# 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,
  Trainer in Berlin

# 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

# 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: Top of the current 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**

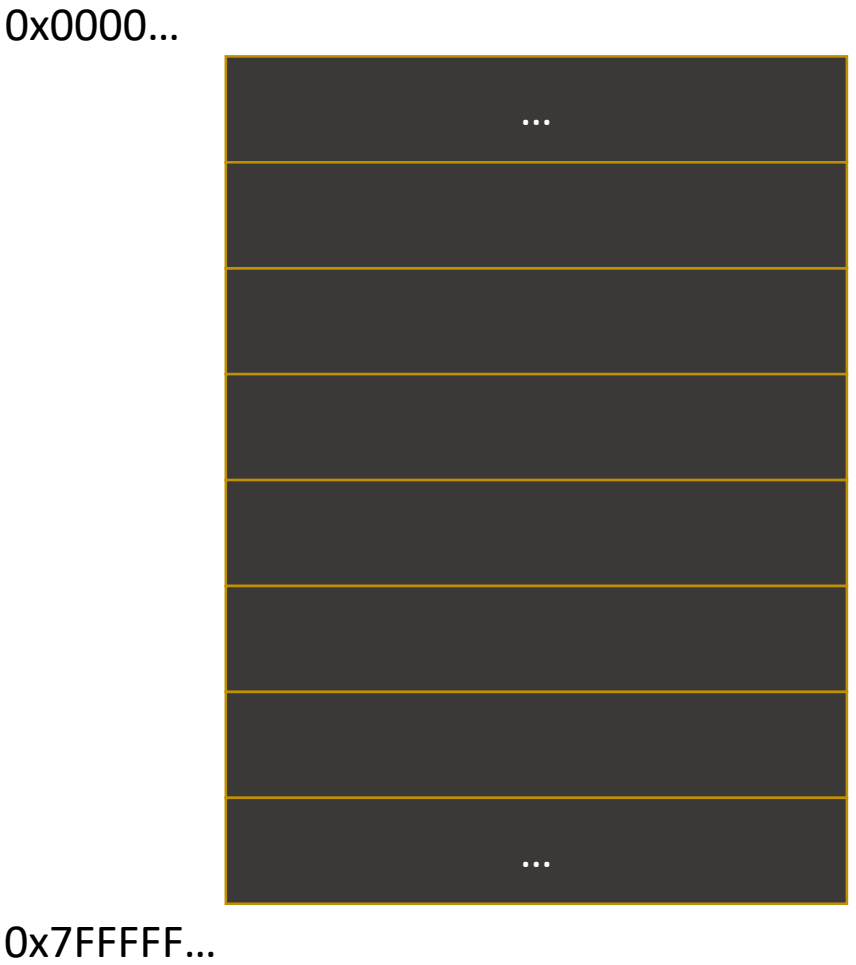


# Recap Buffer Overflow

```
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*)
 …
```

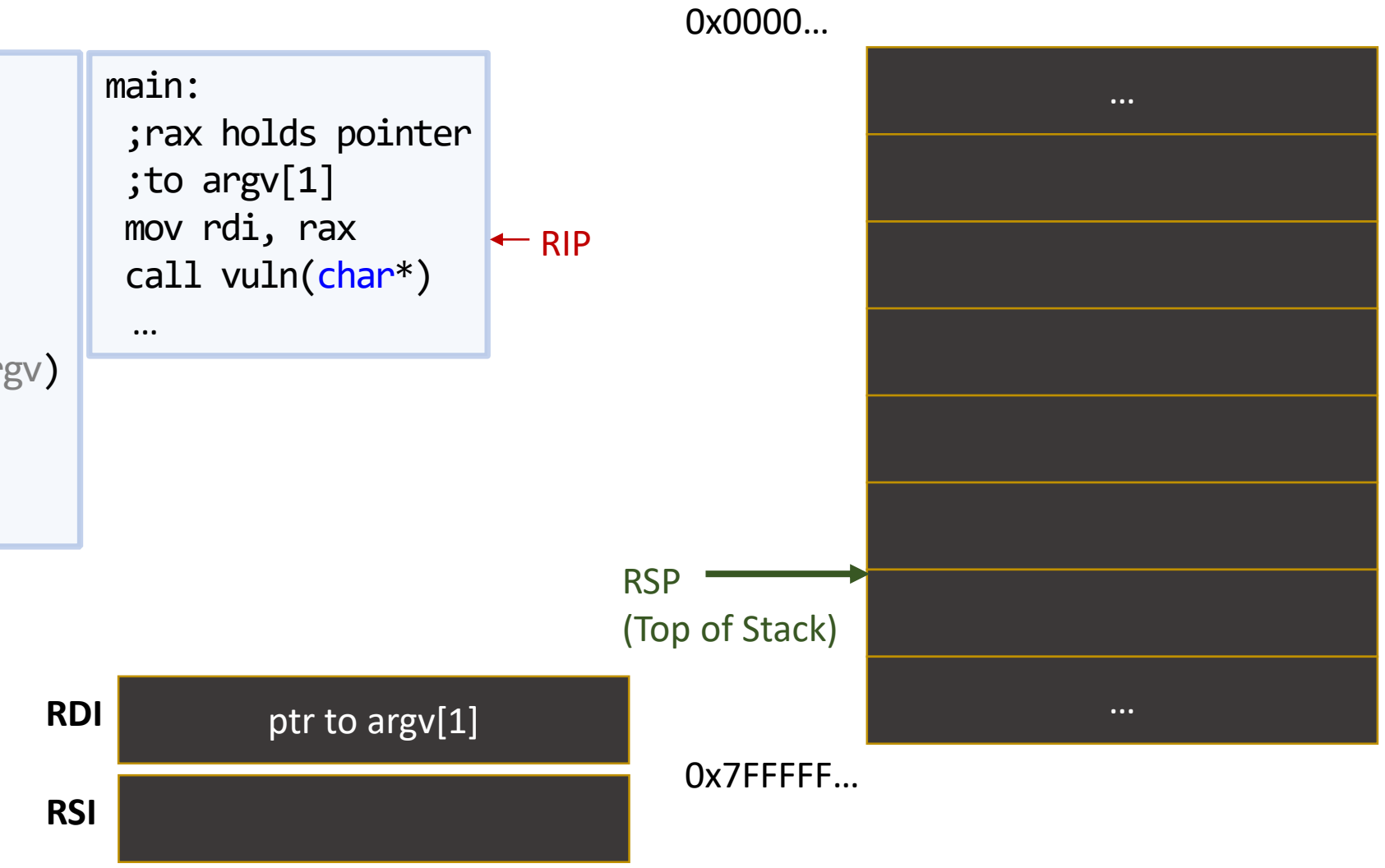
# Recap Buffer Overflow

```
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



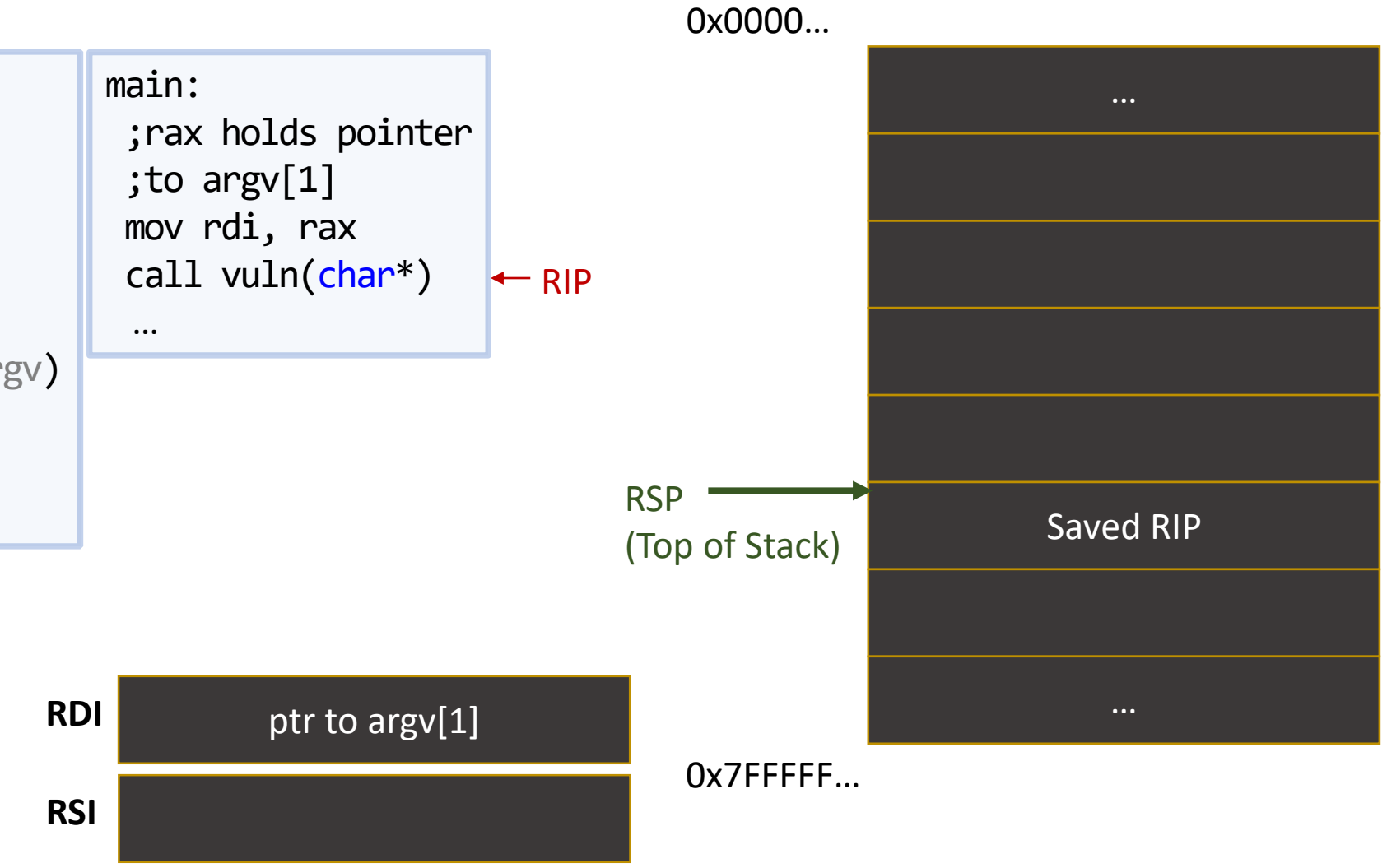
# Recap Buffer Overflow

```
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

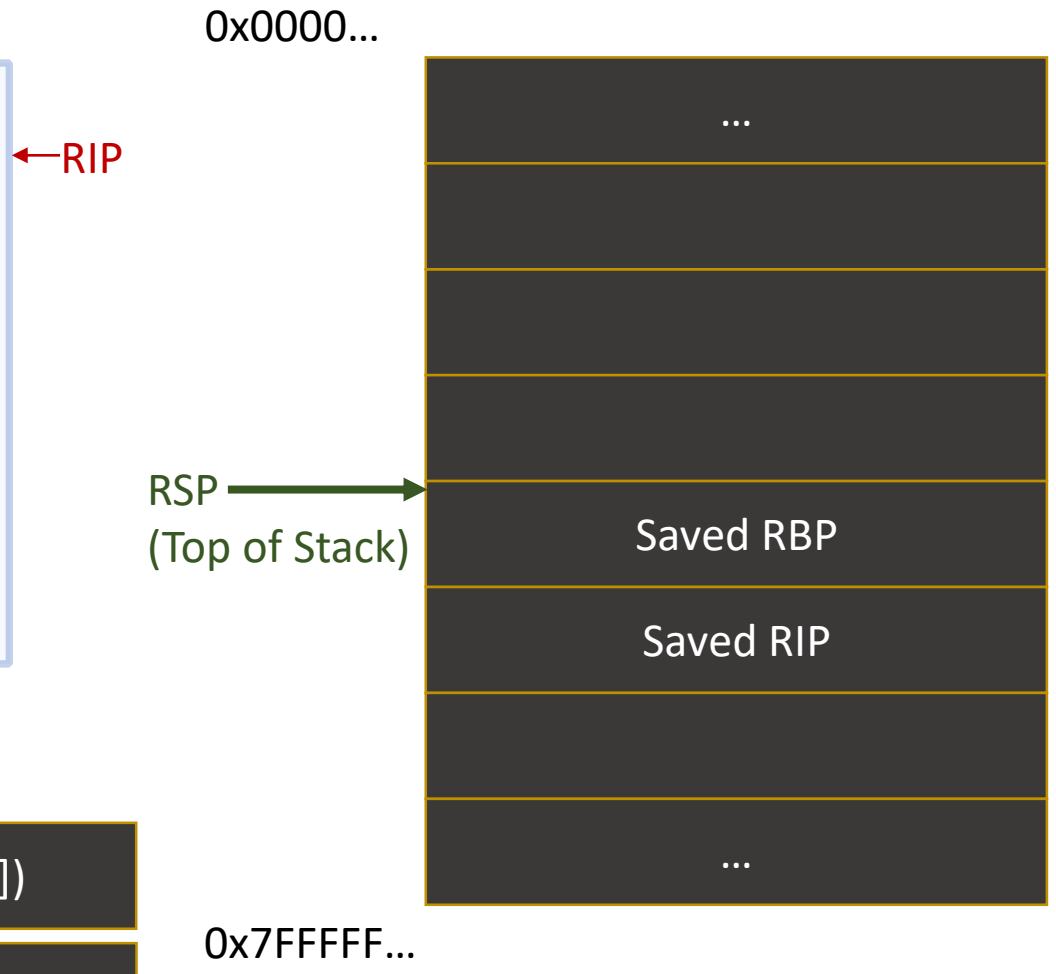


# Recap Buffer Overflow

```
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
```



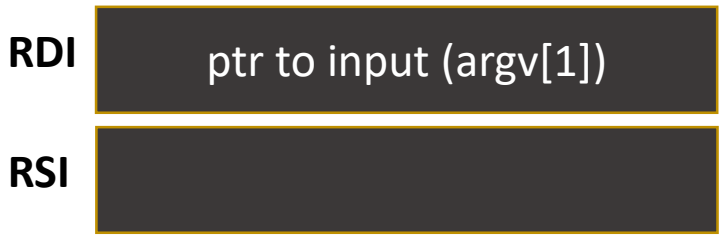
# Recap Buffer Overflow

```
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

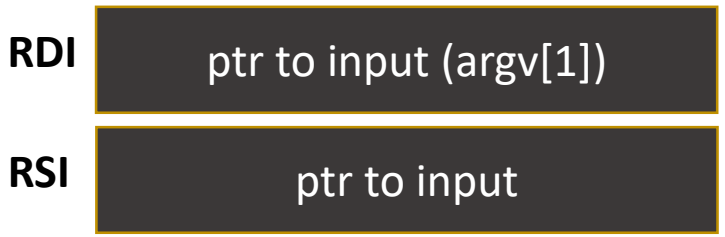
# Recap Buffer Overflow

```
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
```

RIP

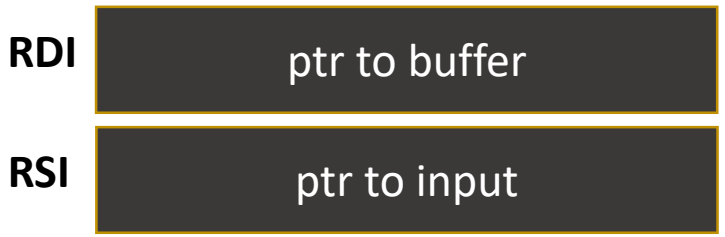
# Recap Buffer Overflow

```
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
```

← RIP

# Recap Buffer Overflow

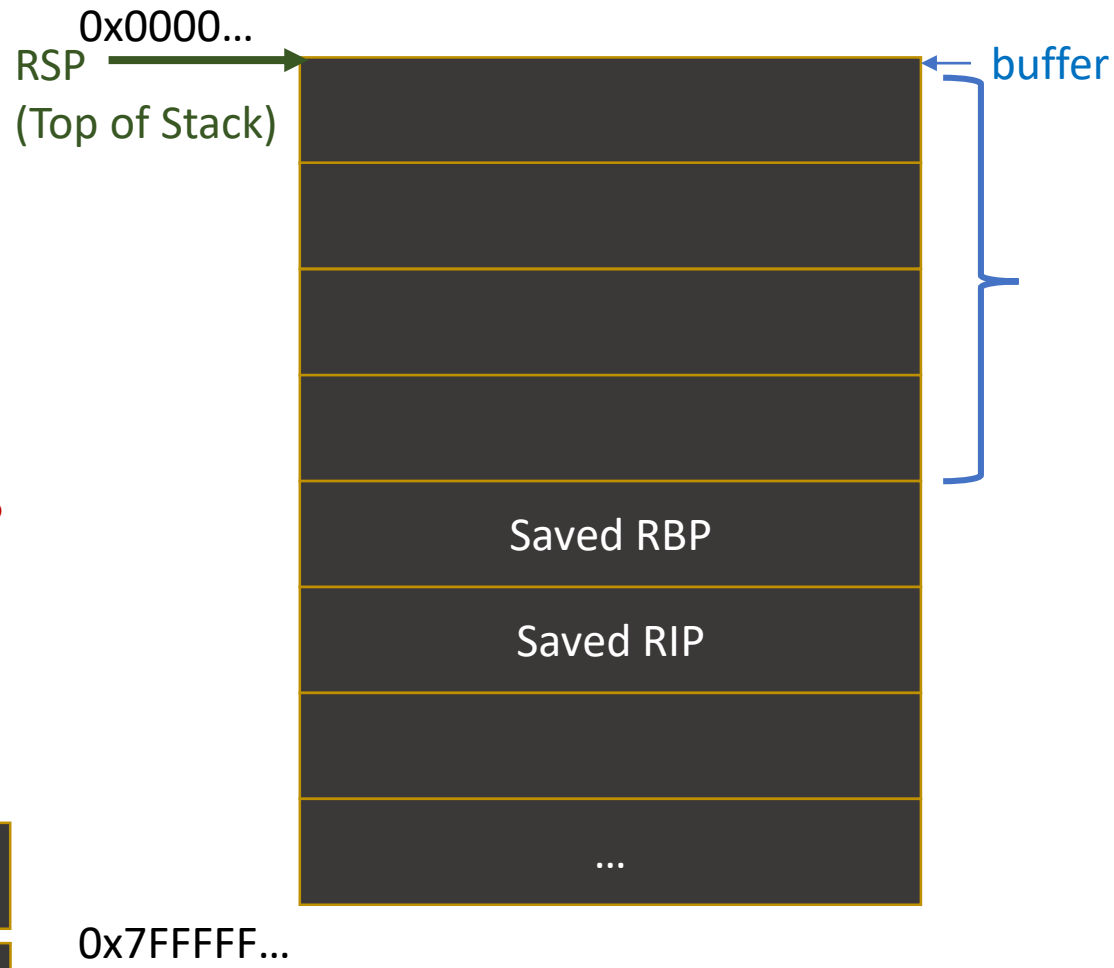
Program call:
> ./myprogram AAA… (31*A)

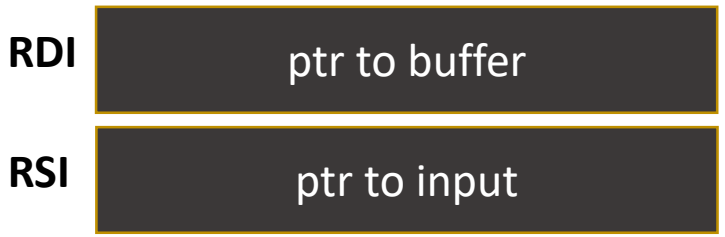
```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

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



# Recap Buffer Overflow

Program call:
> ./myprogram AAA… (31*A)

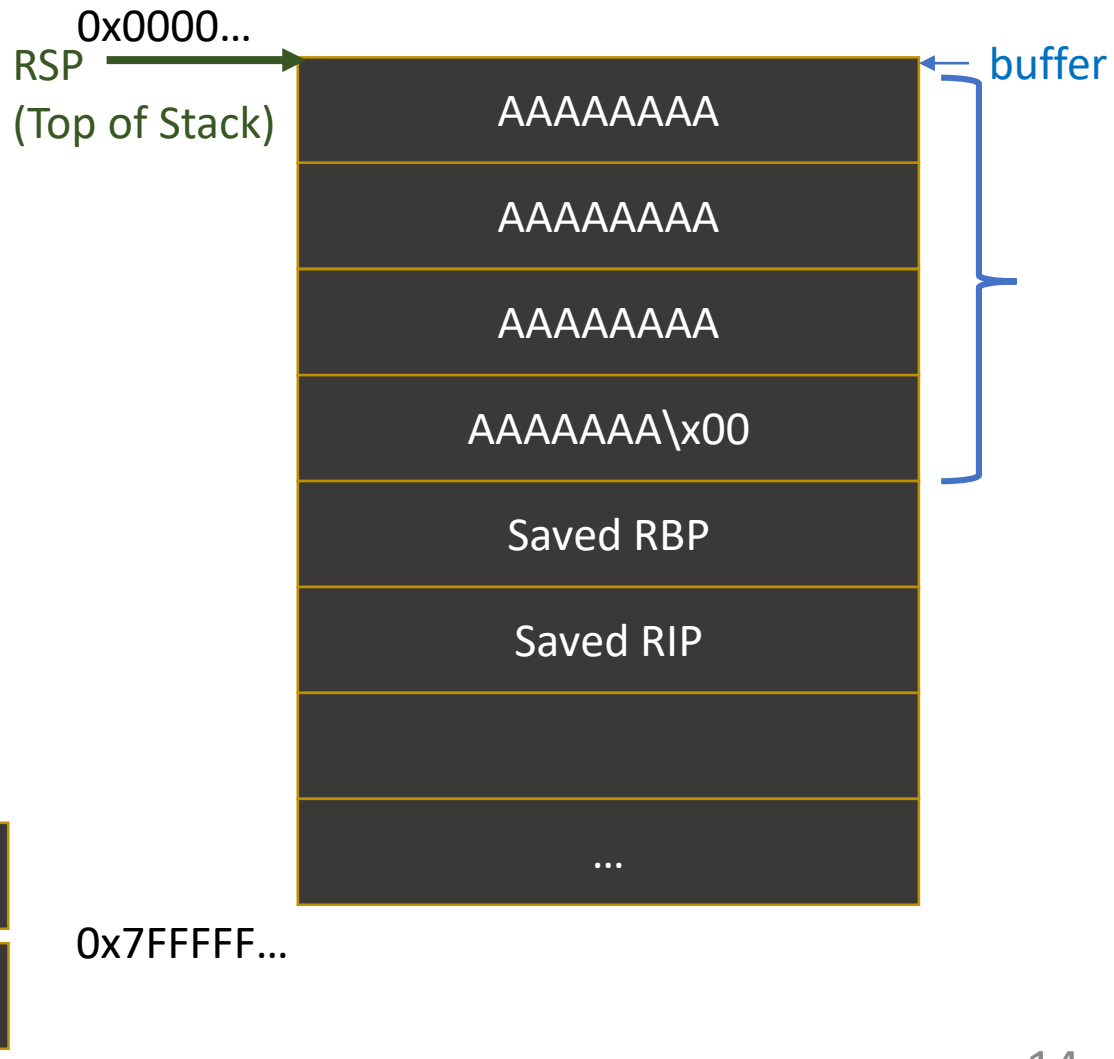
No overflow, we have 32 Bytes and write 32 Bytes

