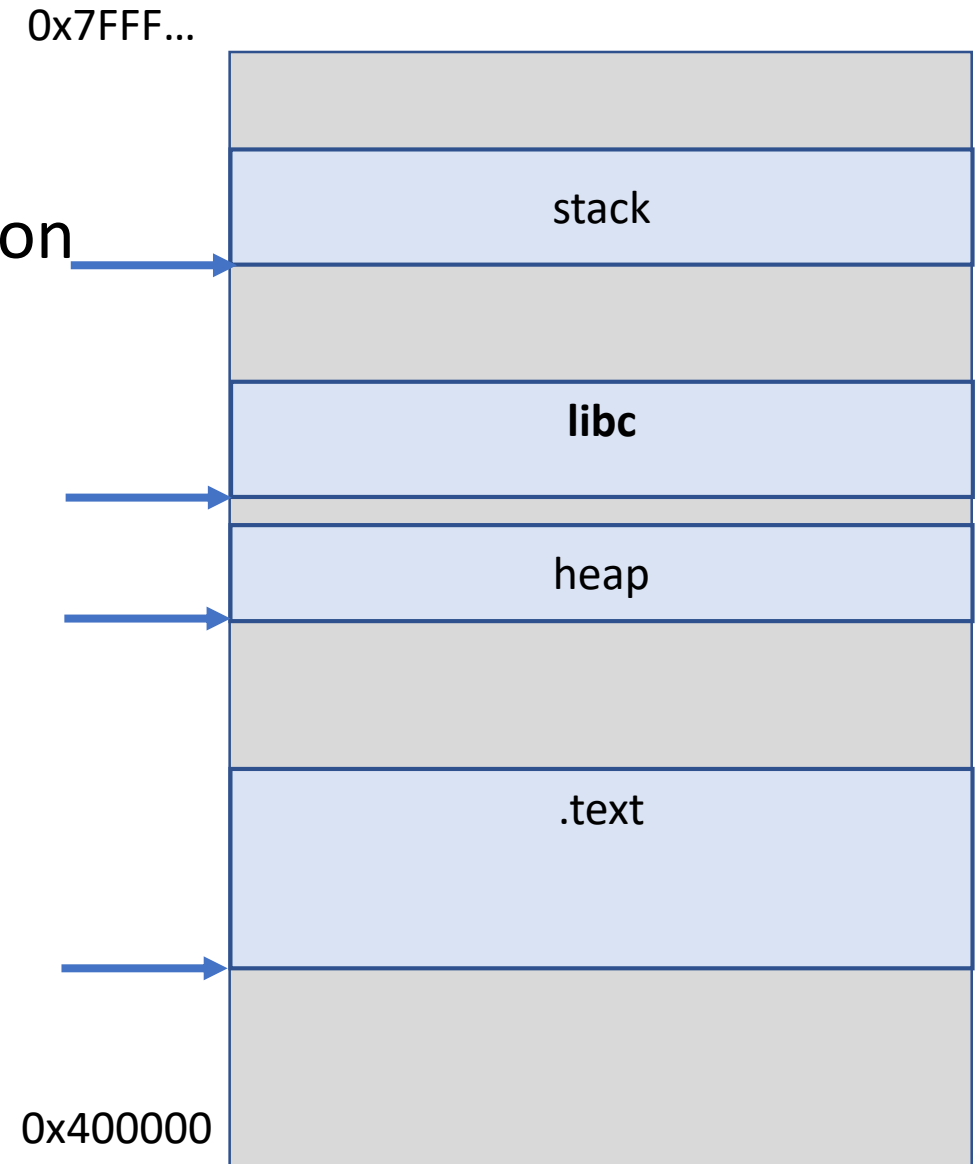
64 Bit – ASLR enabled

- 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



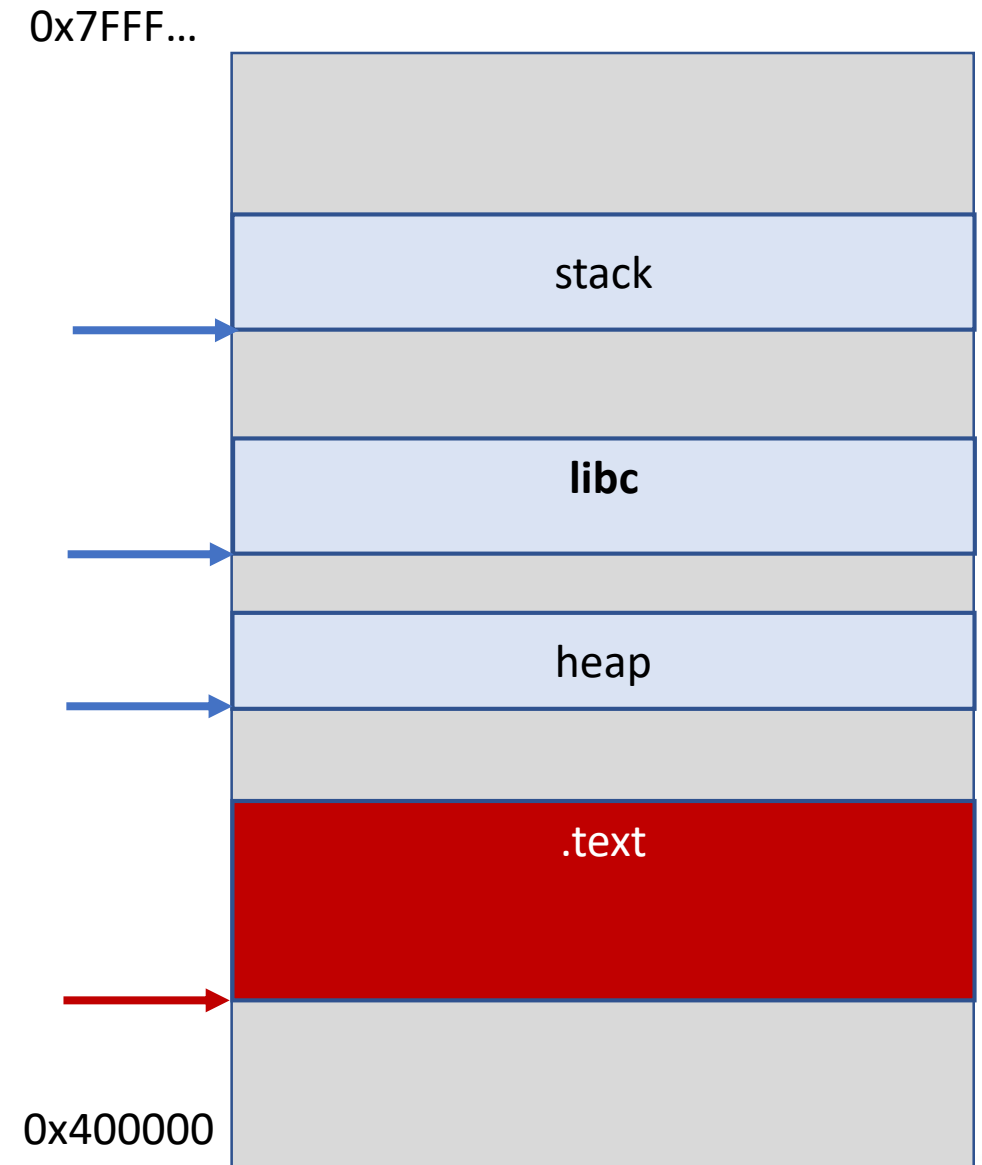
64 Bit – ASLR enabled