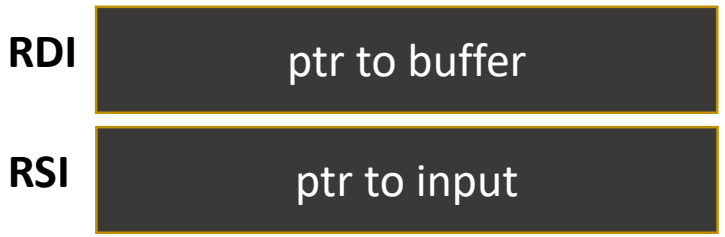
```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

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



# Recap Buffer Overflow

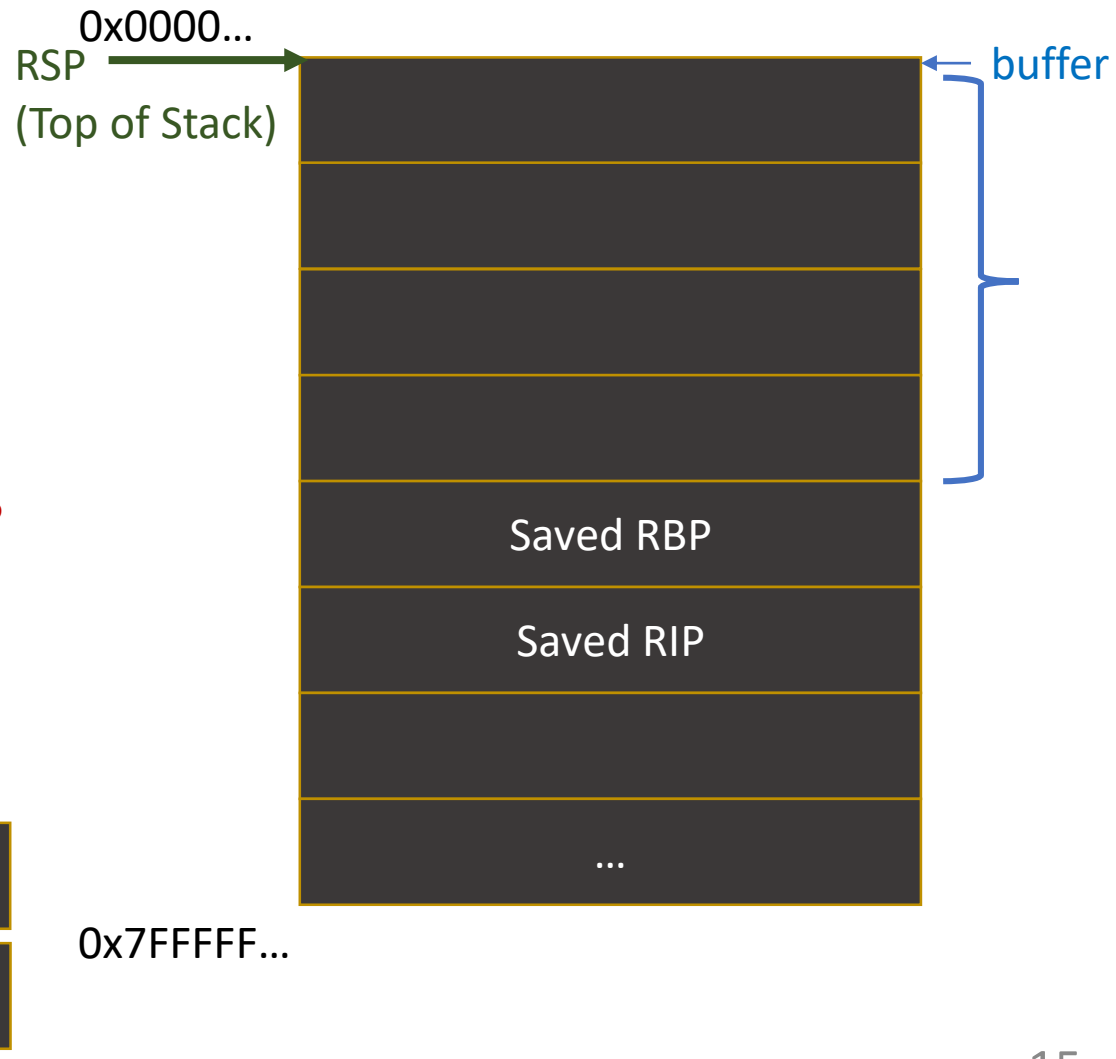
Program call:
> ./myprogram AAA… (56*A)

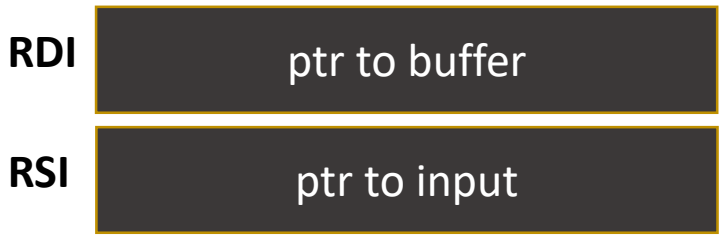
```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

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



# Recap Buffer Overflow

Program call:
> ./myprogram AAA… (56*A)

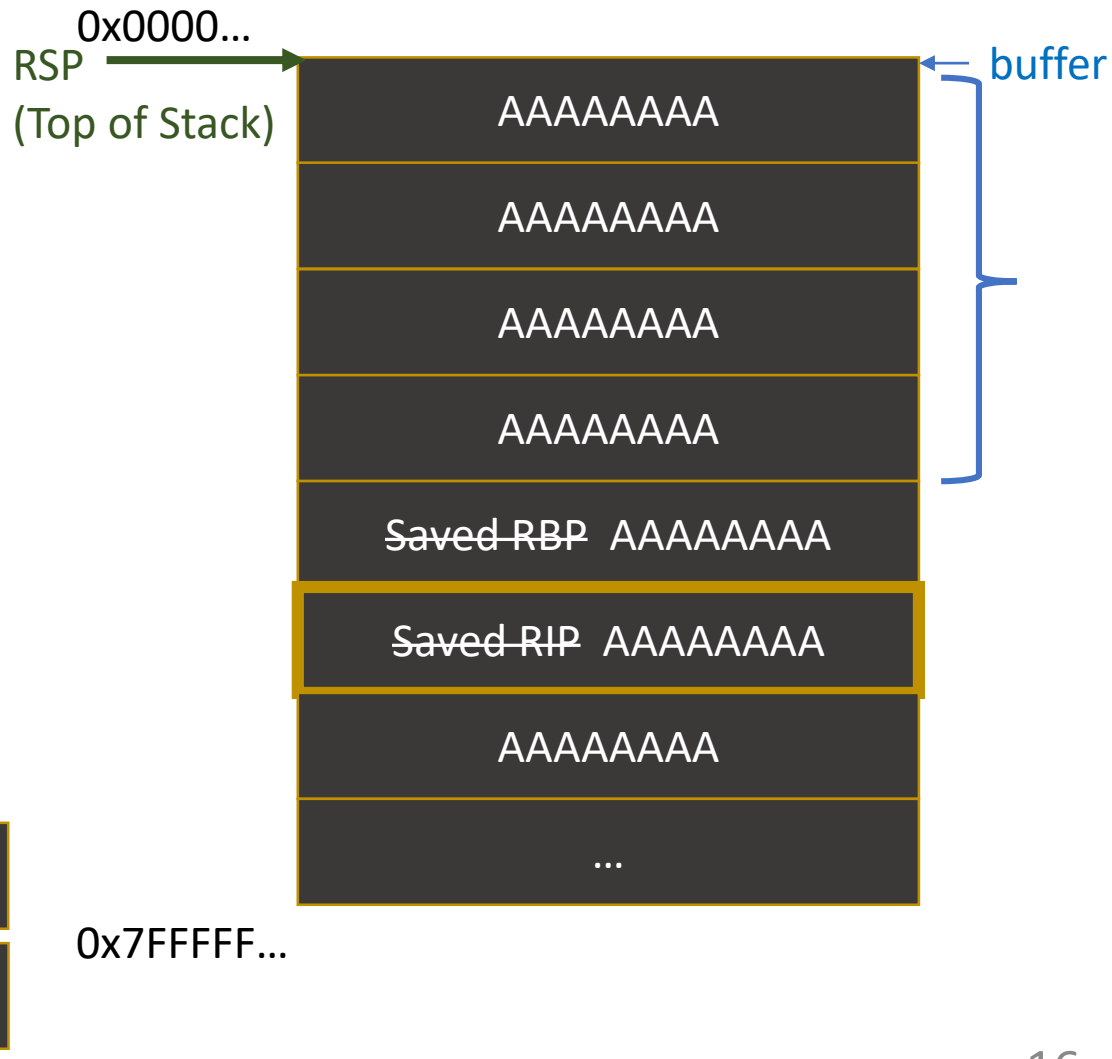
! Buffer Overflow, we have 32 bytes and write 56 bytes

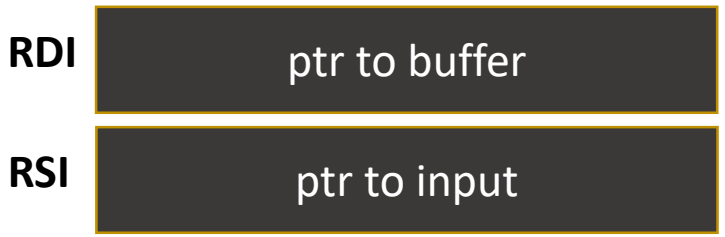
```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

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



Possible exploit: shellcode

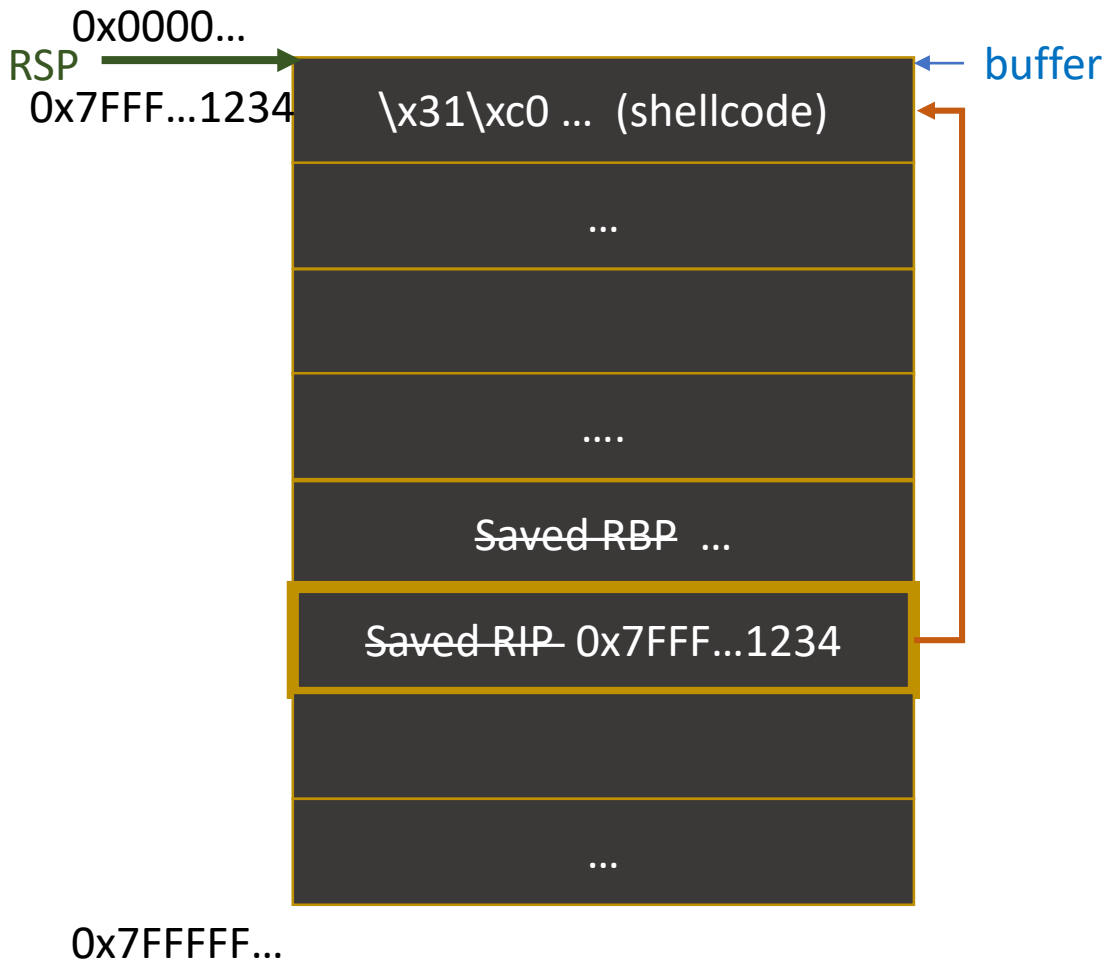
# Recap Buffer Overflow

```
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



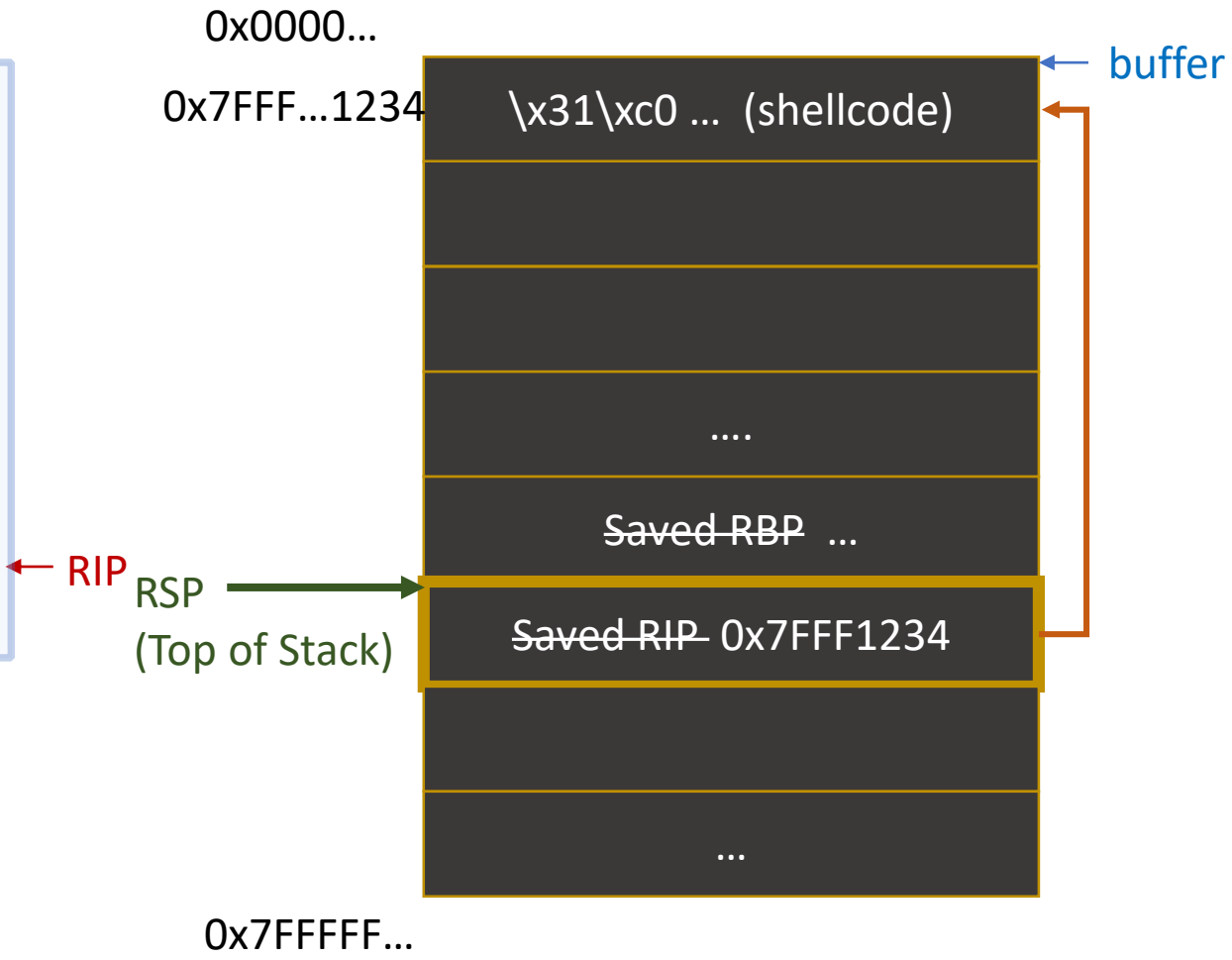
Possible exploit: shellcode

# 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<br>pop rbp | "pop rip" |

Possible exploit: shellcode

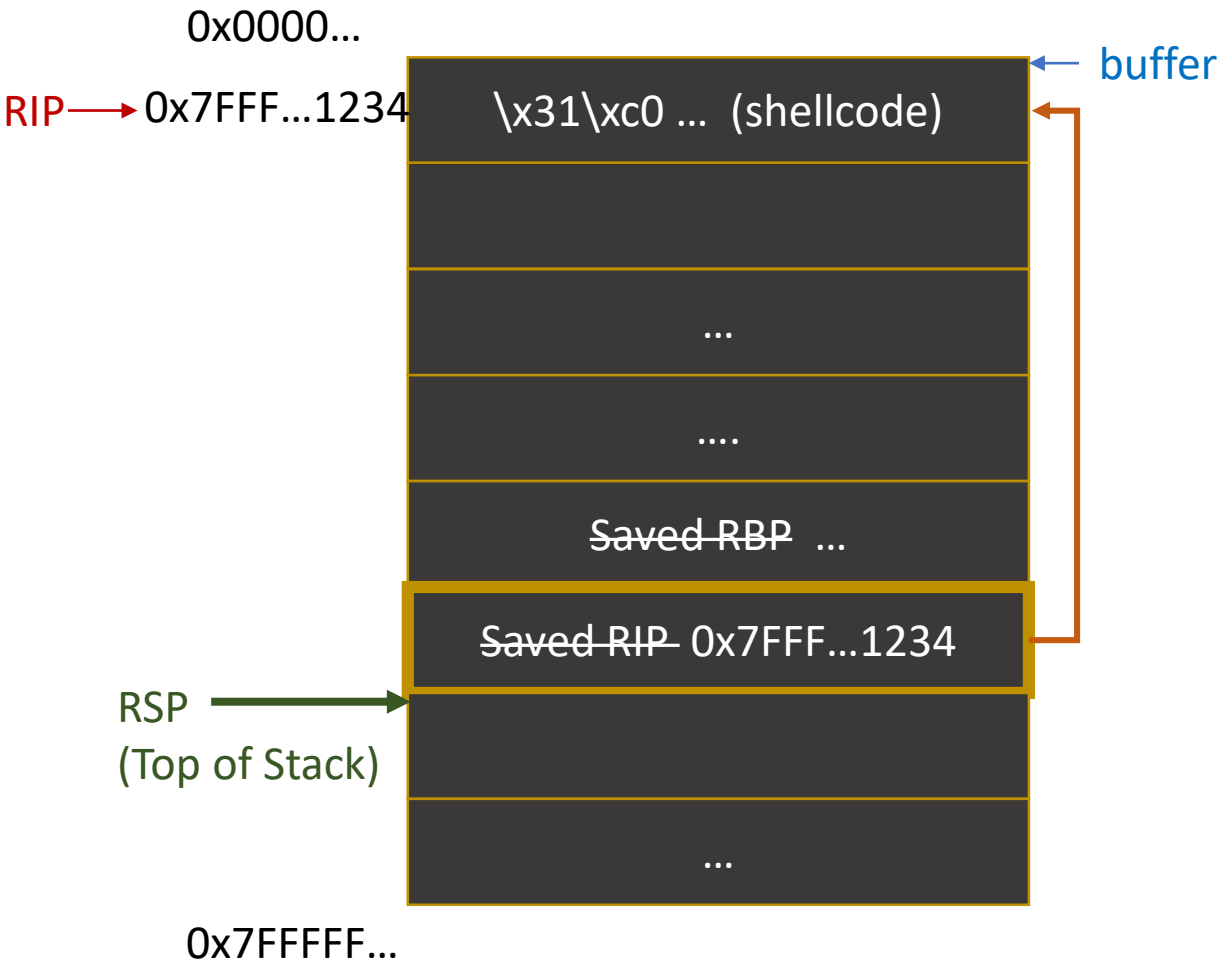
# Recap Buffer Overflow

```
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<br>pop rbp | "pop rip" |



# Patterns - RIP control

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

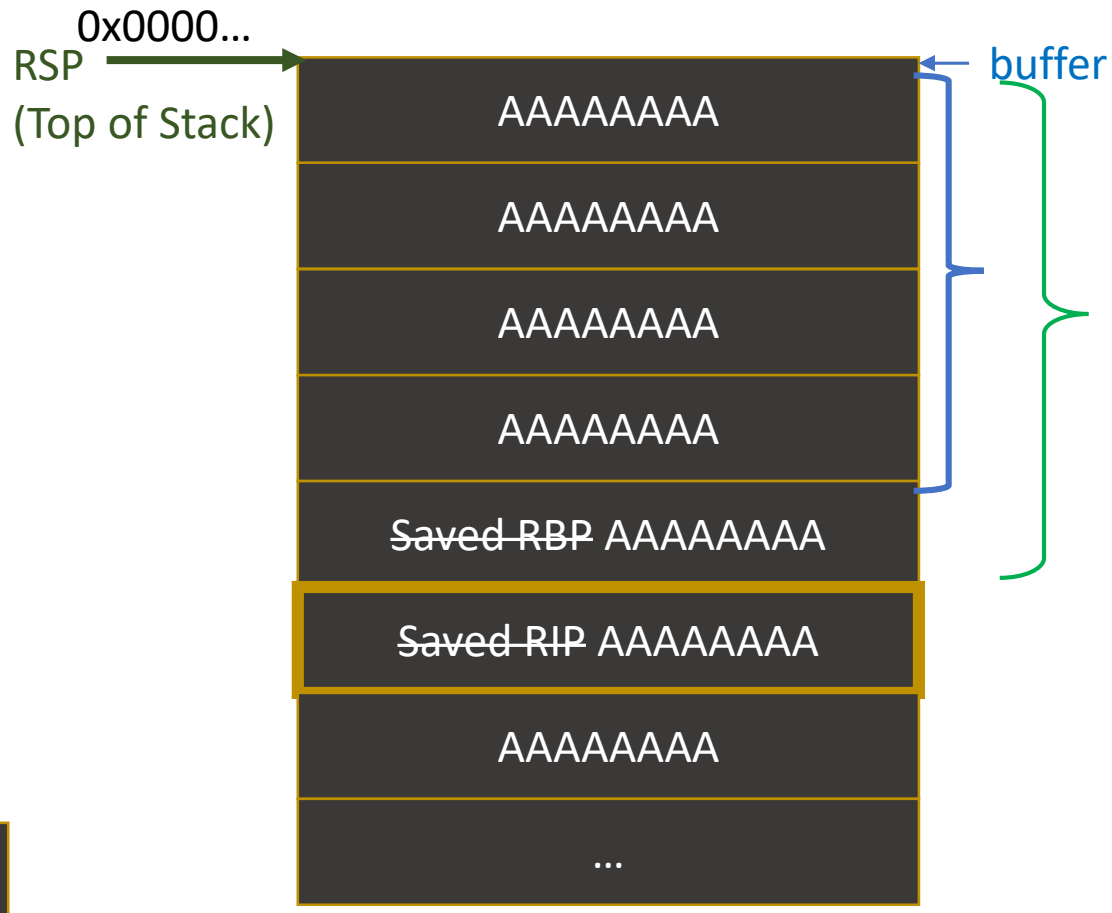
How many A's until we reach the saved RIP?

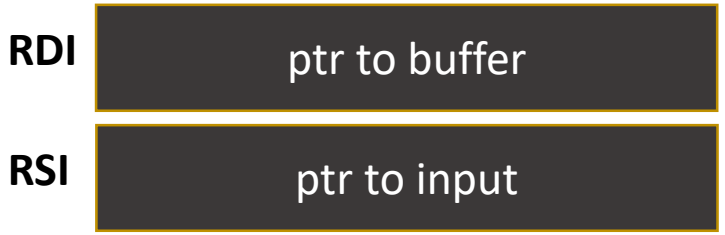
```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

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
```

← RIP



# Patterns - RIP control

Program call:
> ./myprogram AAAAAAAABBBBBBBBCCCCCCCC…

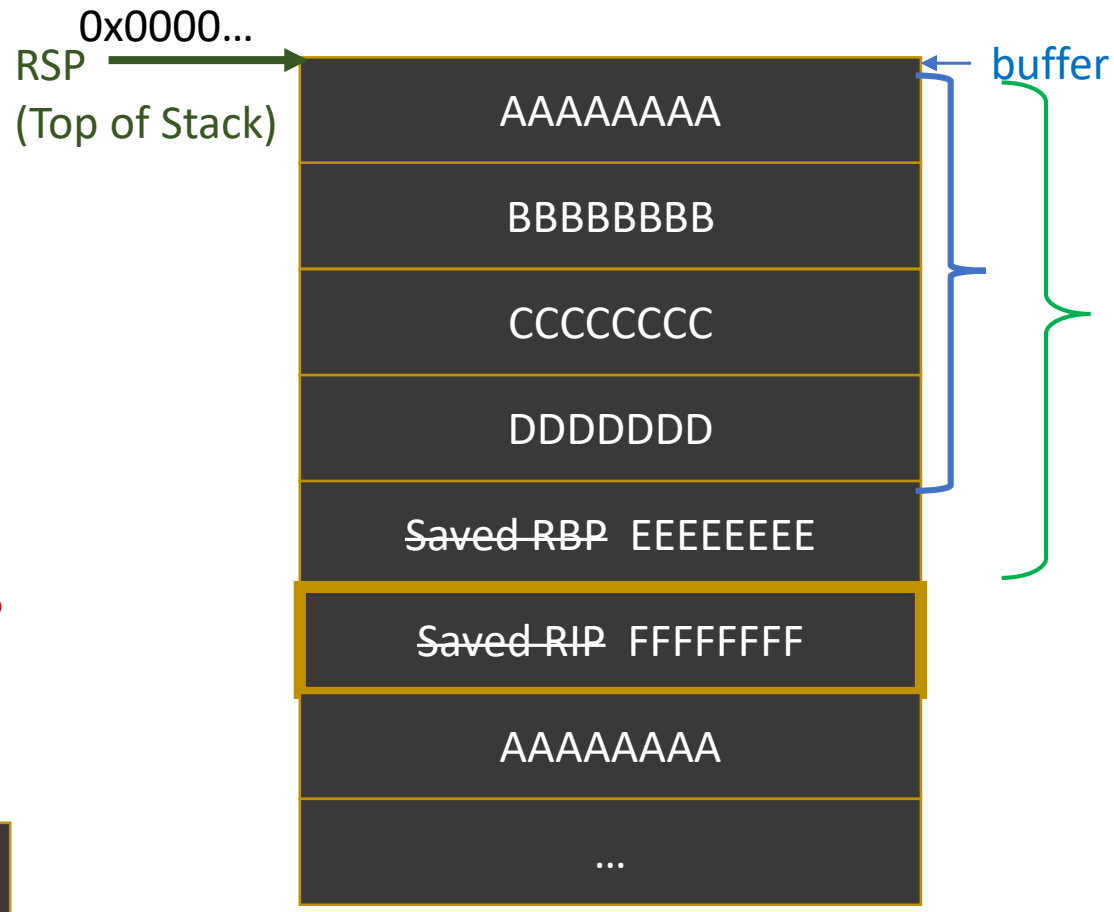
! Buffer Overflow, RIP is now FFFFFFFF

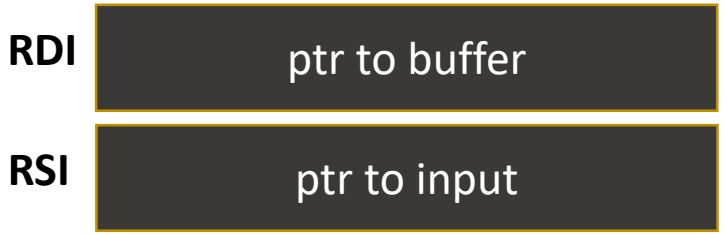
```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