- 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



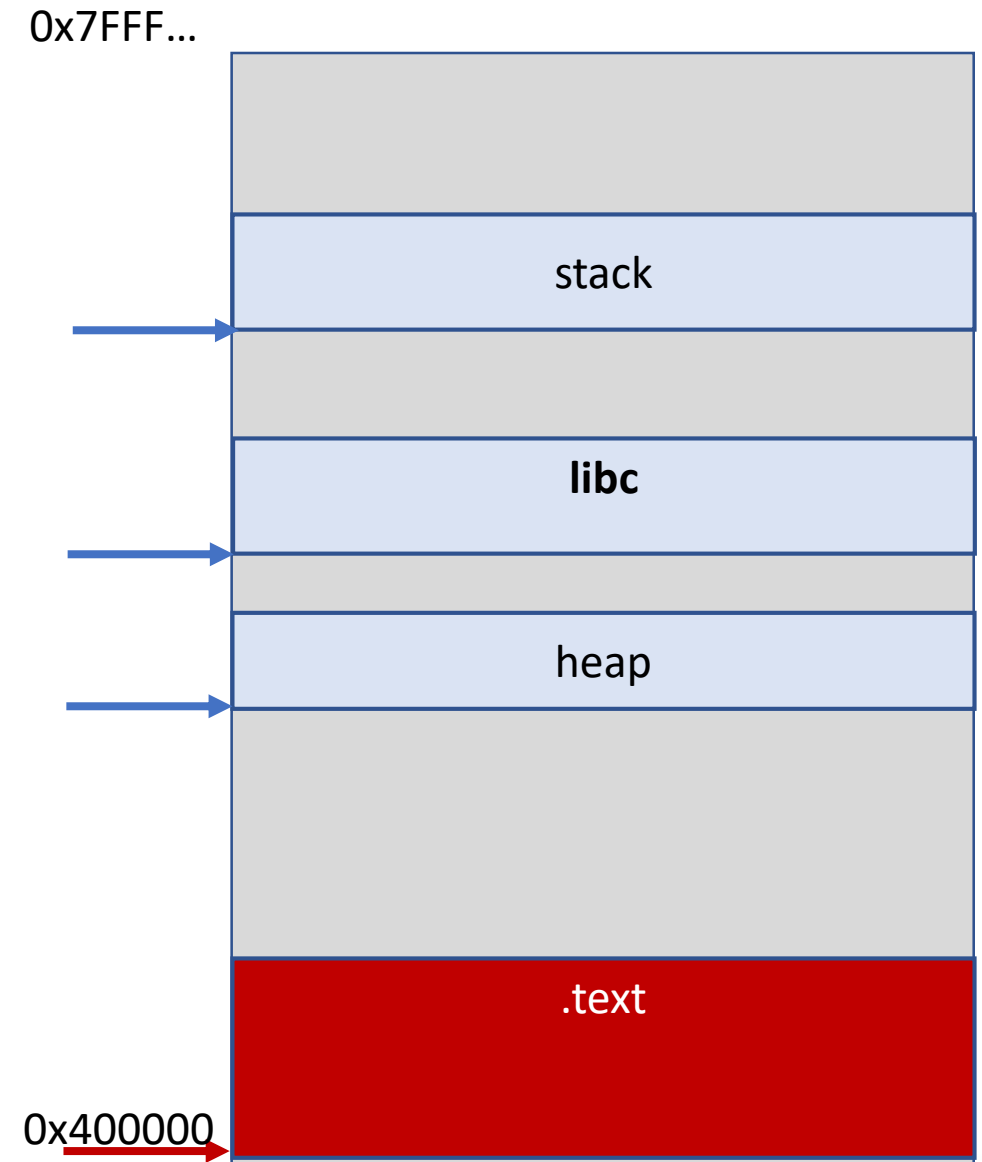
64 Bit – ASLR enabled

- **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