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
```

← RIP



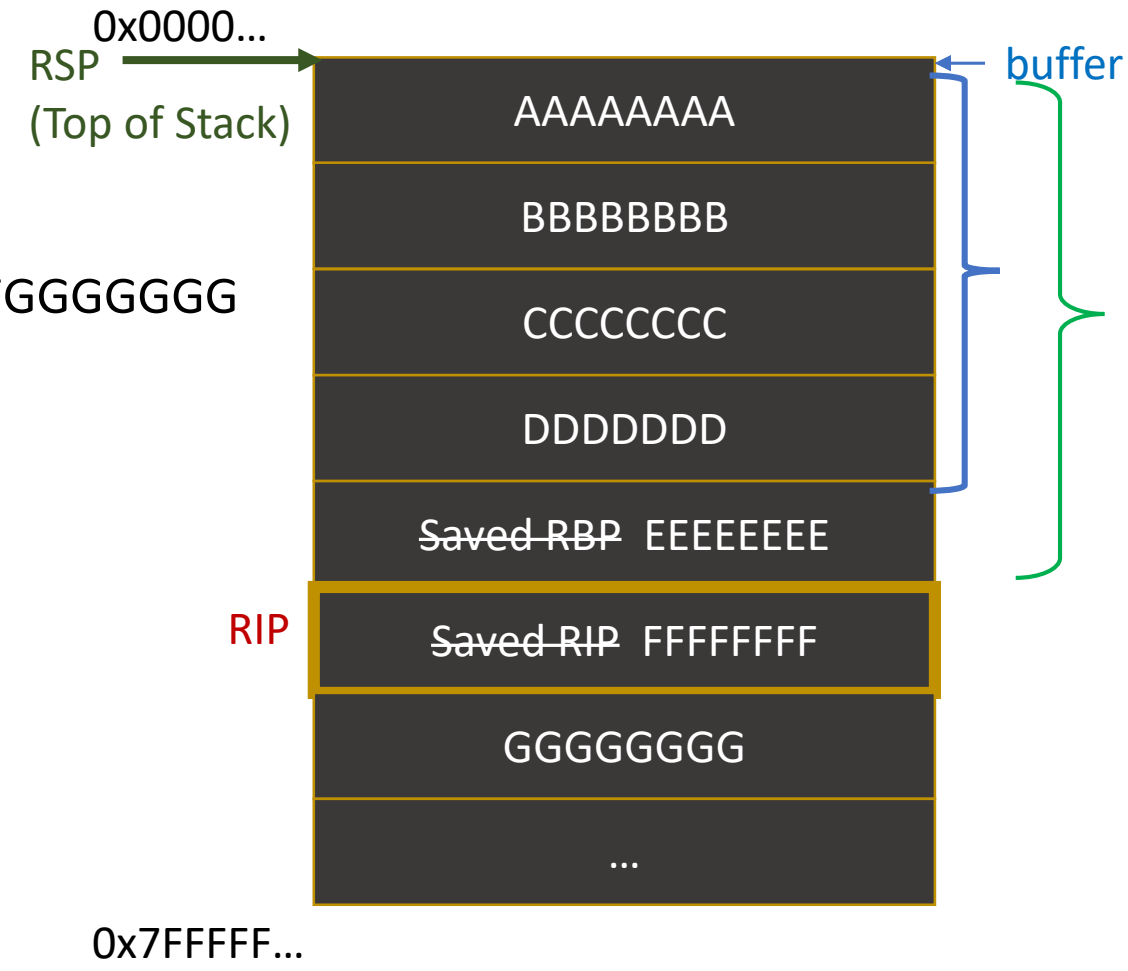
# Patterns - RIP control

Program call:
> ./myprogram AAAAAAAABBBBBBBBCCCCCCCC…

! Buffer Overflow, RIP is now FFFFFFFF

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

Input – String:
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFFGGGGGGG



# Patterns - RIP control

Program call:
> ./myprogram AAAAAAAABBBBBBBBCCCCCCCC…

! Buffer Overflow, RIP is now FFFFFFFF

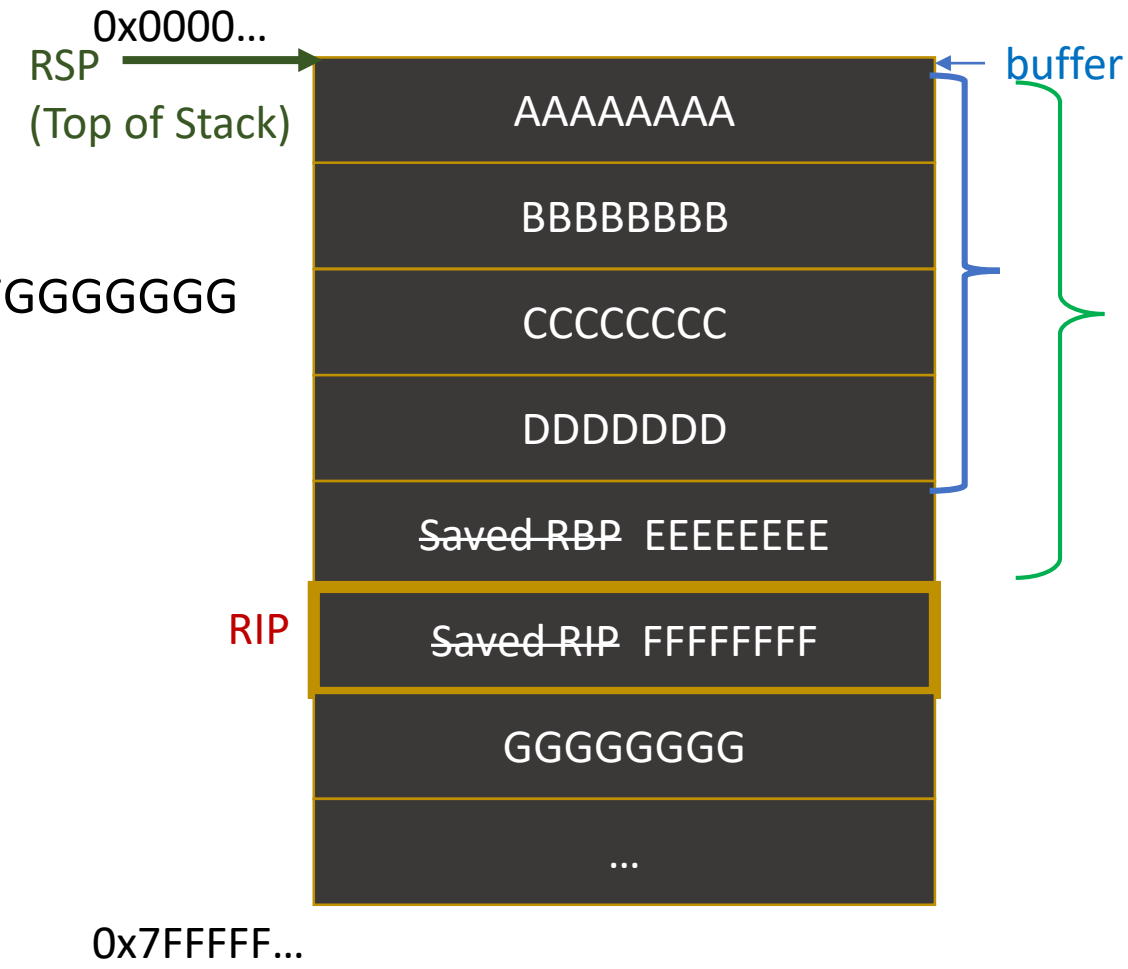
How many bytes do we have to write until we reach RIP?

Input – String:
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFFGGGGGGG

Fill-buff

RIP

Fill-buff = 8*"A" + 8 * "B" + 8*"C" + 8 * "D" + 8*"E"
= 40 Bytes

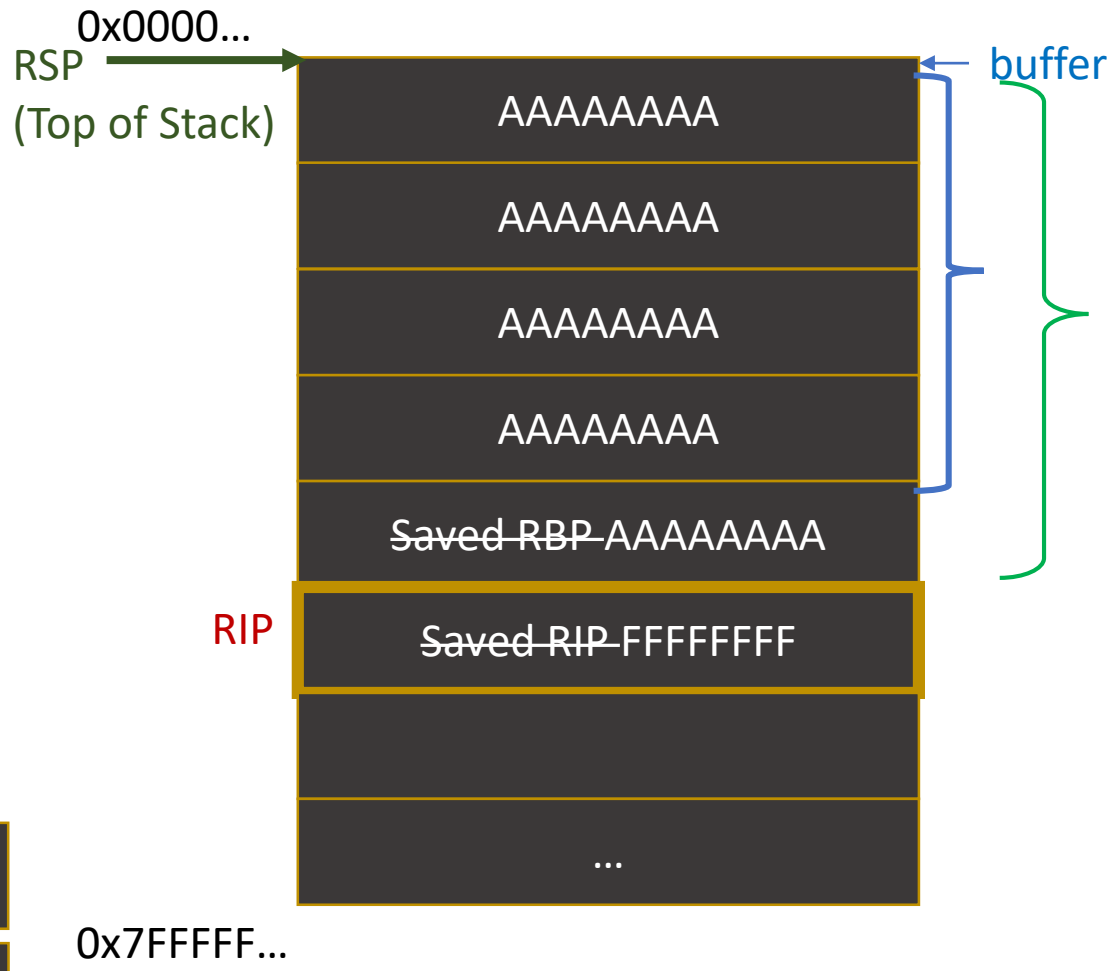


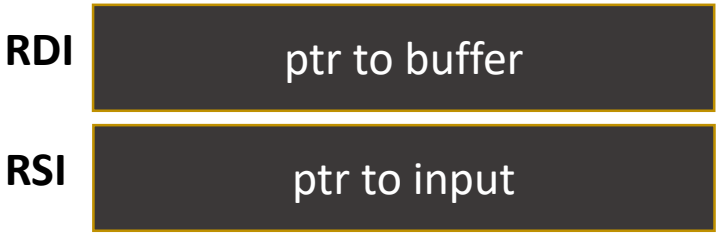
# Patterns - RIP control

Program call:
> ./myprogram AA…(A*40)FFFFFFFF
! Buffer Overflow, RIP is now FFFFFFFF

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

Input – String = 40*"A" + 8*"F"



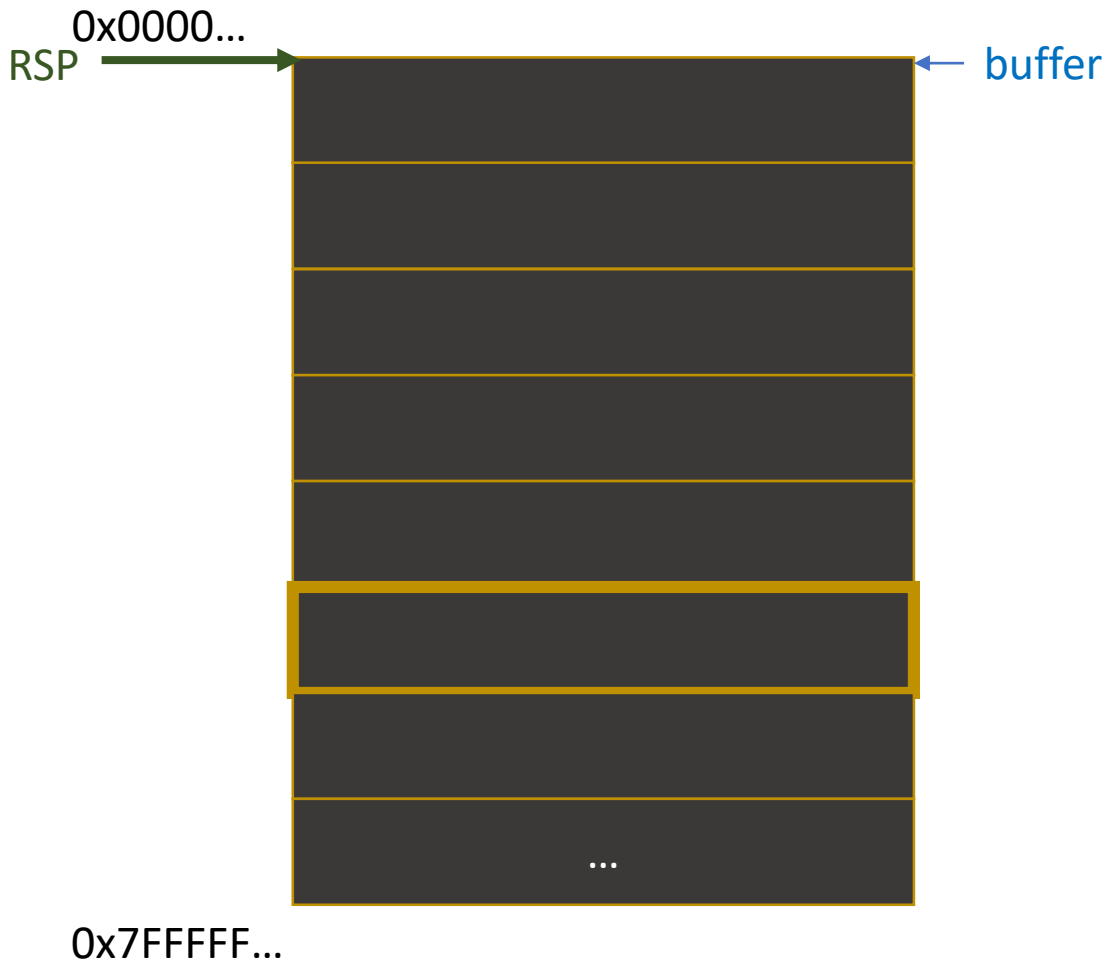



# Cyclic Patterns

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

1. Generate Pattern
   gdb-peda$ pattern create 60

   AAA%AAsAABAA$AAnAACAA-AA..



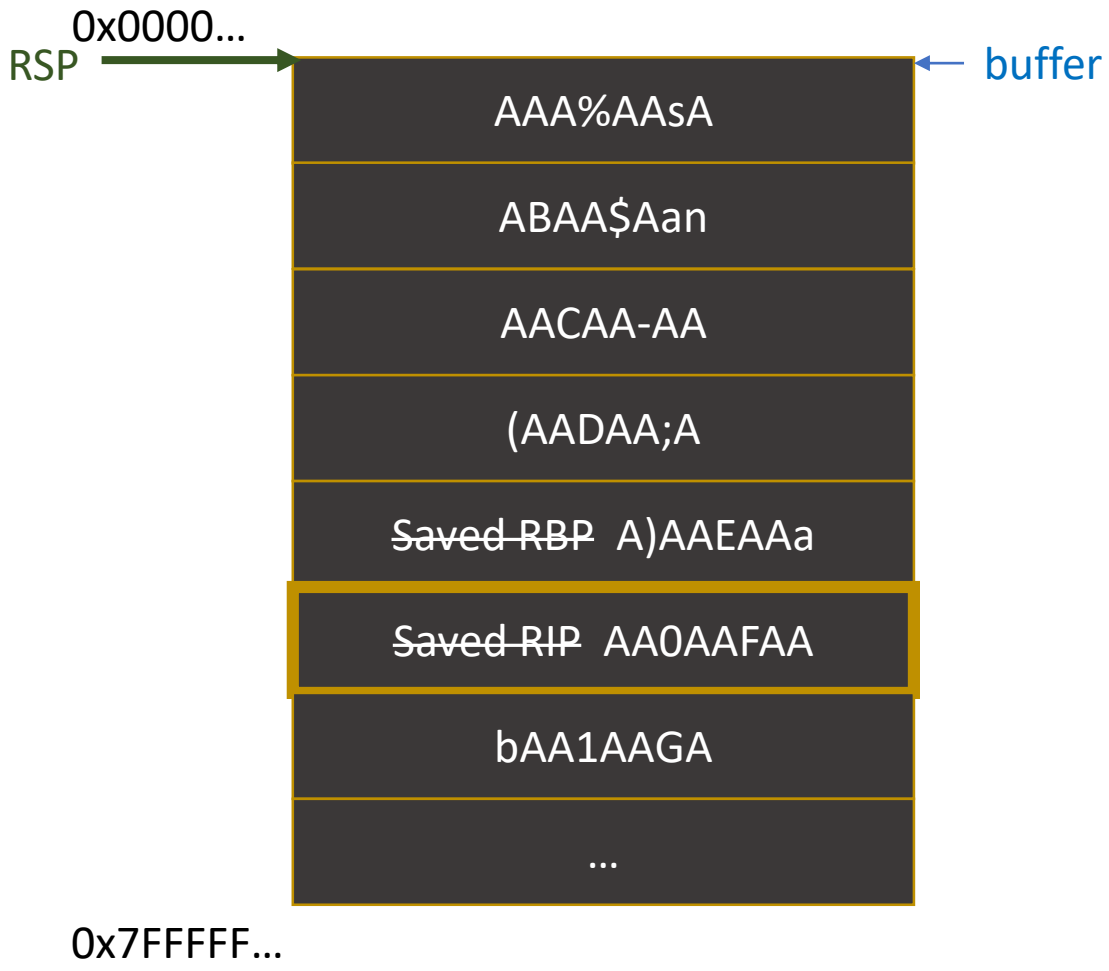
# Cyclic Patterns

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

1. Generate Pattern
   gdb-peda$ pattern create 60

   AAA%AAsAABAA$AAnAACAA-AA

2. Pattern as input for the program
   gdb-peda$ run AAA%AAsAABAA$AAnAACAA-AA



# Cyclic Patterns

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

1. Generate Pattern
   gdb-peda$ pattern create 60

   AAA%AAsAABAA$AAnAACAA-AA

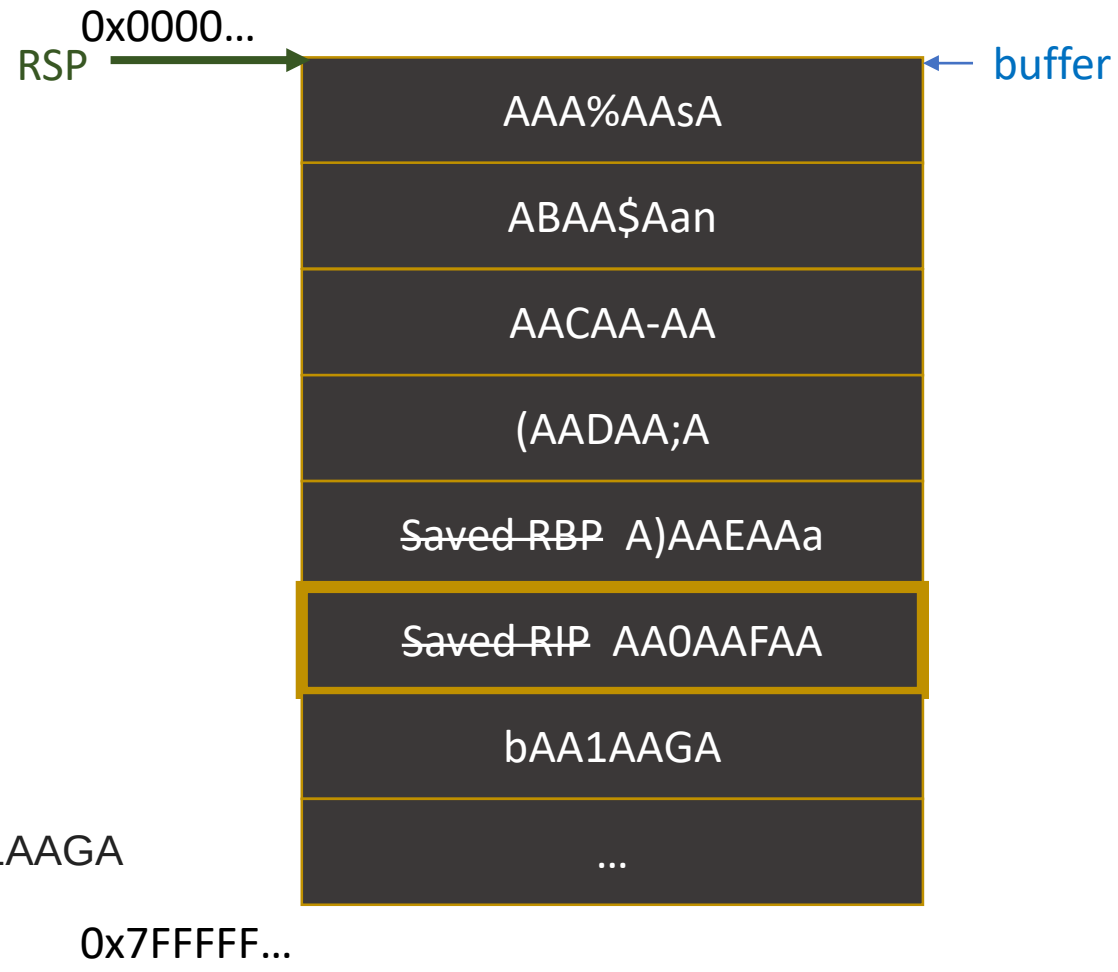
2. Pattern as input for the program
   gdb-peda$ run AAA%AAsAABAA$AAnAACAA-AA

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$AAnAACAA-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$AAnAACAA-AA

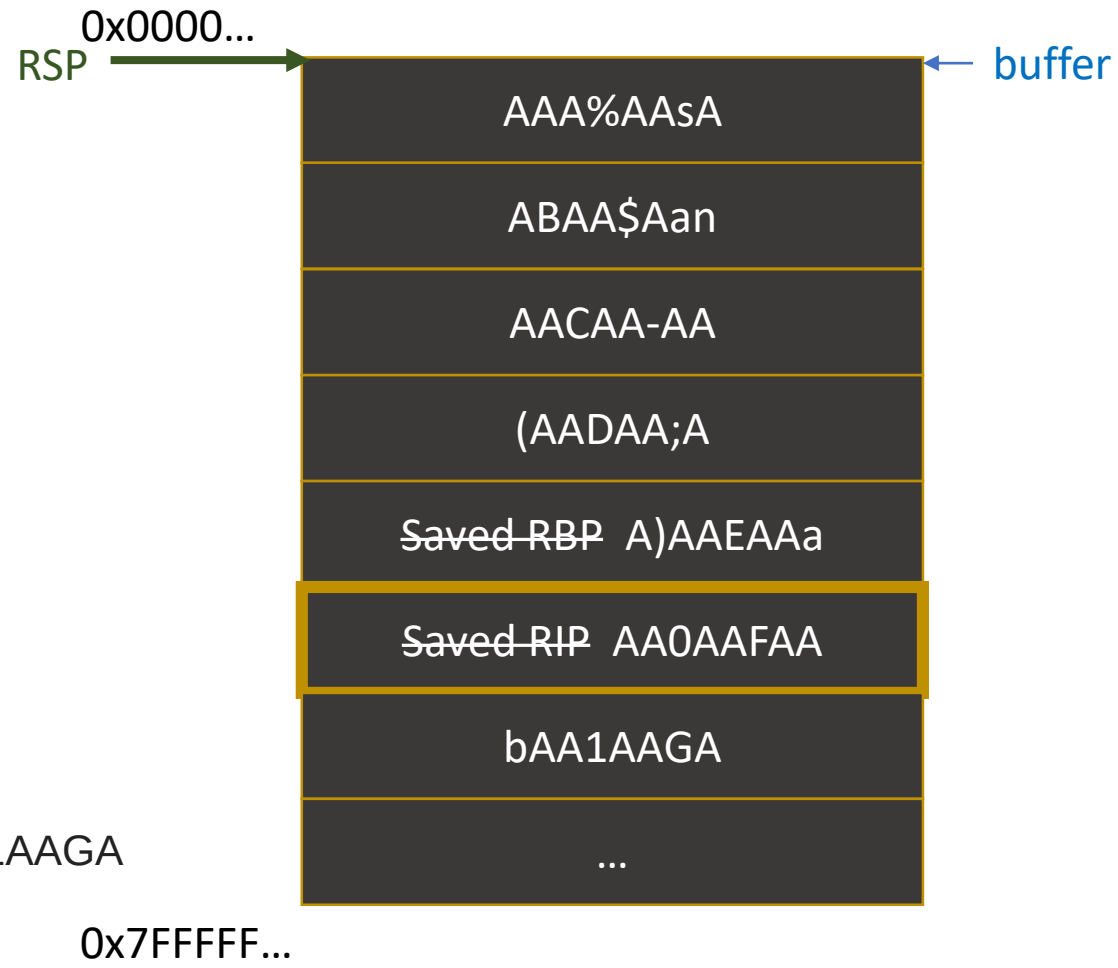
2. Pattern as input for the program
   gdb-peda$ run AAA%AAsAABAA$AAnAACAA-AA

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$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGA

AA0AAFAA



# Cyclic Patterns

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

1. Generate Pattern
   gdb-peda$ pattern create 60

   AAA%AAsAABAA$AAnAACAA-AA

2. Pattern as input for the program
   gdb-peda$ run AAA%AAsAABAA$AAnAACAA-AA

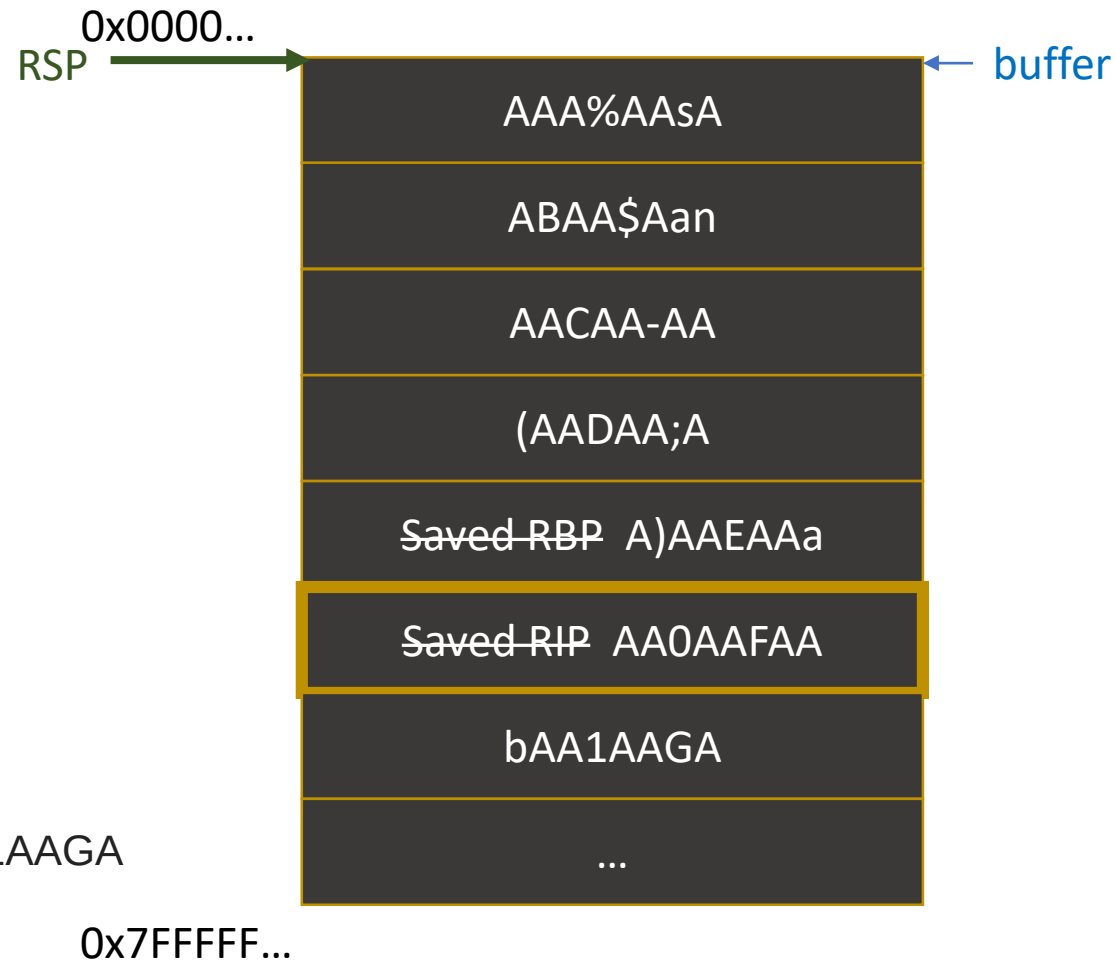
3. Find the part in the pattern that overwrote RIP
   gdb-peda$ pattern offset **AA0AAFAA**

Gdb-peda internally uses pattern matching for that:

AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGA

AA0AAFAA

Match!
Offset: 40



# Cyclic Patterns

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

1. Generate Pattern
   gdb-peda$ pattern create 60

   AAA%AAsAABAA$AAnAACAA-AA

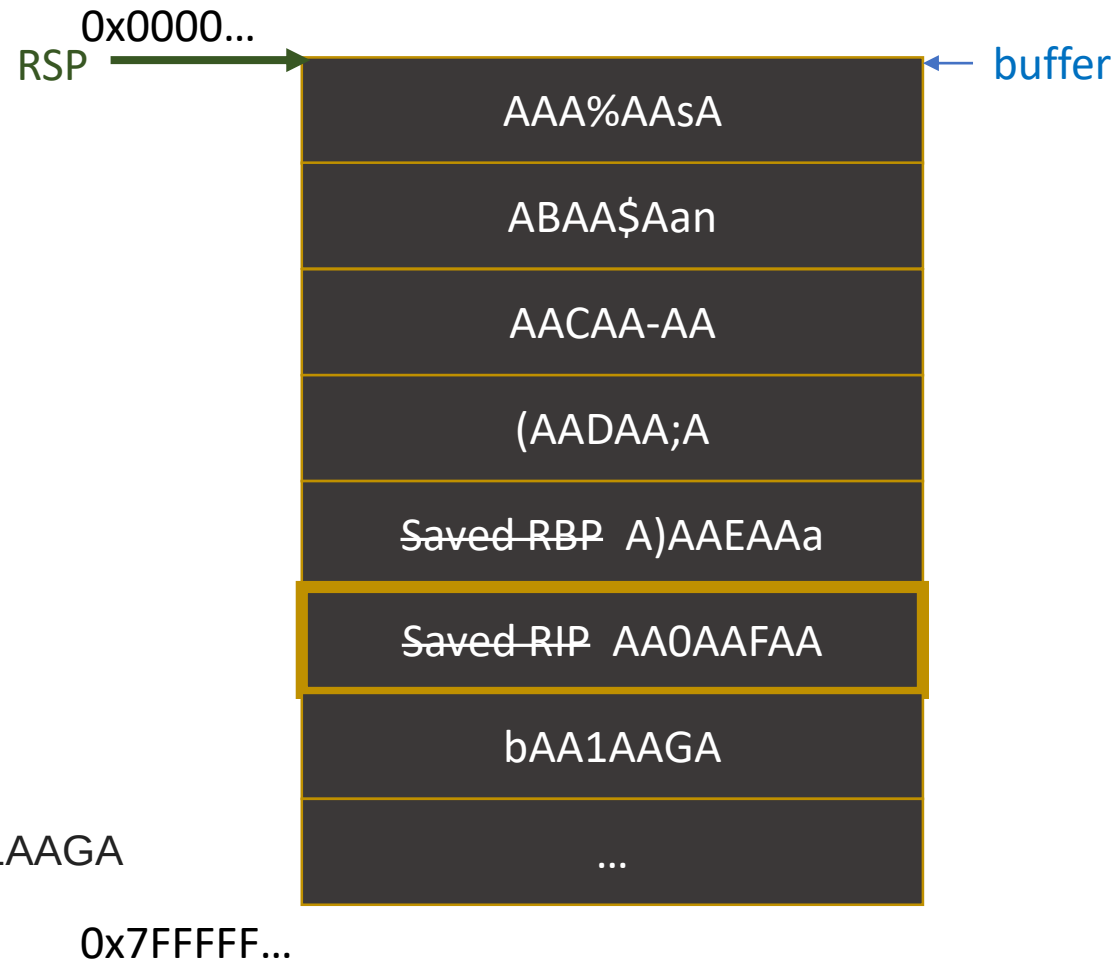
2. Pattern as input for the program
   gdb-peda$ run AAA%AAsAABAA$AAnAACAA-AA

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$AAnAACAA-AA(AADAA;AA)AAEAAa**AA0AAFAA**bAA1AAGA

Match!
Offset: 40



# 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

```c
#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

```c
#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

```c
#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 libray) 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 libarary)
  => 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



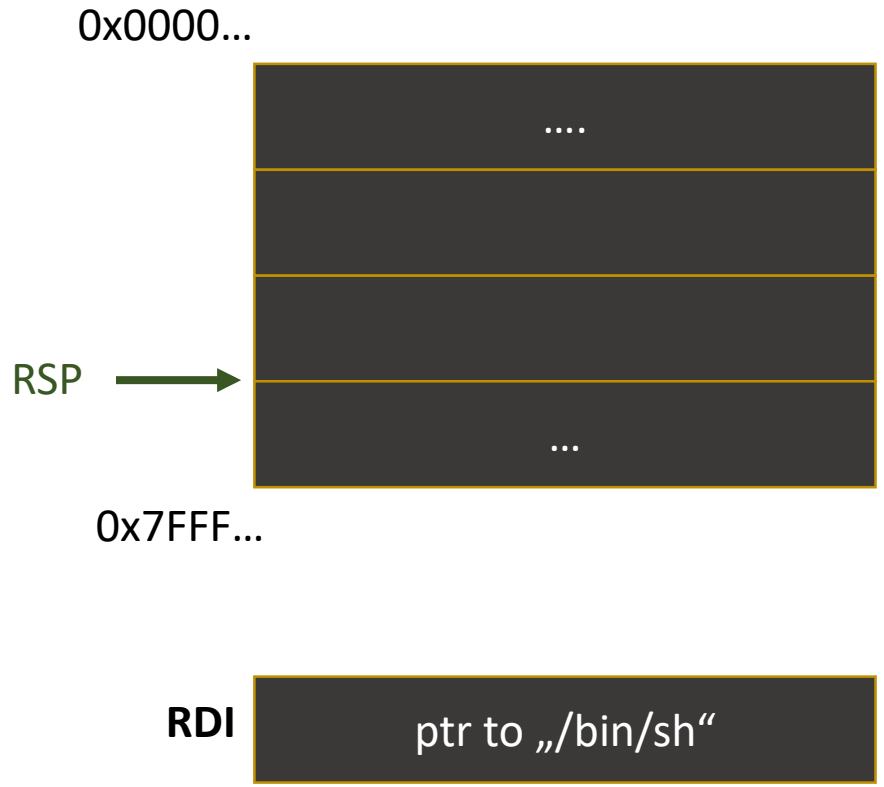
# 64 Bit – Calling convention

system("/bin/sh")

```
.binsh:
.string "/bin/sh"

main :
        mov rdi, OFFSET.binsh
        call system
```

RIP



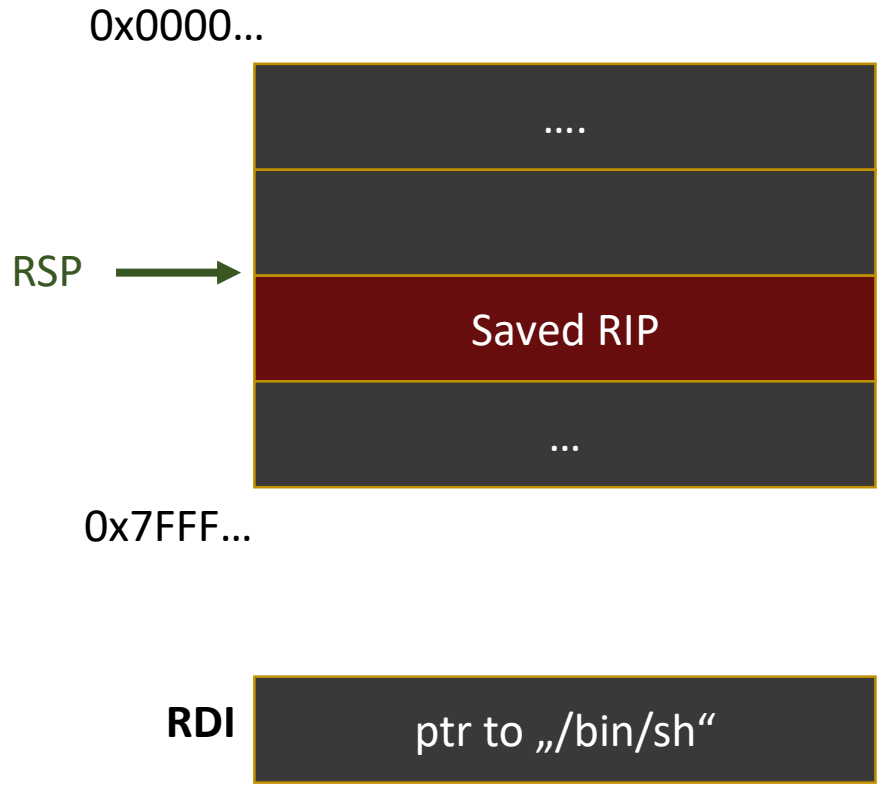
# 64 Bit – Calling convention

system("/bin/sh")

```
.binsh:
.string "/bin/sh"

main :
        mov rdi, OFFSET.binsh
        call system
```

← RIP

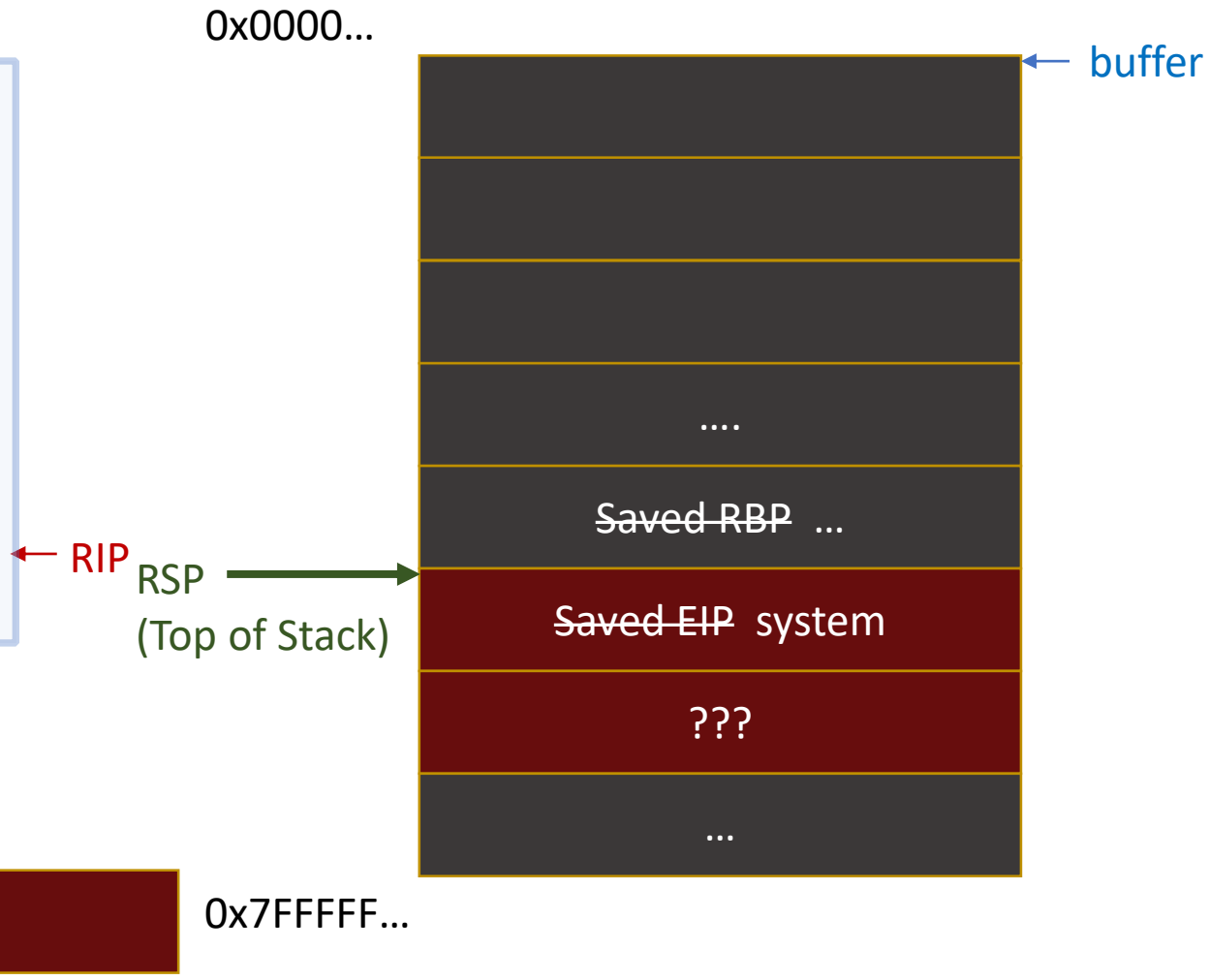


# Payload strategy

```
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: | Ret: |
|---|---|
| mov rsp, rbp<br>pop rbp | "pop rip" |

# Building ROP chains…


HACKERS CAN TURN YOUR HOME COMPUTER INTO A BOMB

By RANDY JEFFRIES / Weekly World News

WASHINGTON — Right now, computer hackers have the ability to turn your home computer into a bomb and blow you to Kingdom Come — and they can do it anonymously from

[1]

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

# Building ROP chains…

[1]

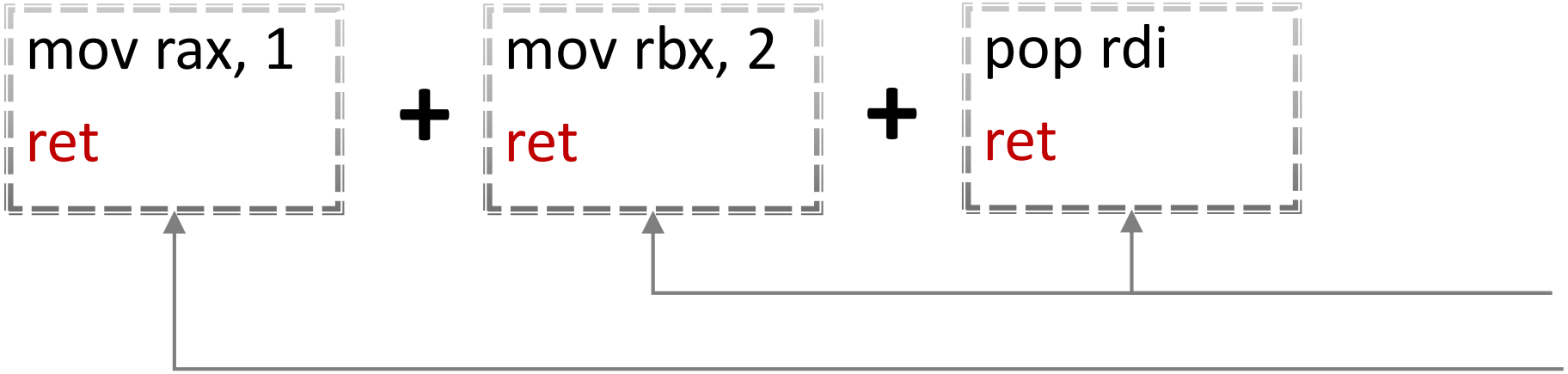

- Take snippets from the binary
- glue them together
- get the wanted code

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

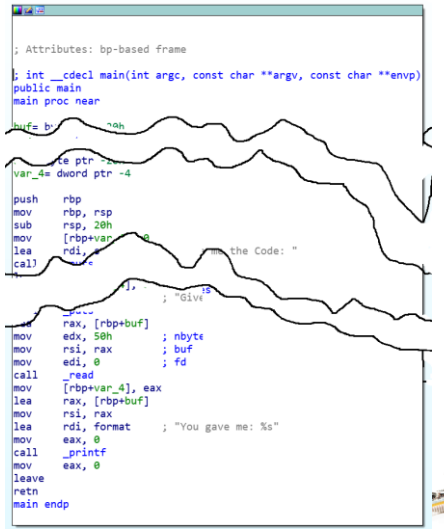
# Building ROP chains…

[1]


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




vuln_binary




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

# Building ROP-chains …

( „ret“ = pop RIP)

0x400111

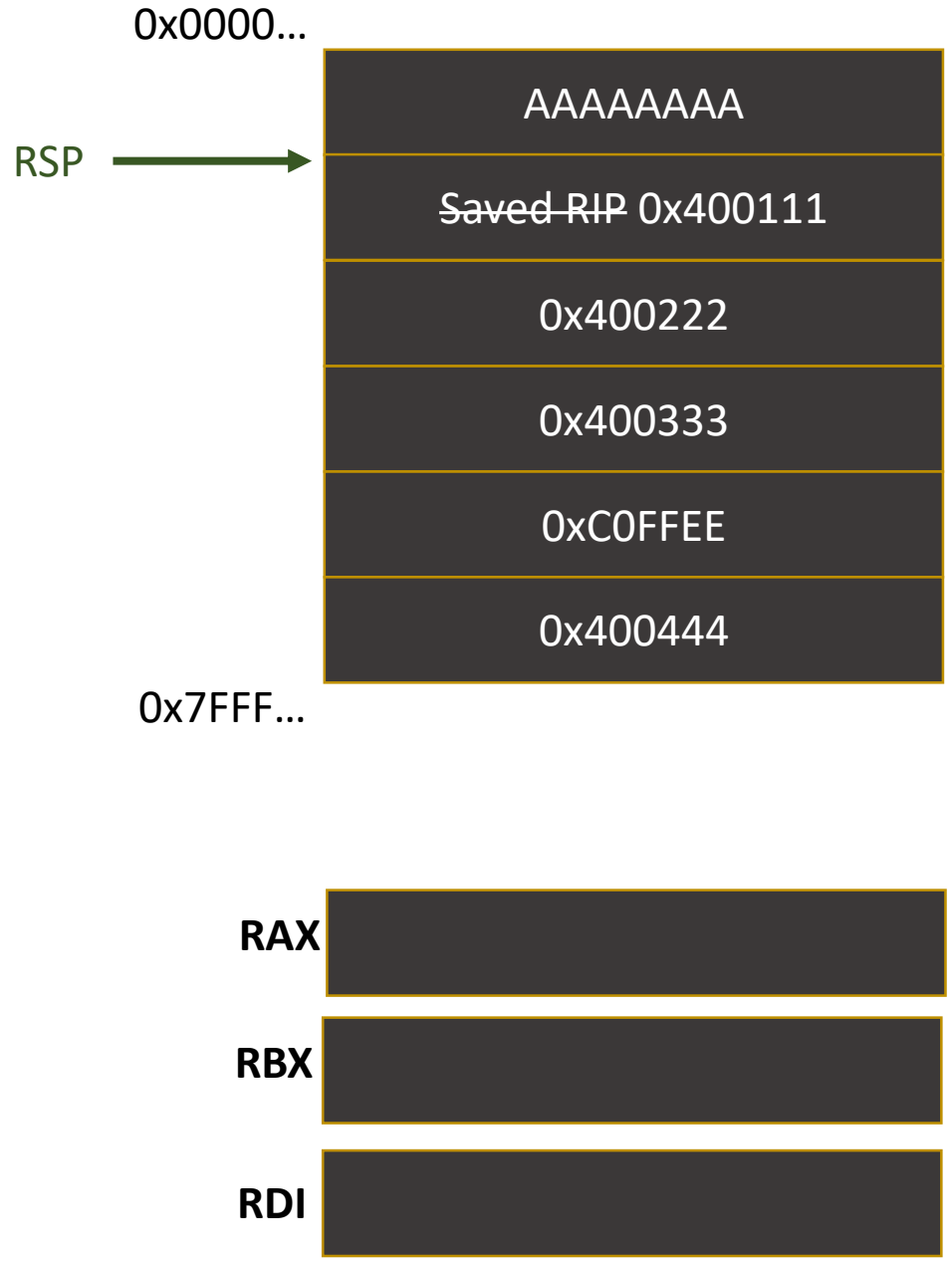
```
mov rax, 1
ret
```

0x400222

```
mov rbx, 2
ret
```

0x400333

```
pop rdi
ret
```



# Building ROP-chains …

0x400111

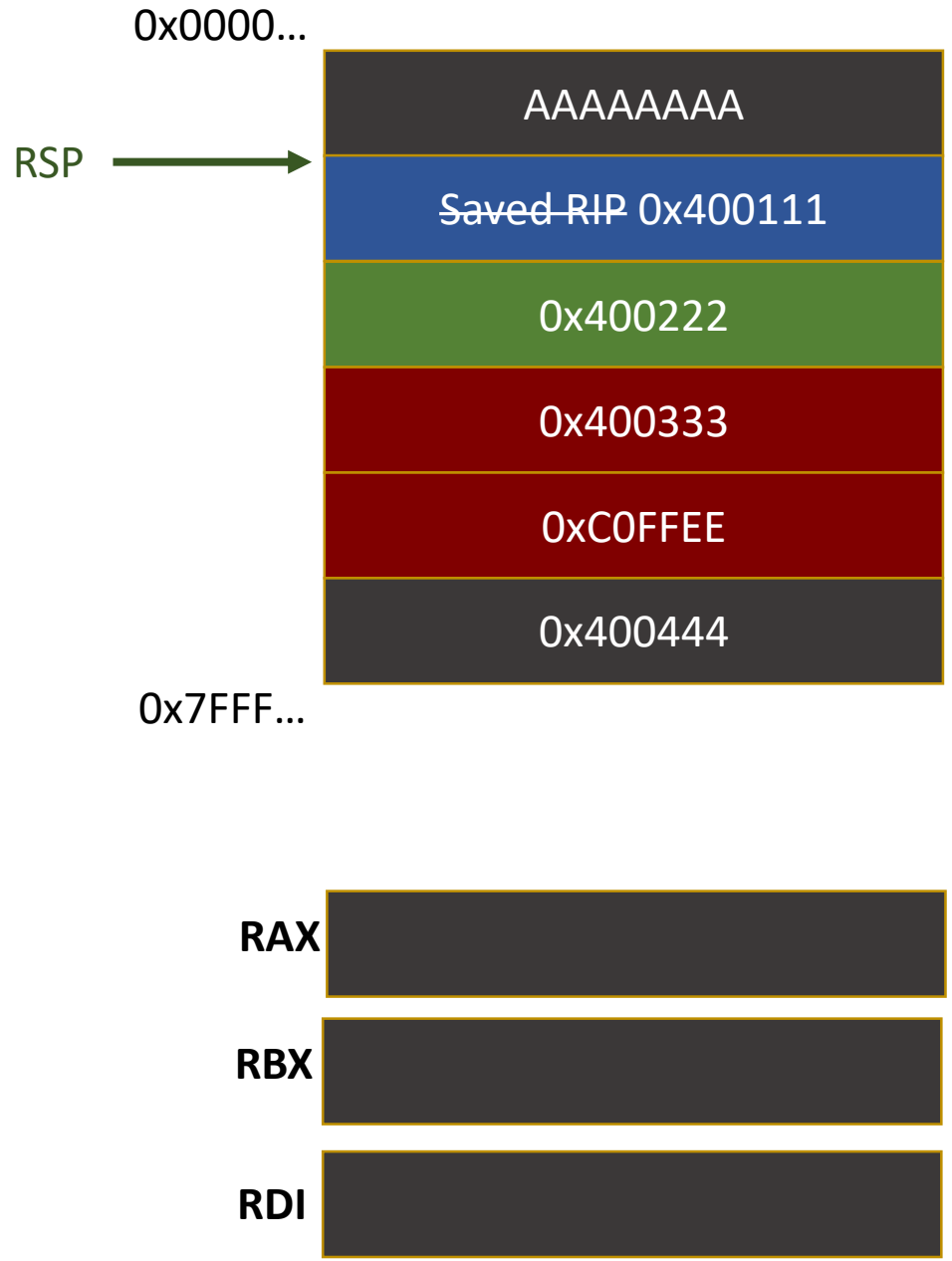
```
mov rax, 1
ret
```

0x400222

```
mov rbx, 2
ret
```

0x400333

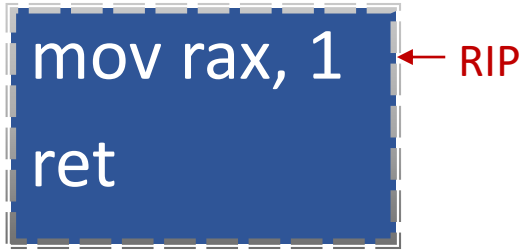
```
pop rdi
ret
```



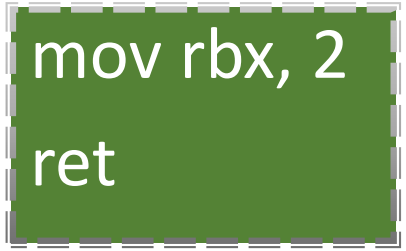
# Building ROP-chains …

( „ret“ = pop RIP)

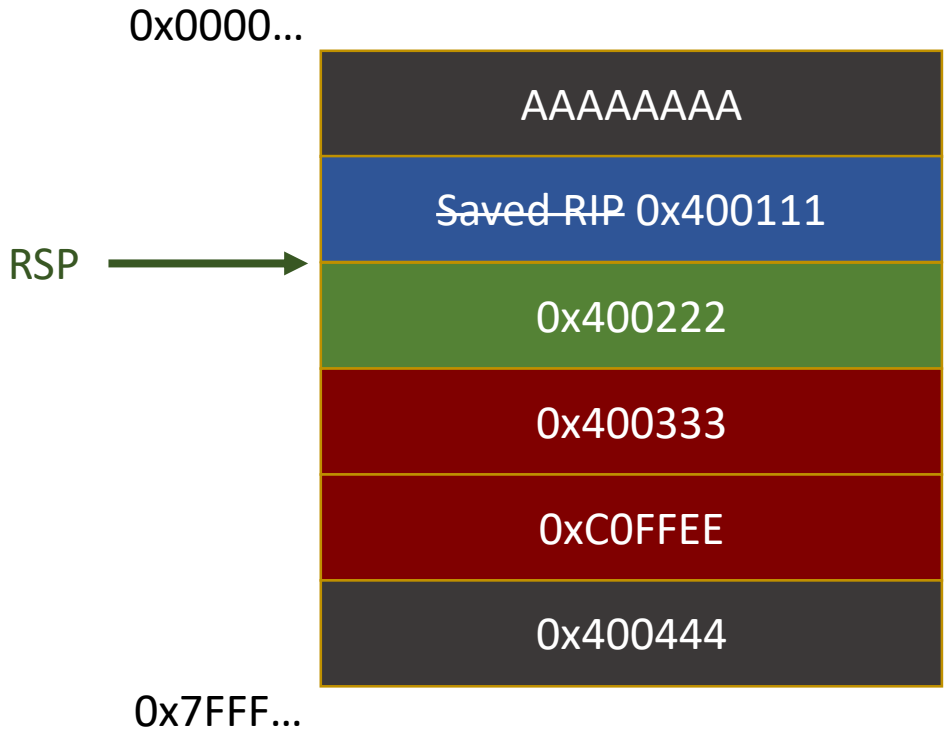
0x400111




0x400222




0x400333


# Building ROP-chains …

( „ret“ = pop RIP)

0x400111