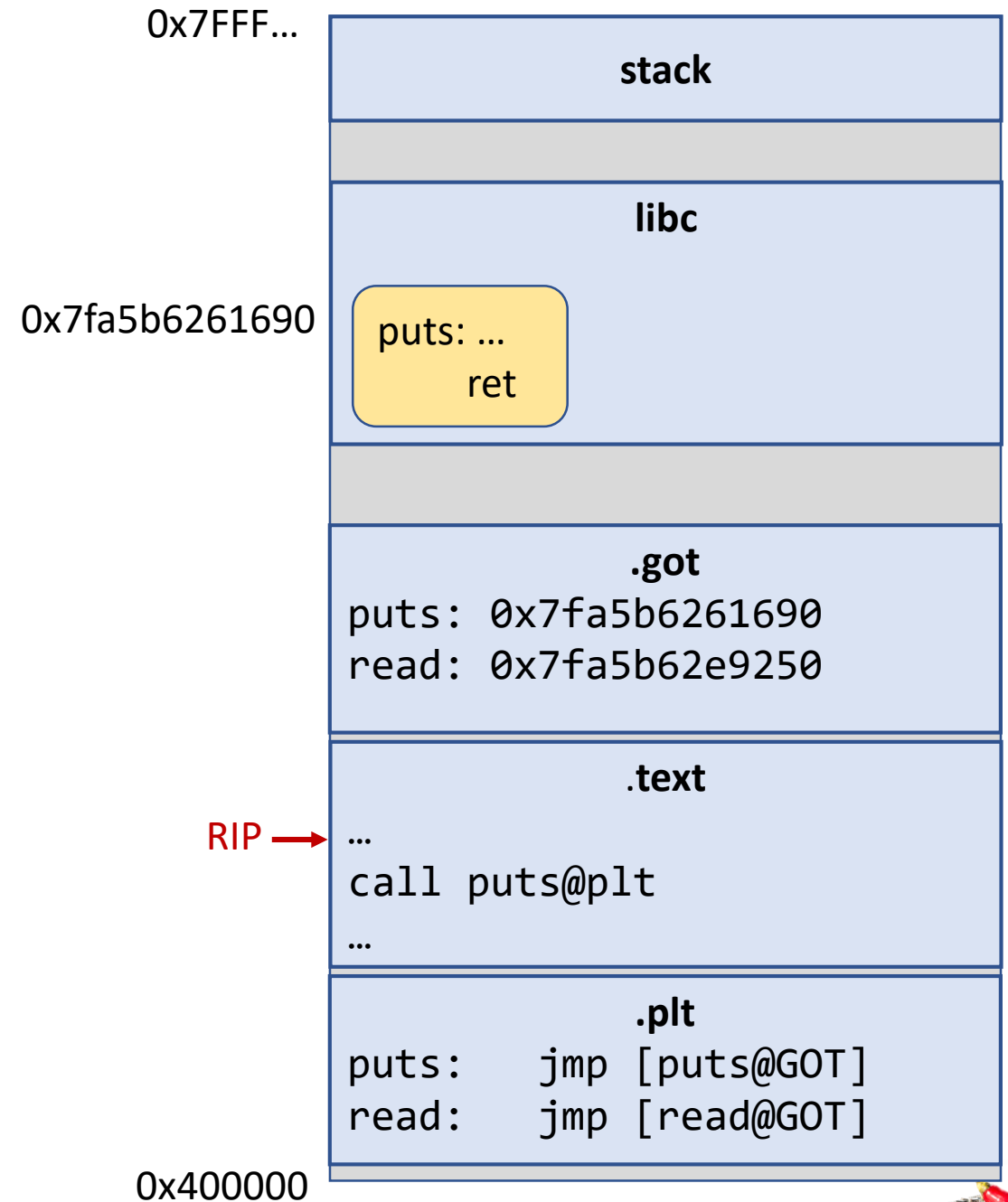


GOT and PLT

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

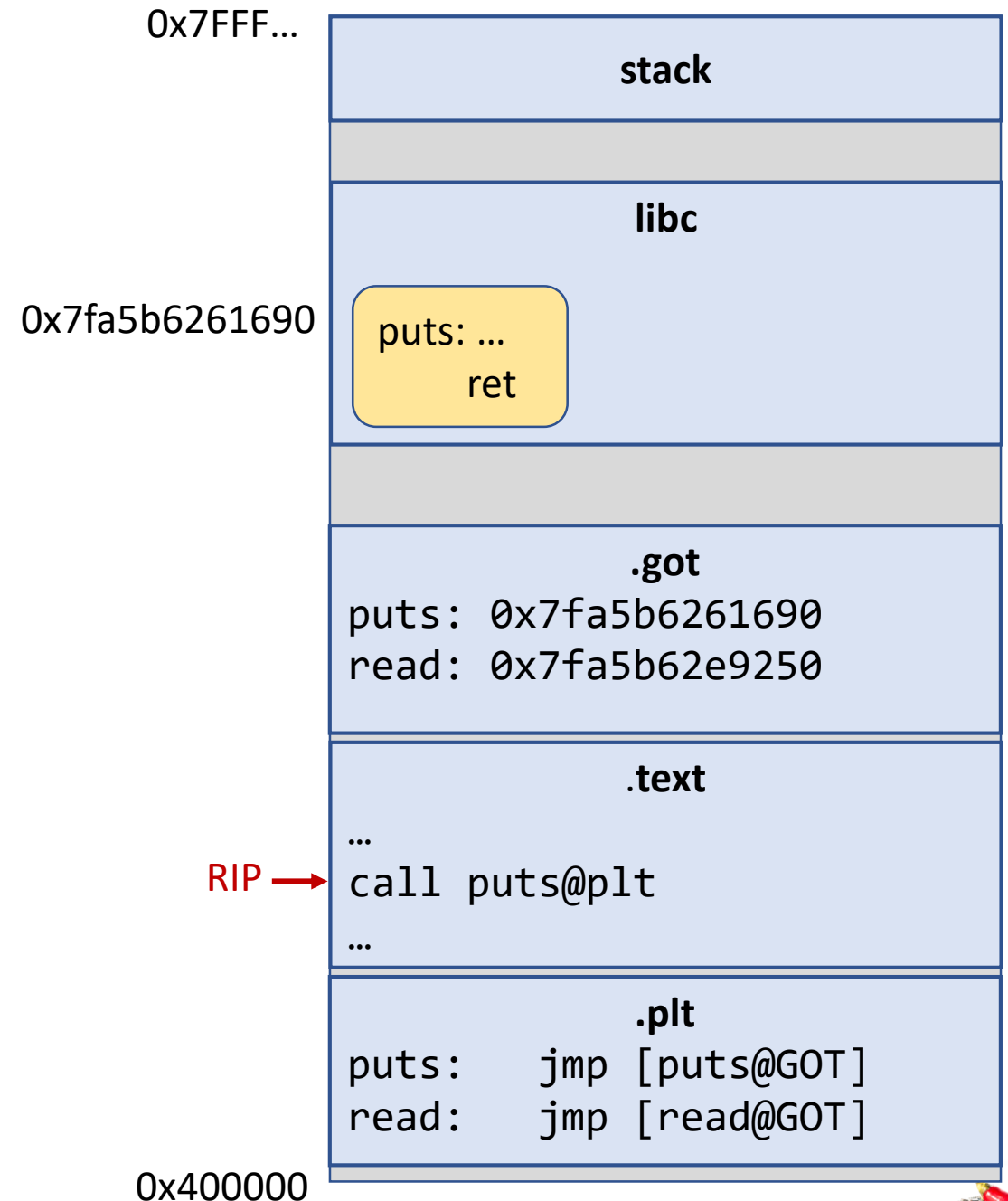


GOT and PLT

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