```
mov rax, 1
ret
```

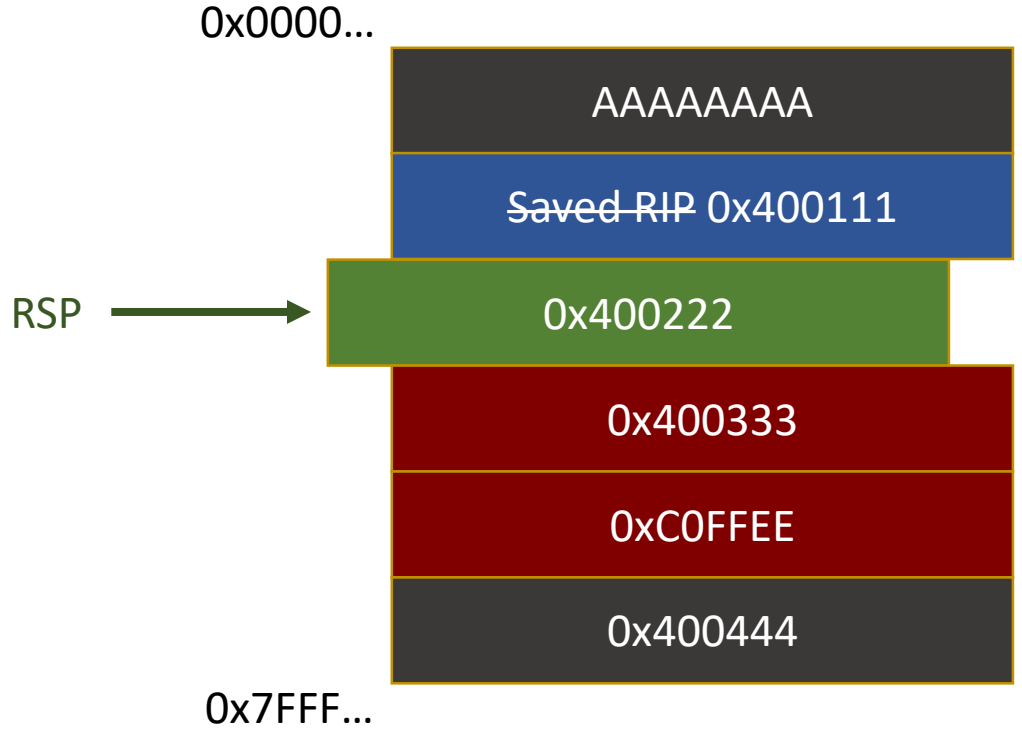
← RIP

0x400222

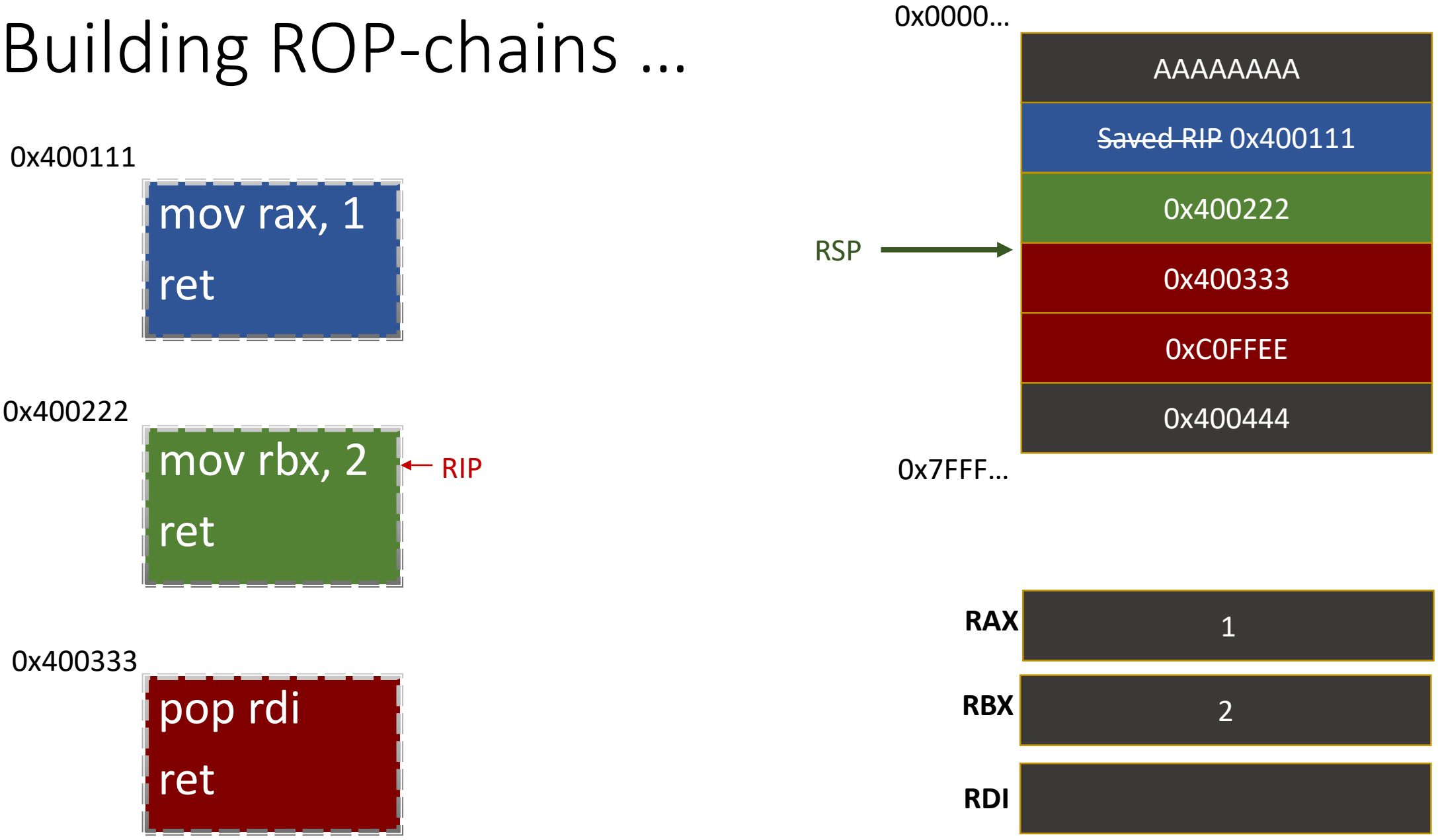
```
mov rbx, 2
ret
```

0x400333

```
pop rdi
ret
```




# Building ROP-chains …



# Building ROP-chains …

( „ret“ = pop RIP)

0x400111

```
mov rax, 1
ret
```

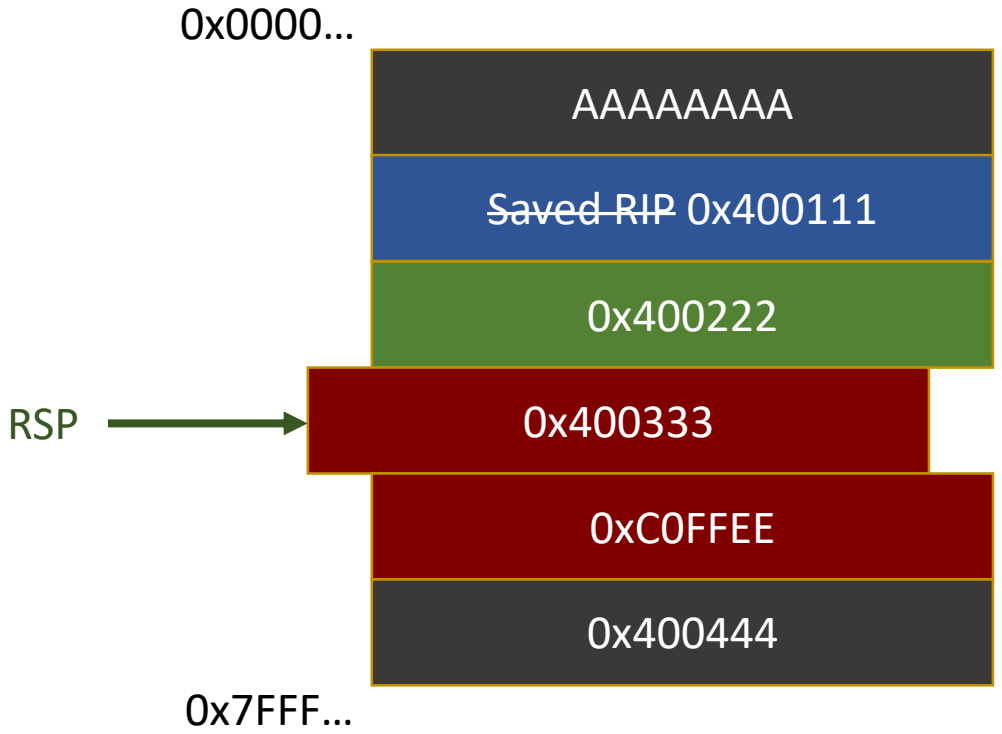
0x400222

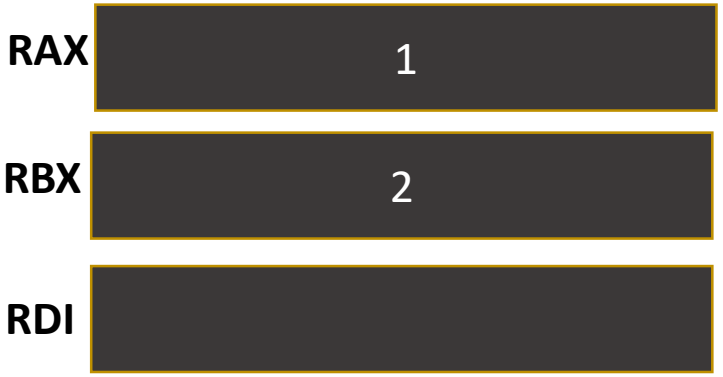
```
mov rbx, 2
ret
```

← RIP

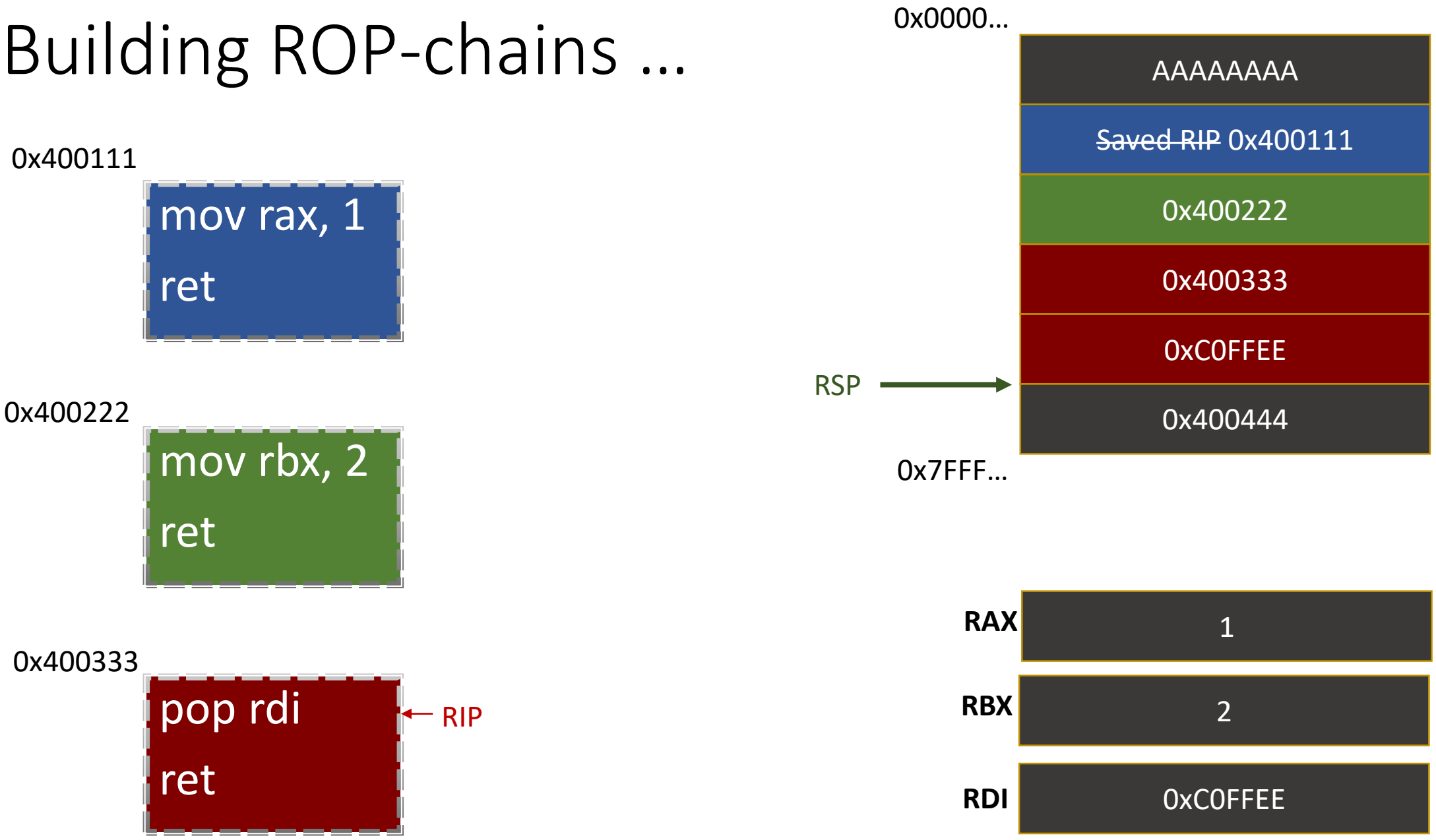
0x400333

```
pop rdi
ret
```



# Building ROP-chains …



# x86 – ROPgadget

Why is there code we don‘t see while disassembling?



# x86 – ROPgadget

Why is there code we don‘t see while disassembling?

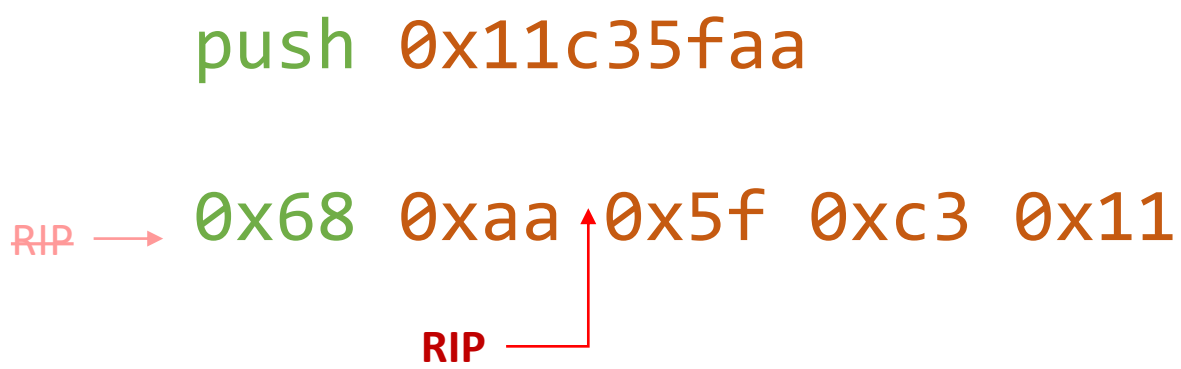
```
push 0x11c35faa

RIP → 0x68 0xaa 0x5f 0xc3 0x11
```



# x86 – ROPgadget

Why is there code we don‘t see while disassembling?



# x86 – ROPgadget

Why is there code we don‘t see while disassembling?

```
push 0x11c35faa

0x68 0xaa 0x5f 0xc3 0x11

RIP → pop rdi; ret
```

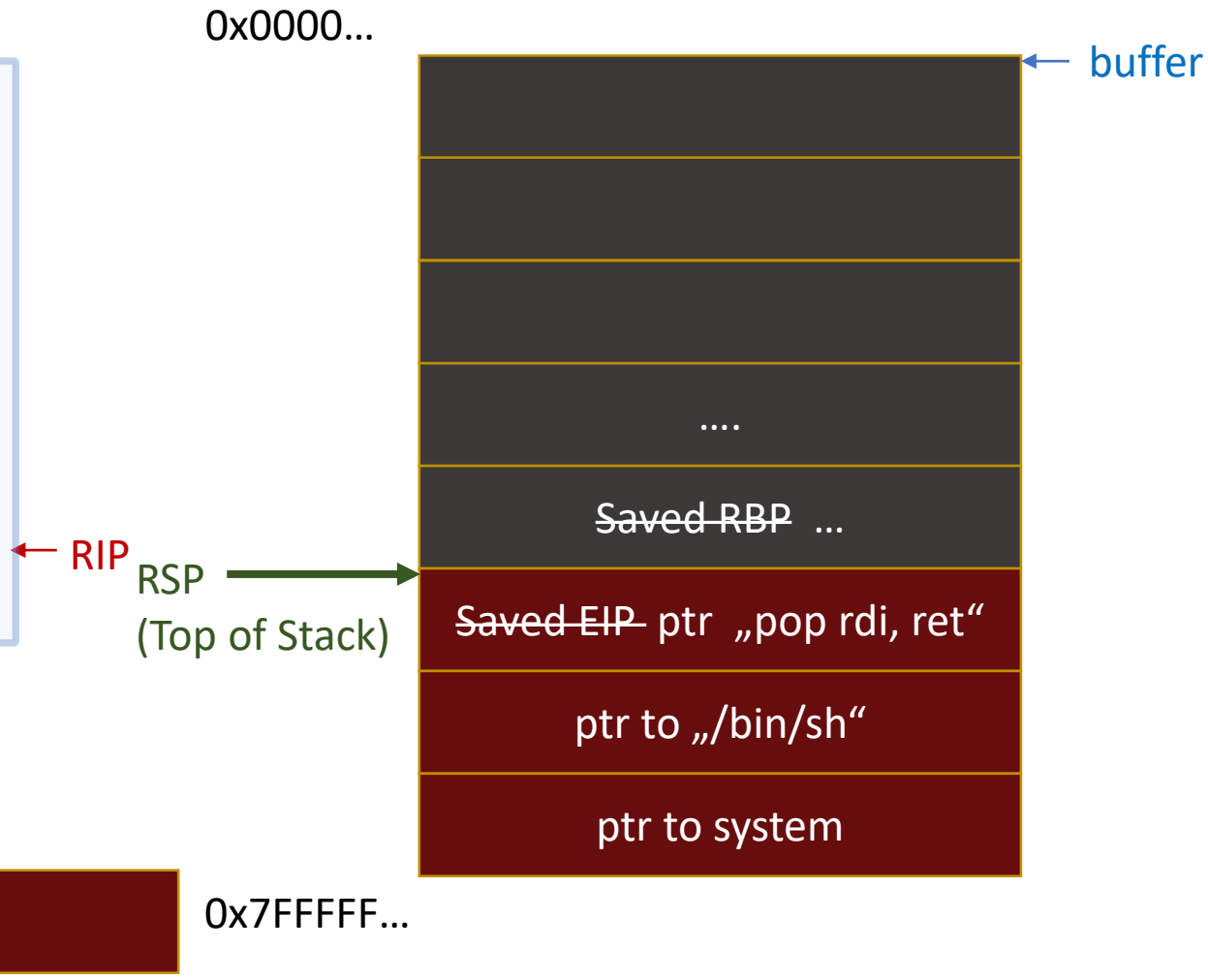


# Payload strategy