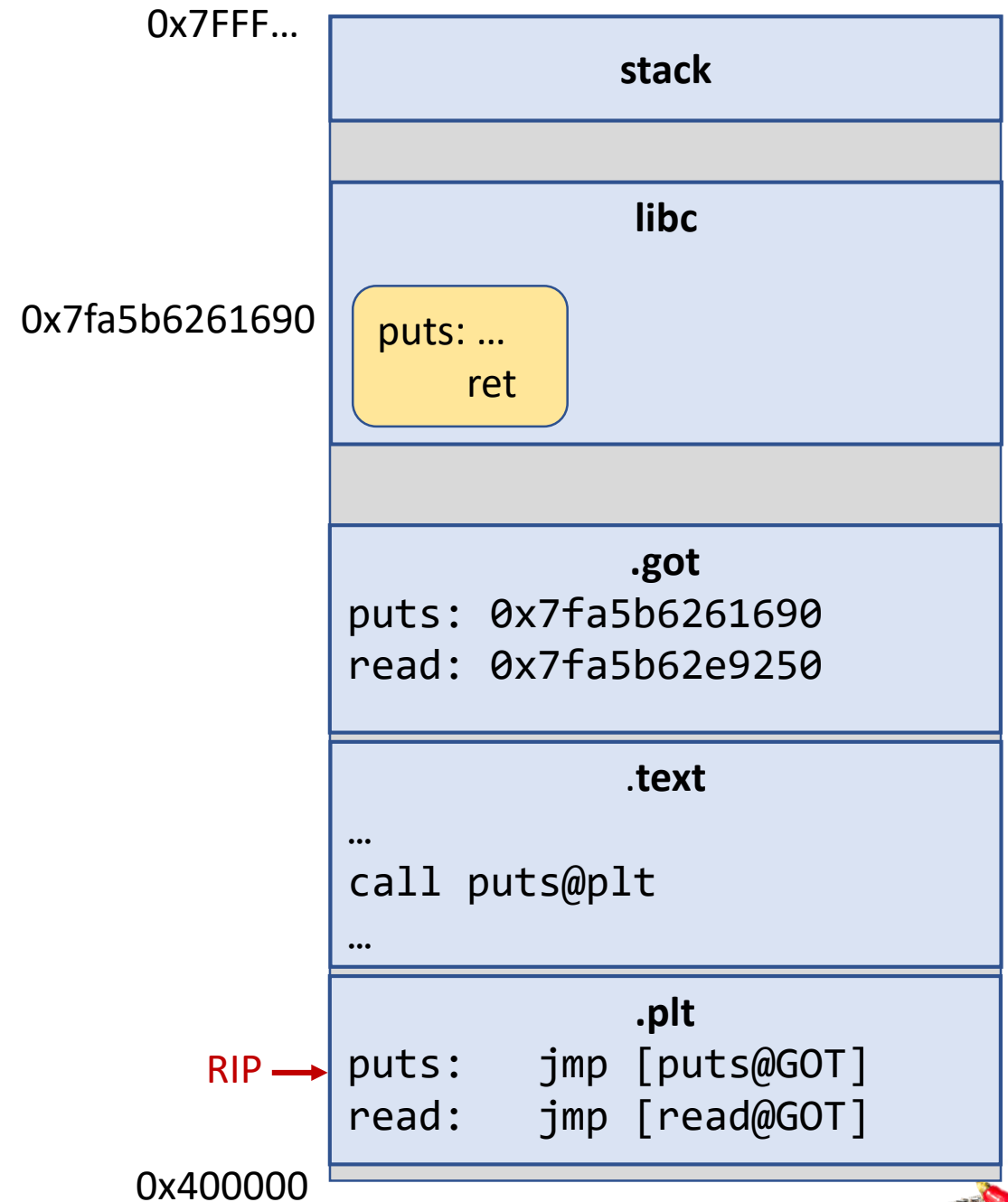


GOT and PLT

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

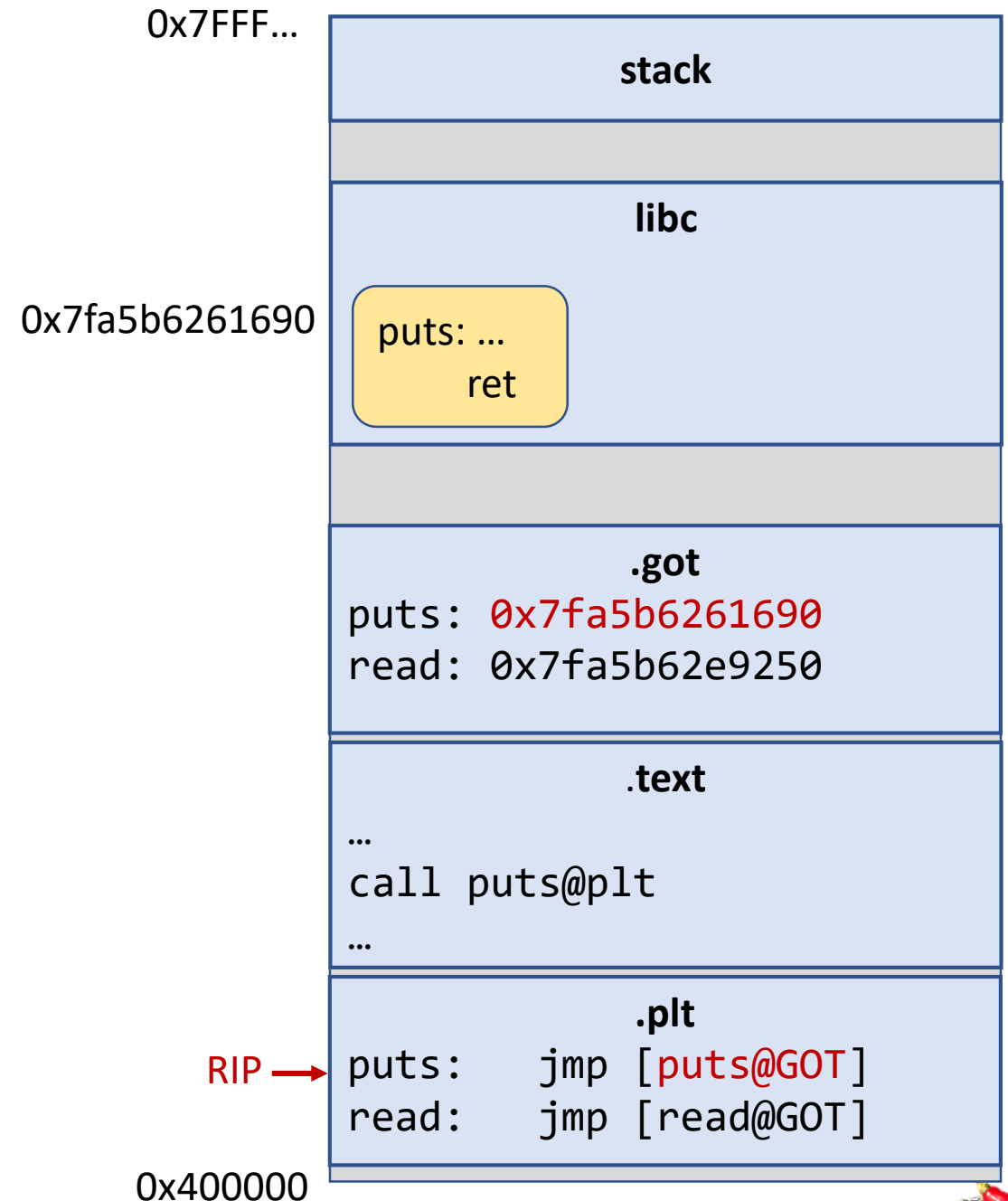


GOT and PLT

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

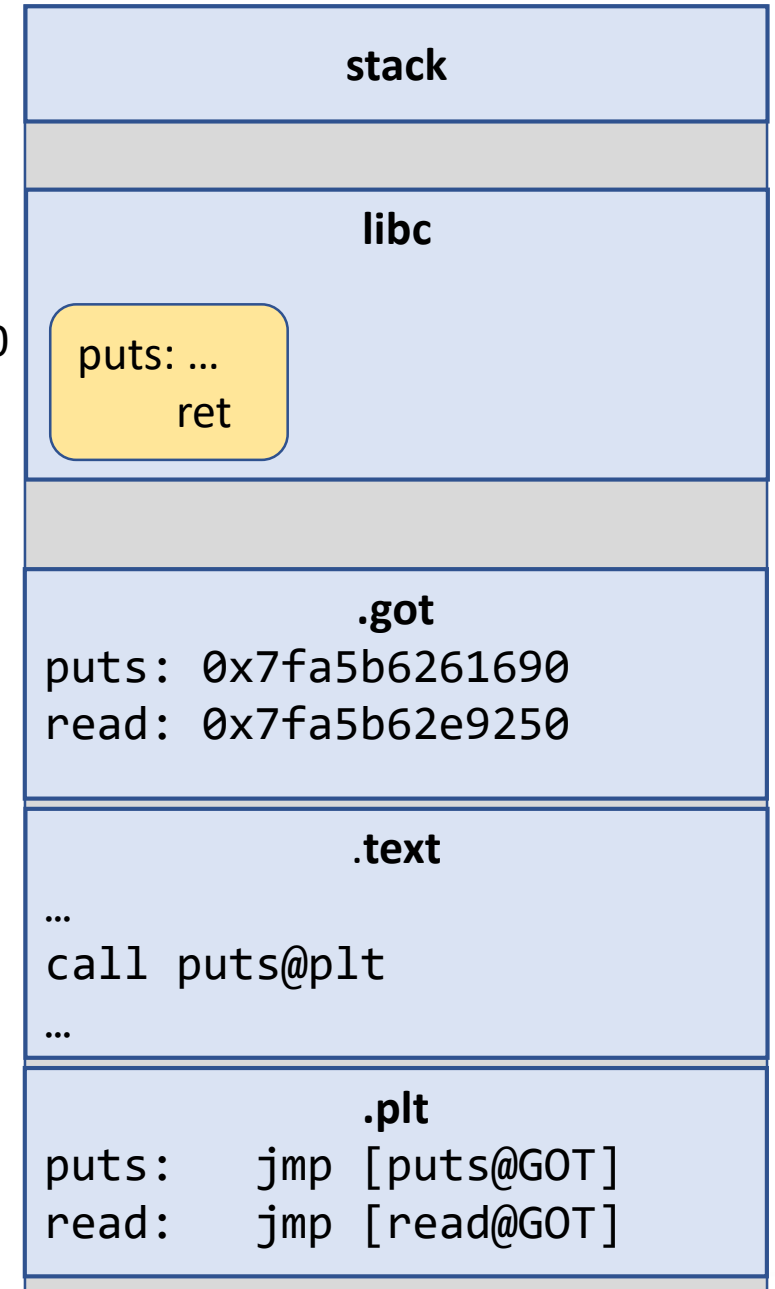
GOT: Global Offset Table

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...