```
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
```



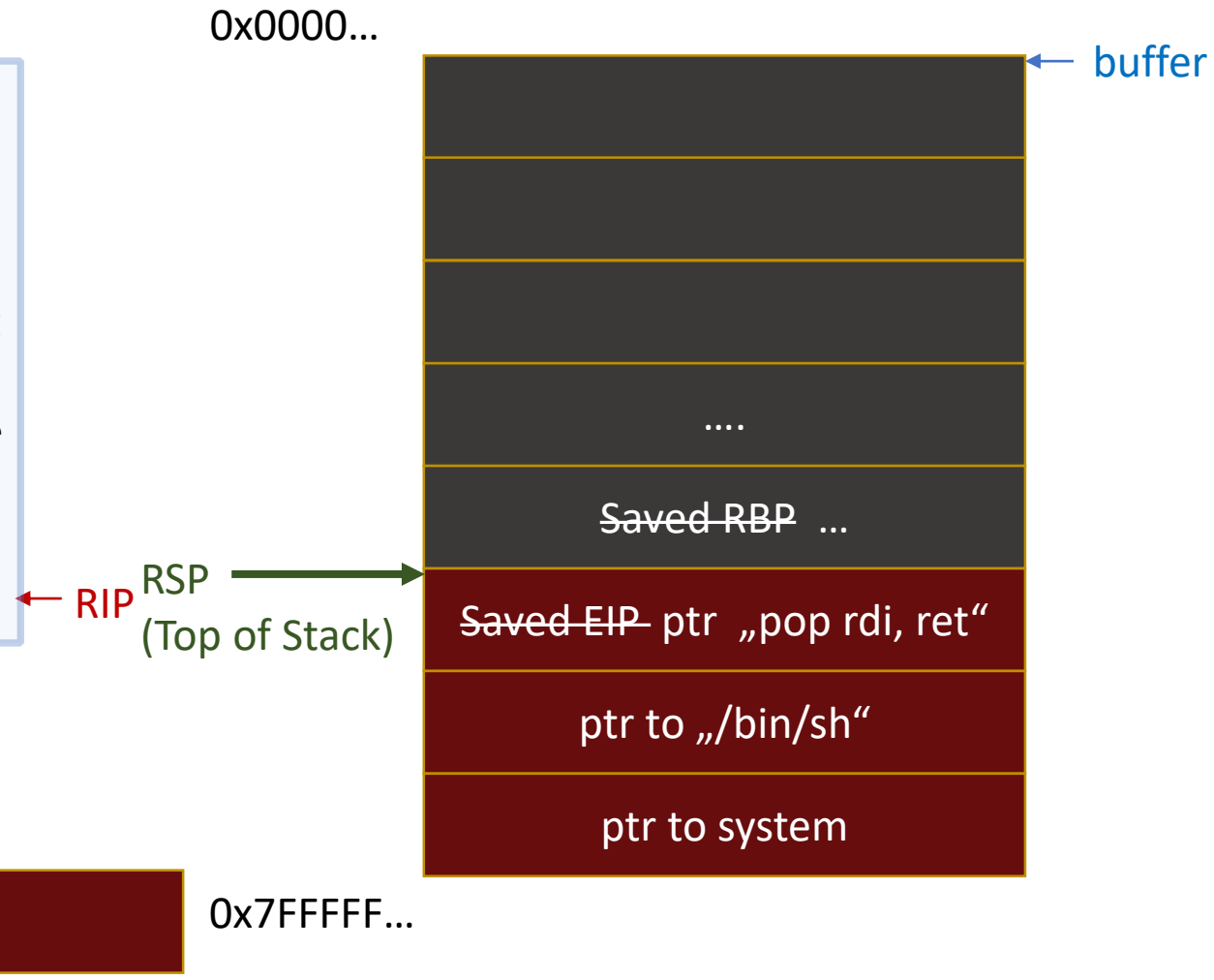

| Leave: | Ret: |
|---|---|
| mov rsp, rbp<br>pop rbp | "pop rip" |

# 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: | Ret: |
|---|---|
| mov rsp, rbp<br>pop rbp | "pop rip" |

# Payload strategy

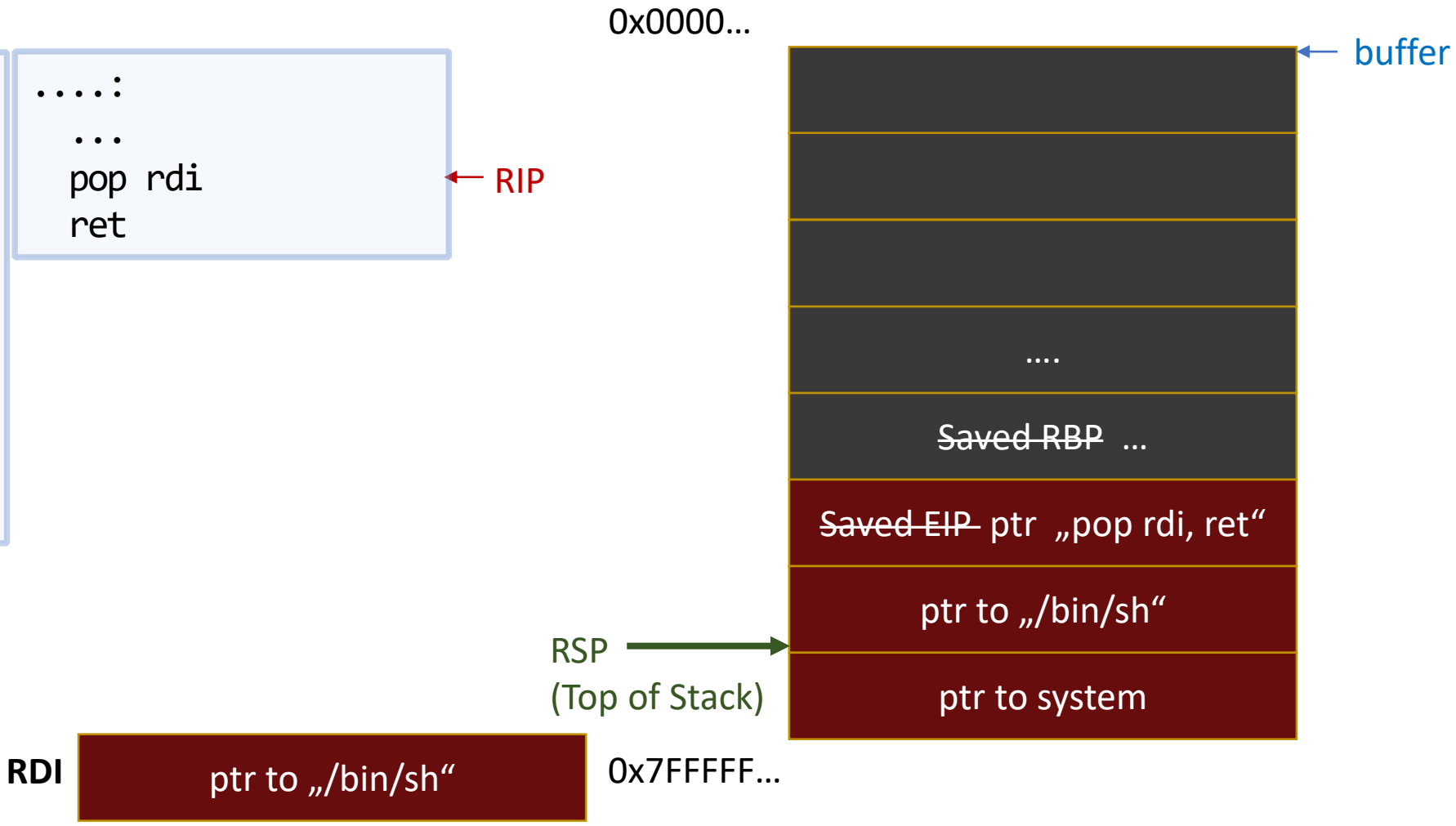
```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

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

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

← RIP

| Leave: | Ret: |
|---|---|
| mov rsp, rbp<br>pop rbp | "pop rip" |



# Payload strategy

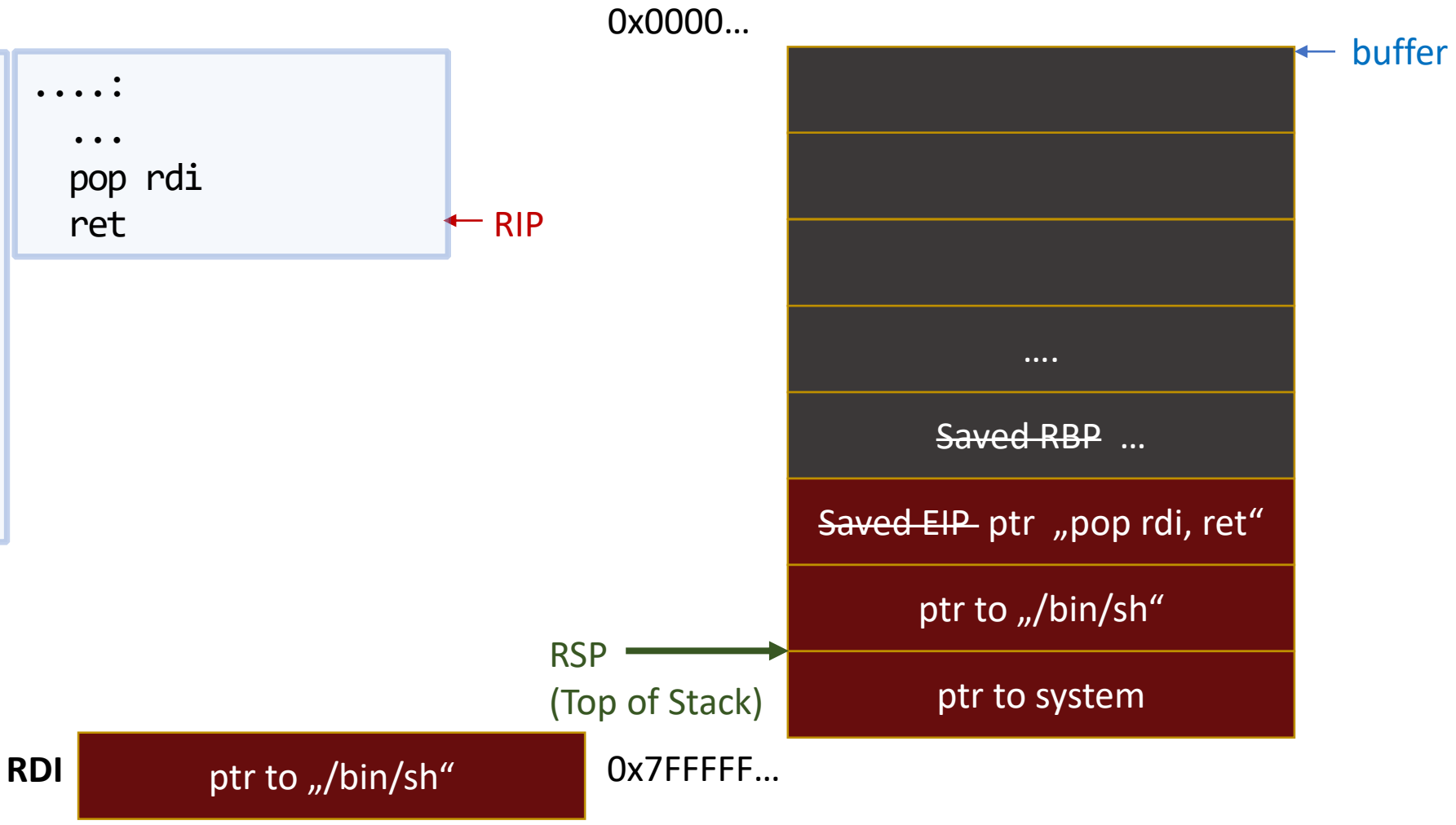
```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

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

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

← RIP

| Leave: | Ret: |
|---|---|
| mov rsp, rbp<br>pop rbp | "pop rip" |



# Payload strategy

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

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