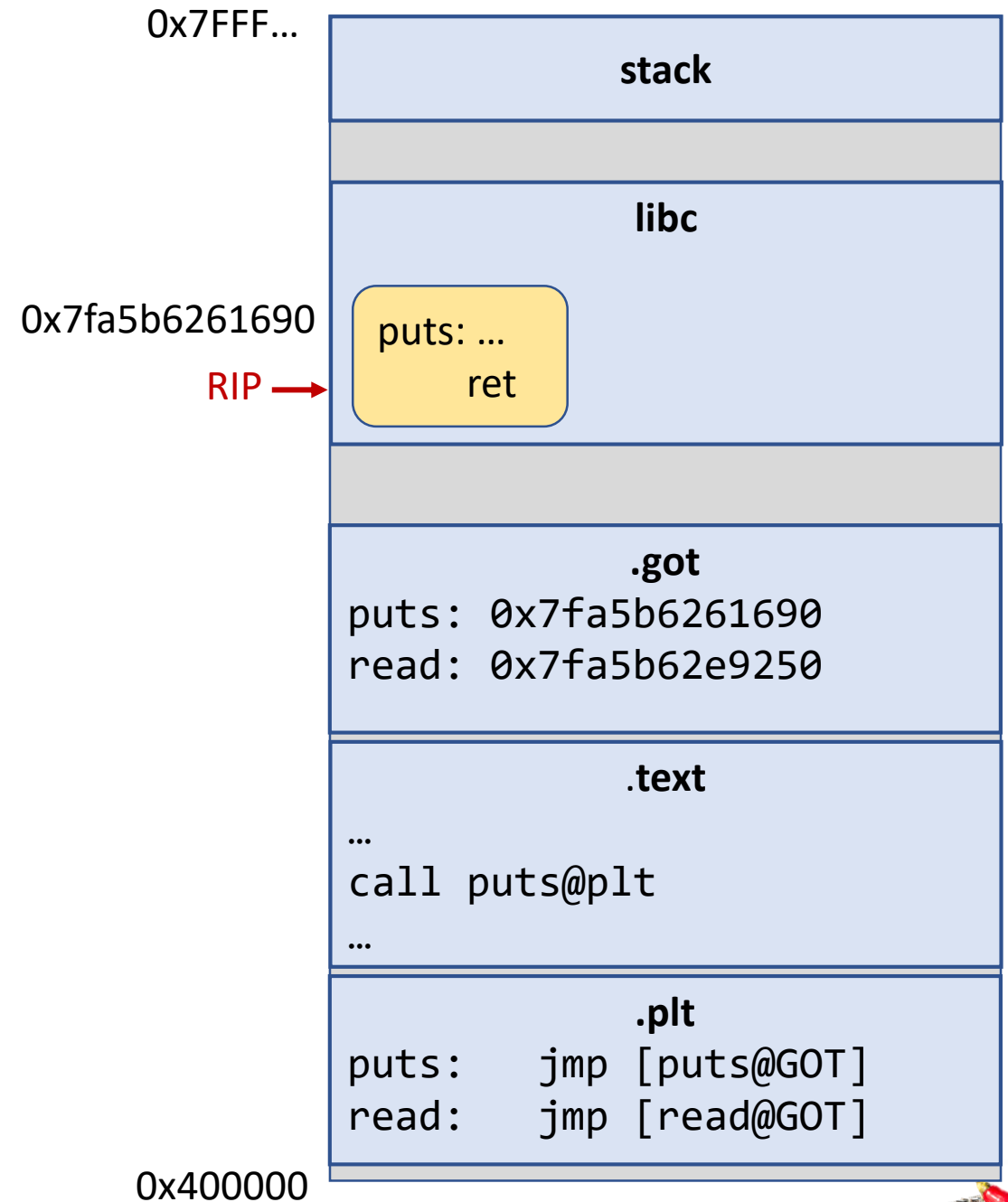
0x400000

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

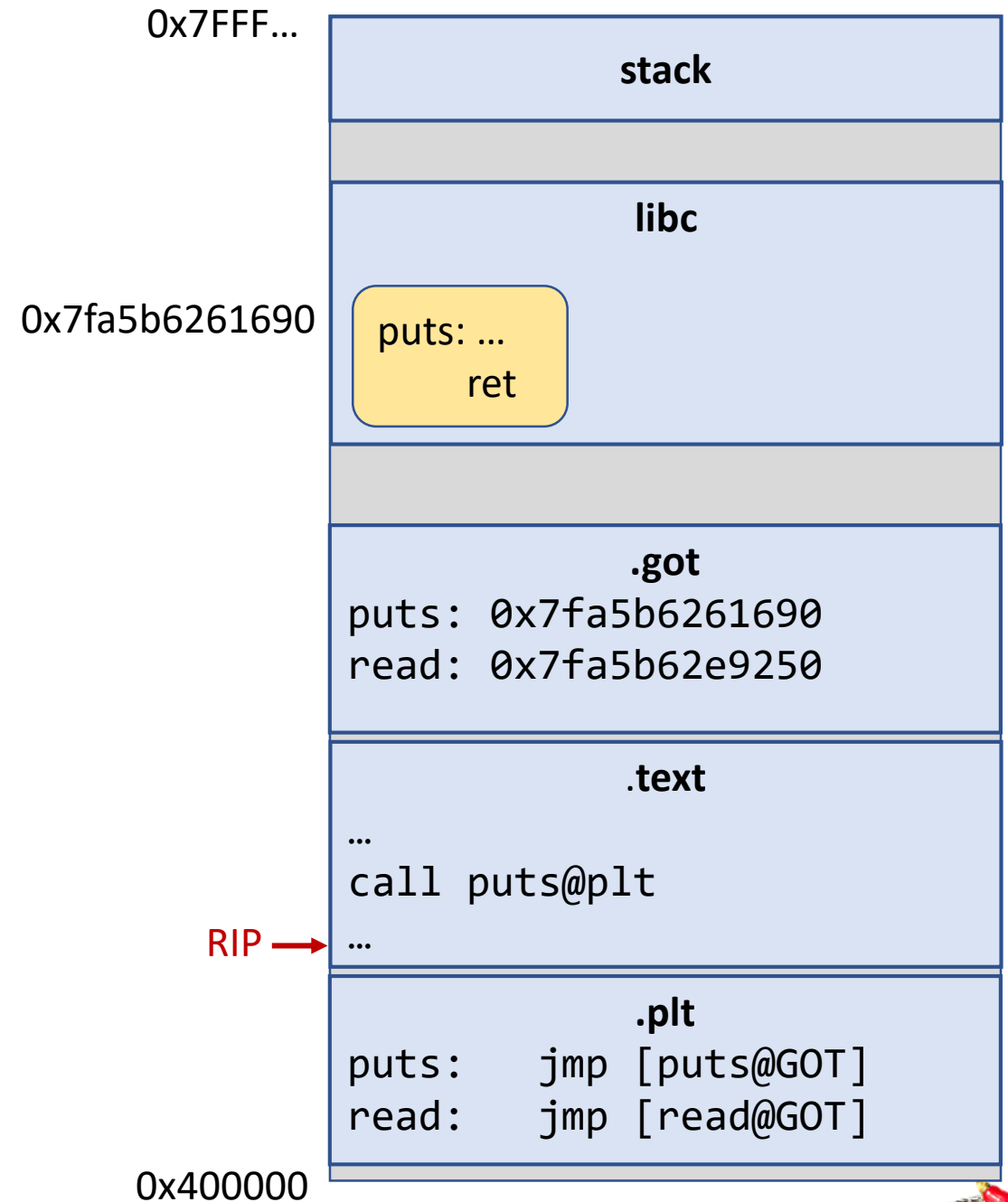


GOT and PLT

GOT: Global Offset Table

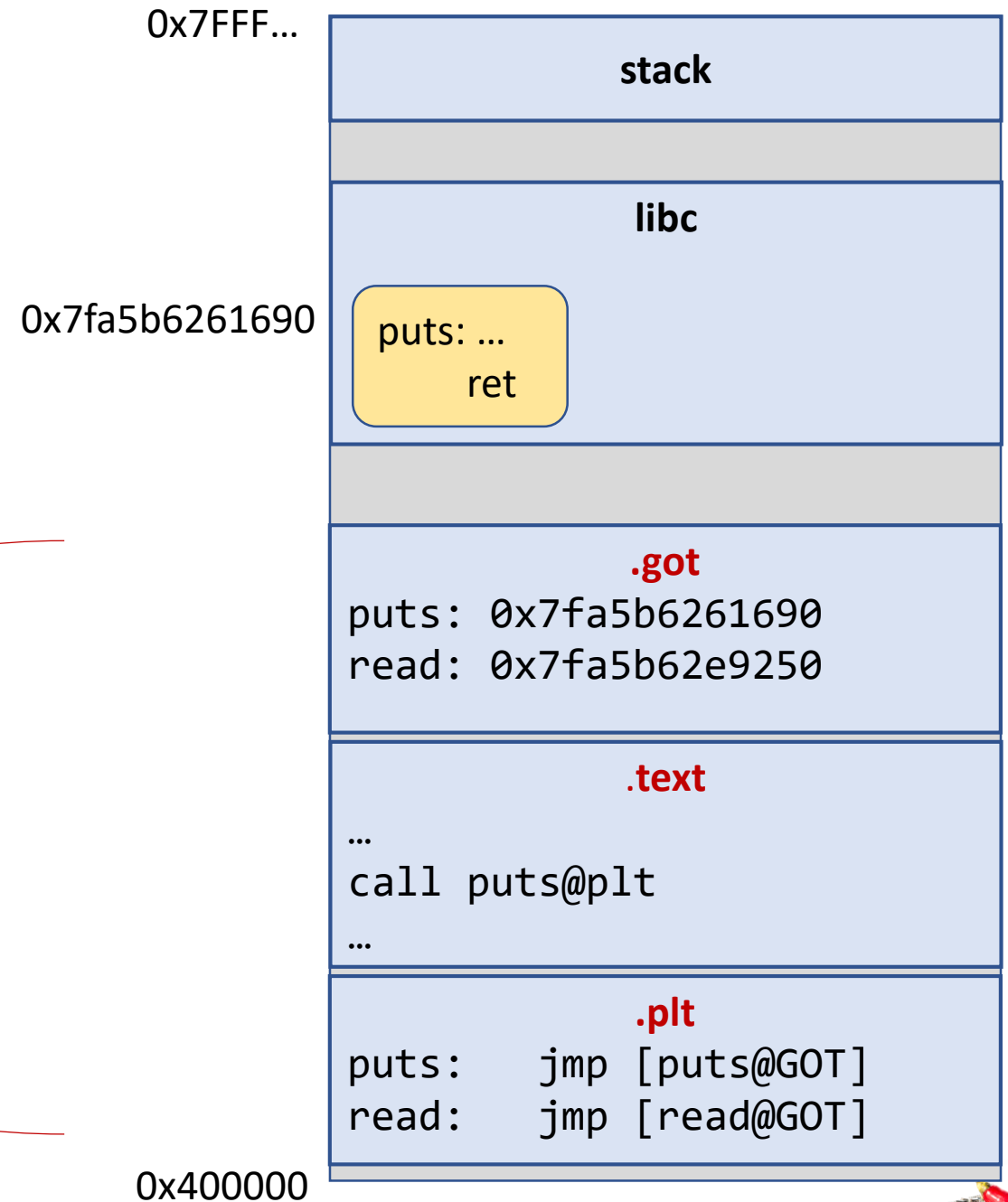
PLT: Procedure Linkage Table

- 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

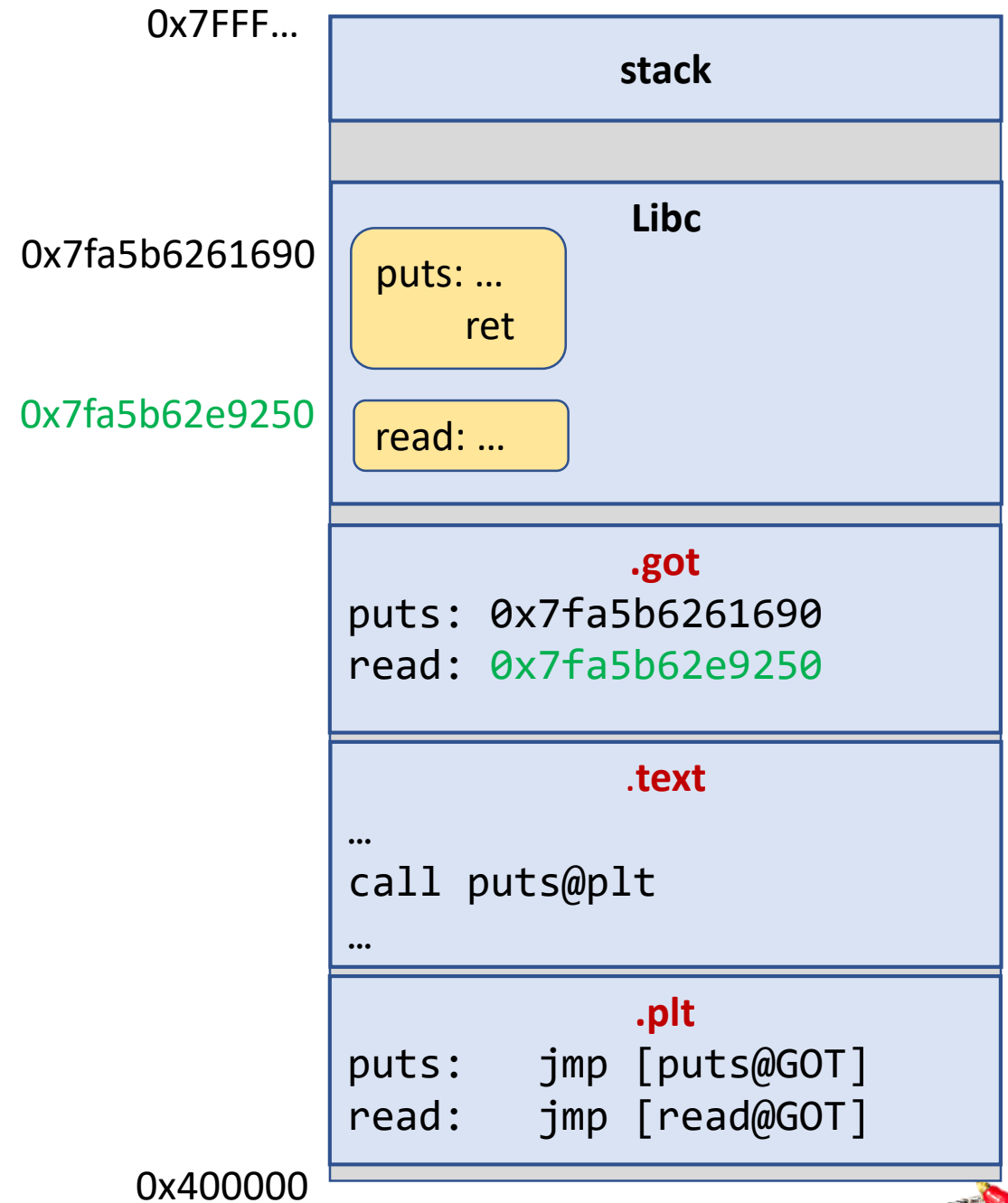


GOT and PLT

Not randomized if not compiled
as PIE



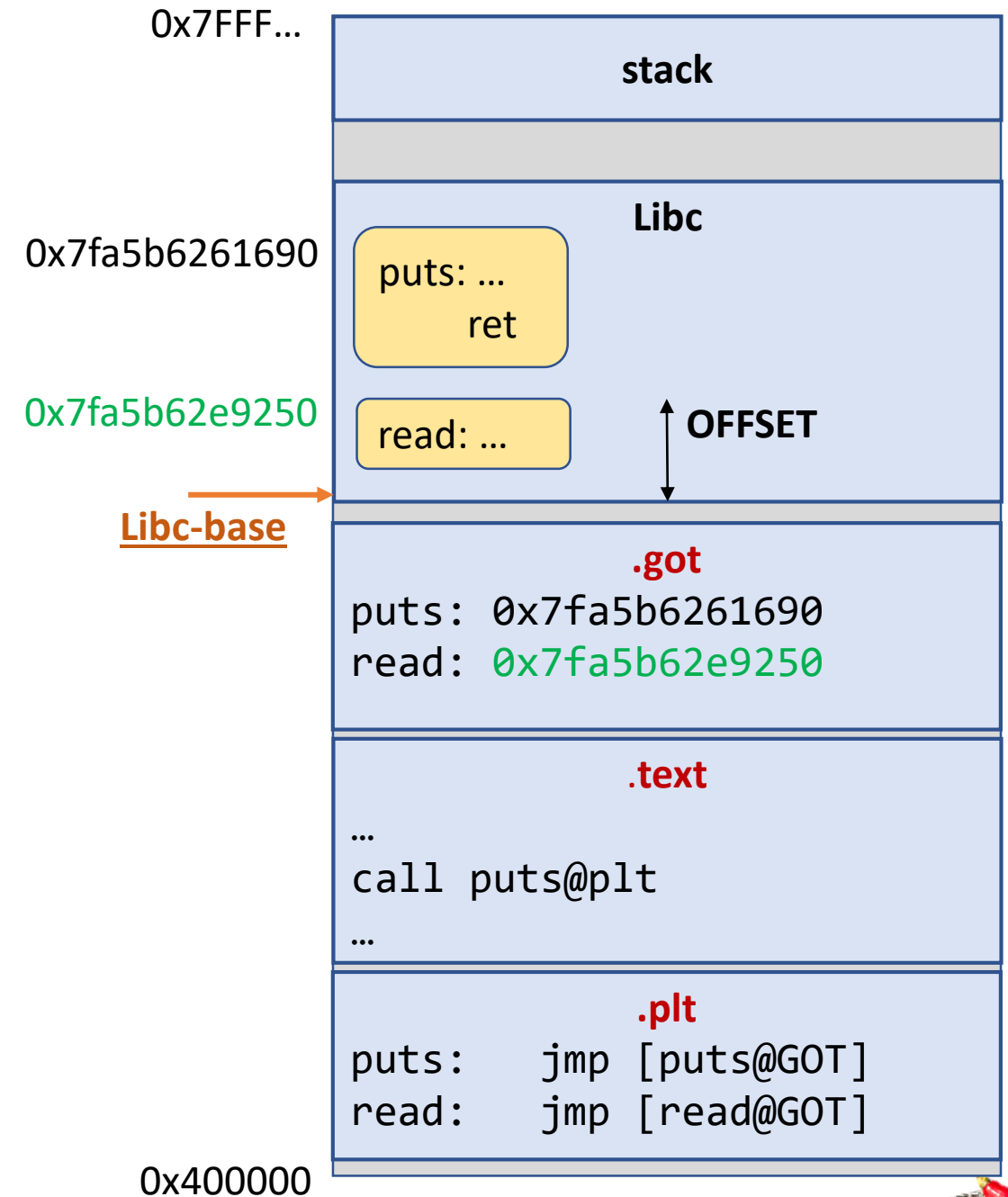
GOT and PLT



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://ropemporium.com/>

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

<https://ringzer0ctf.com>

<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. 35C3 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

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