```
system:
  ...
```

← RIP

0x0000…

← buffer

| |
|---|
| |
| |
| |
| …. |
| ~~Saved RBP~~ … |
| ~~Saved EIP~~ ptr „pop rdi, ret“ |
| ptr to „/bin/sh“ |
| ptr to system |

RSP →

0x7FFFFF…

| Leave: | Ret: |
|---|---|
| mov rsp, rbp<br>pop rbp | "pop rip" |

**RDI** | ptr to „/bin/sh“ |



# 64 Bit – simple ROP-chain

Payload = "A" * 32

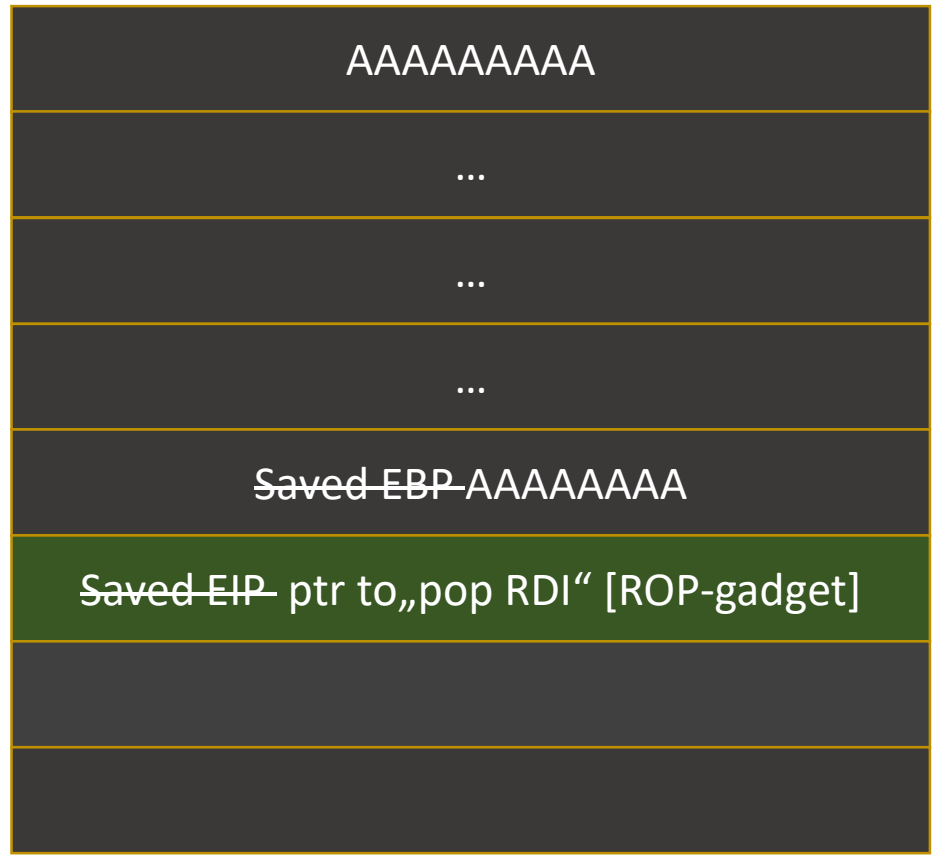
+ "AAAAAAAA" ~~(saved EBP)~~

# 64 Bit – simple ROP-chain

Payload = "A" * 32

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

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



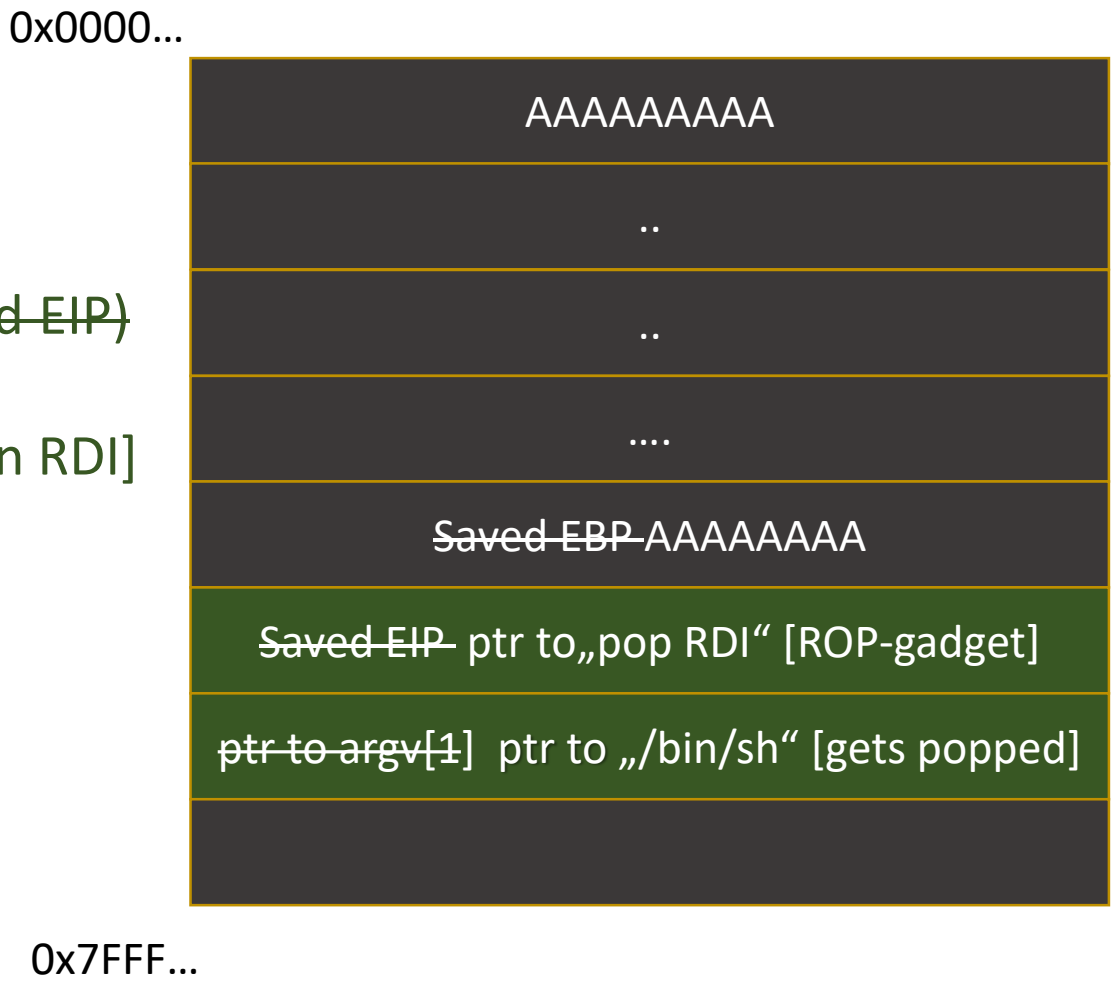
# 64 Bit – simple ROP-chain

Payload = "A" * 32

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

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

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



# 64 Bit – simple ROP-chain

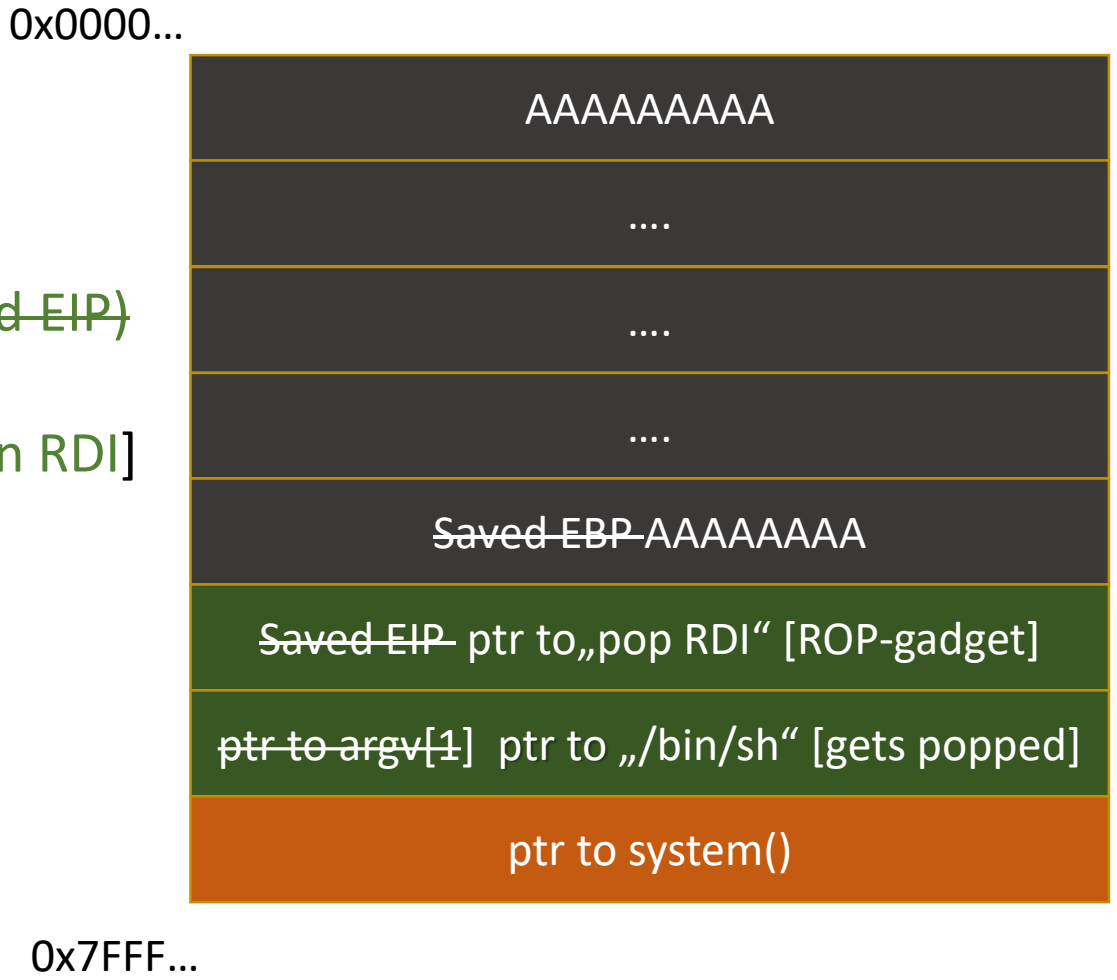
Payload = "A" * 32

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

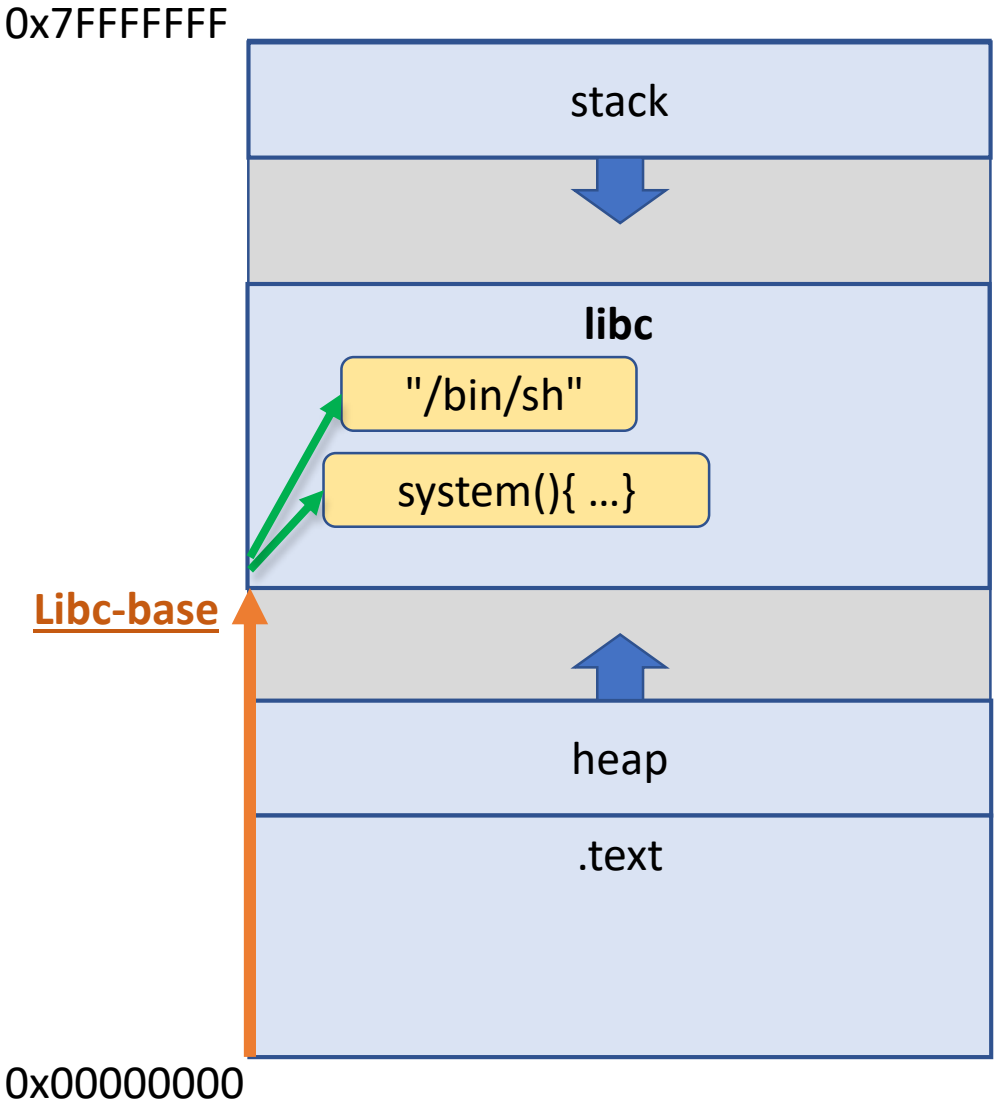
+ address "pop RDI; ret" [ROP-gadget]~~(saved EIP)~~

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

+ address of system



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




**Address_System = Libc-Base + Offset to `system()`**

**Address_Bin_Sh = Libc-Base + Offset to `"/bin/sh"`**

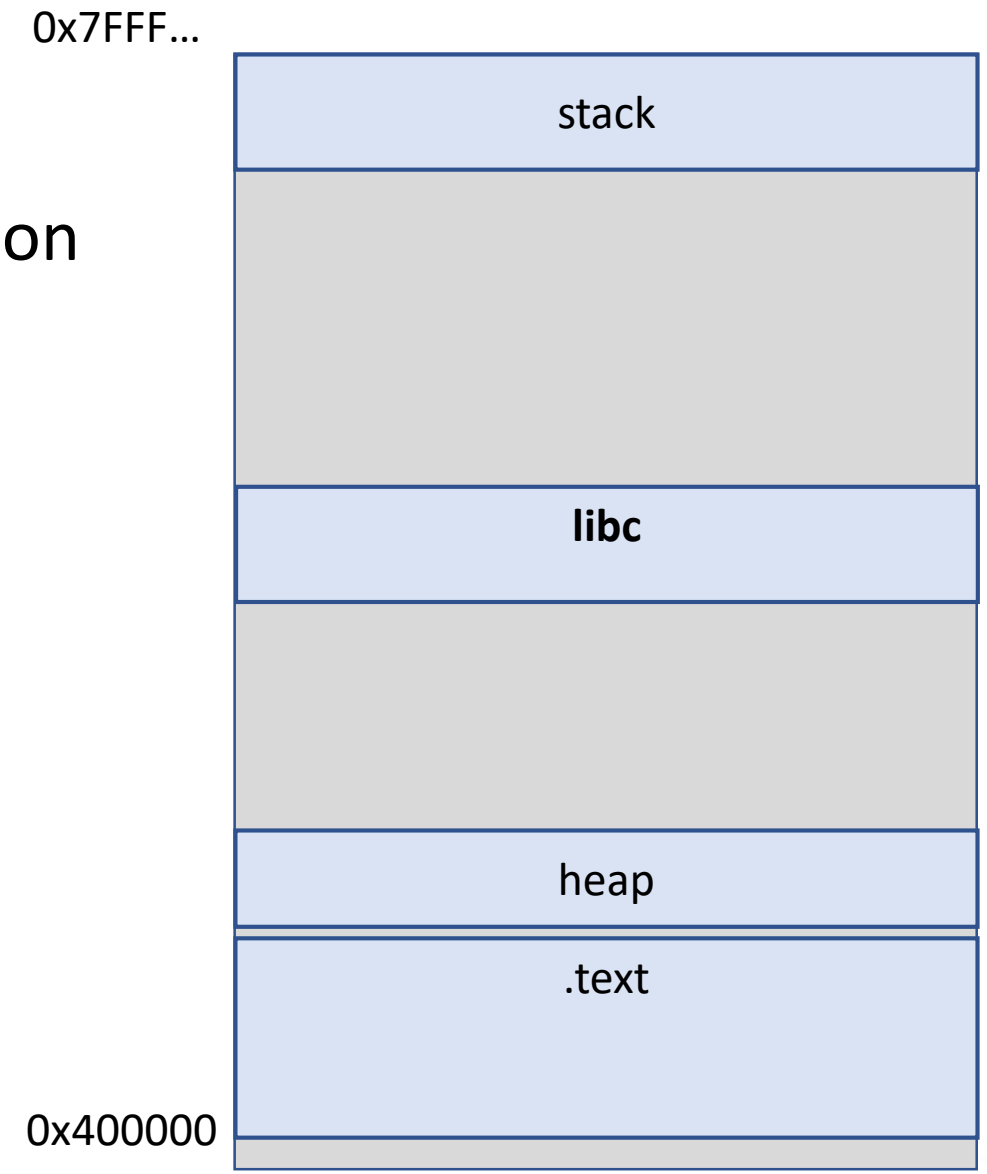
# Many roads lead to Rome …

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



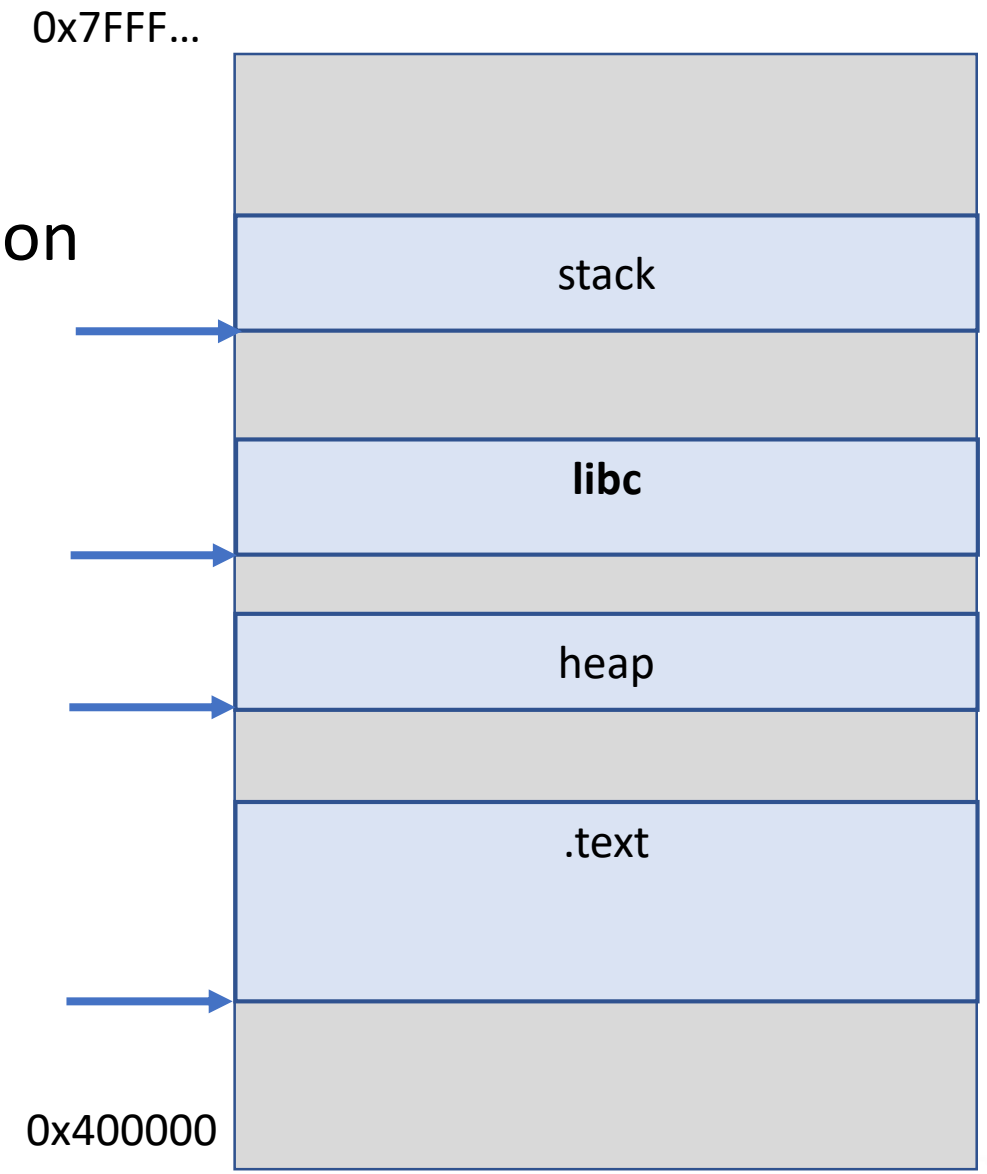
# 64 Bit – ASLR enabled

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



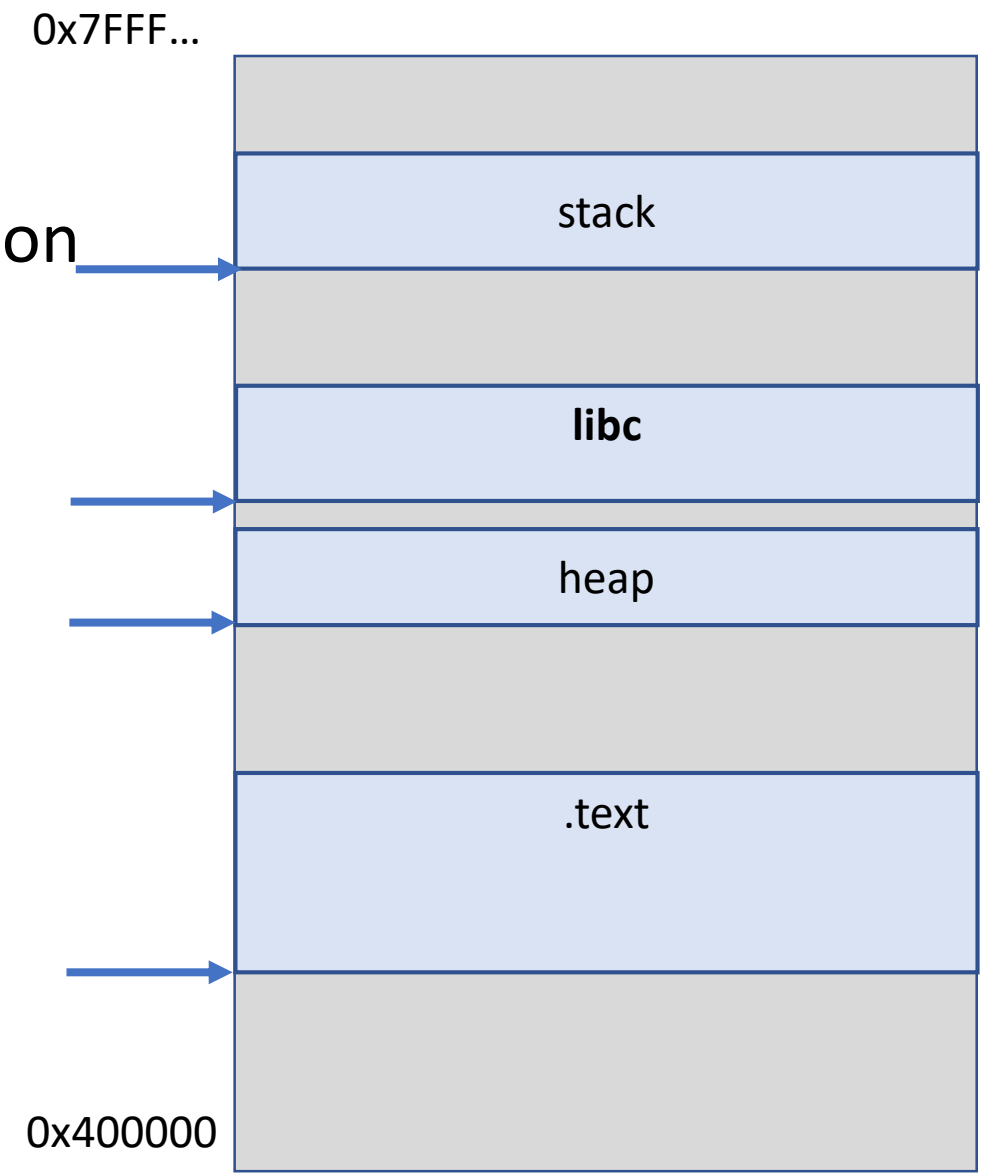
# 64 Bit – ASLR enabled

- 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



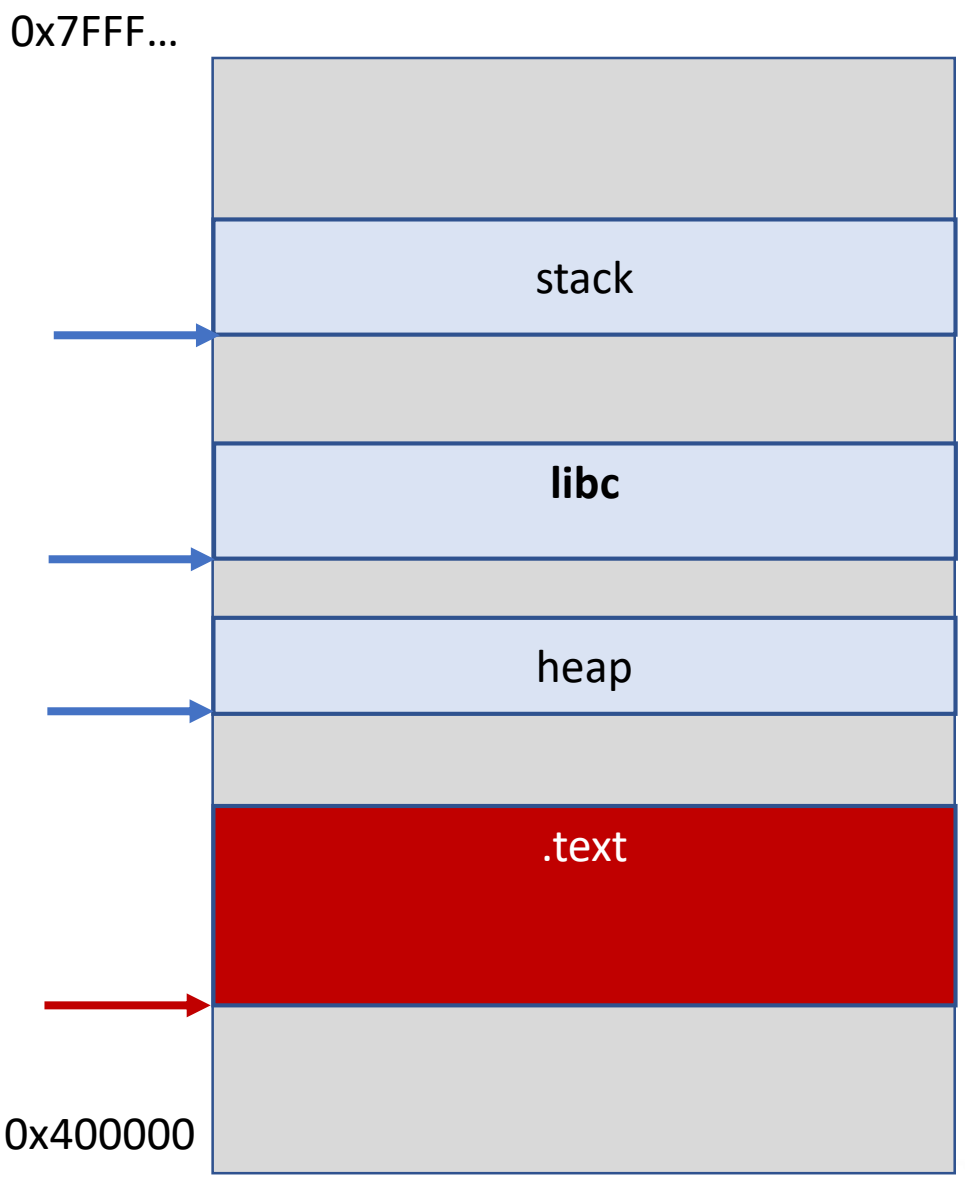
# 64 Bit – ASLR enabled

- 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



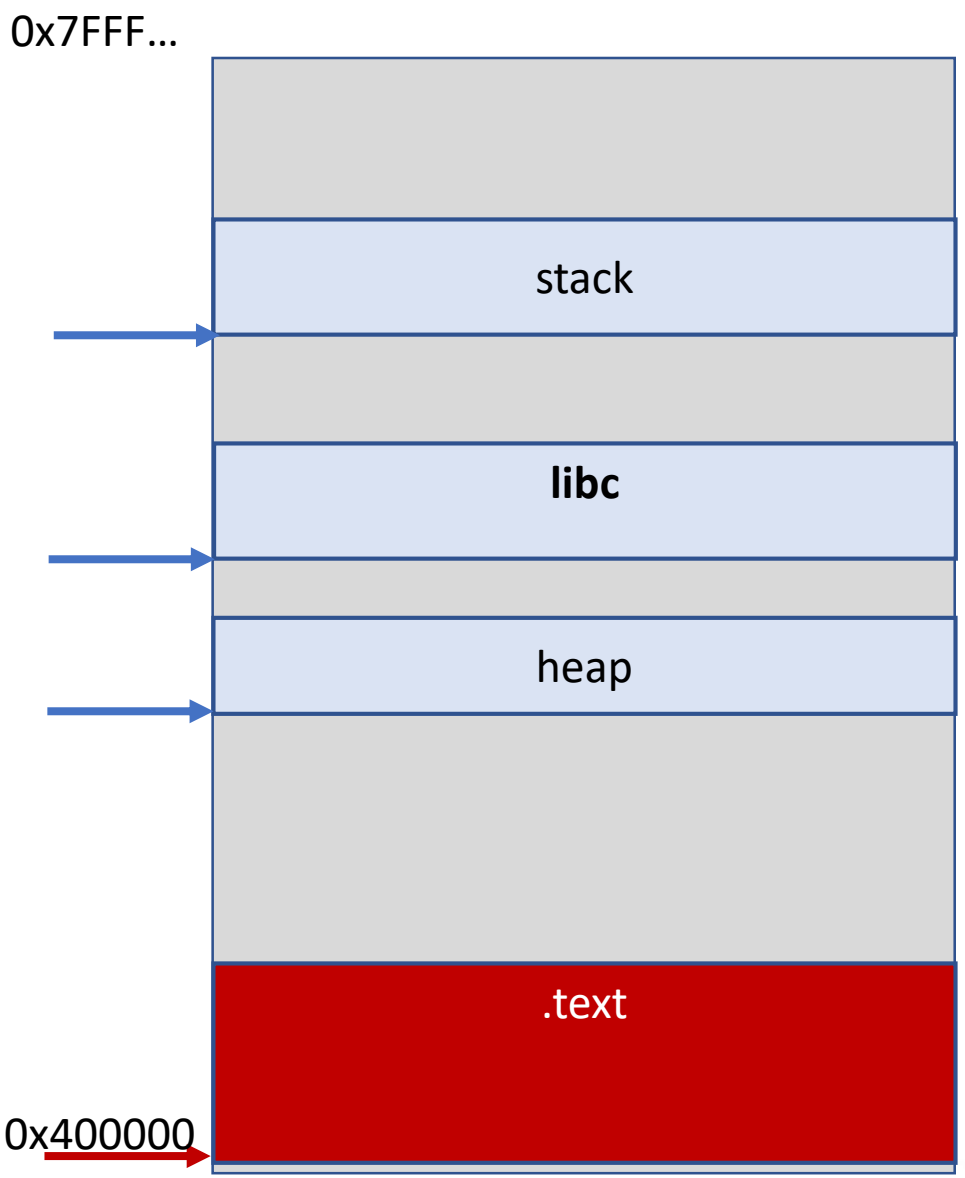
# 64 Bit – ASLR enabled

- **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**: **G**lobal **O**ffset **T**able
**PLT**: **P**rocedure **L**inkage **T**able

- Sections in the binary that enable linking of dynamic libraries

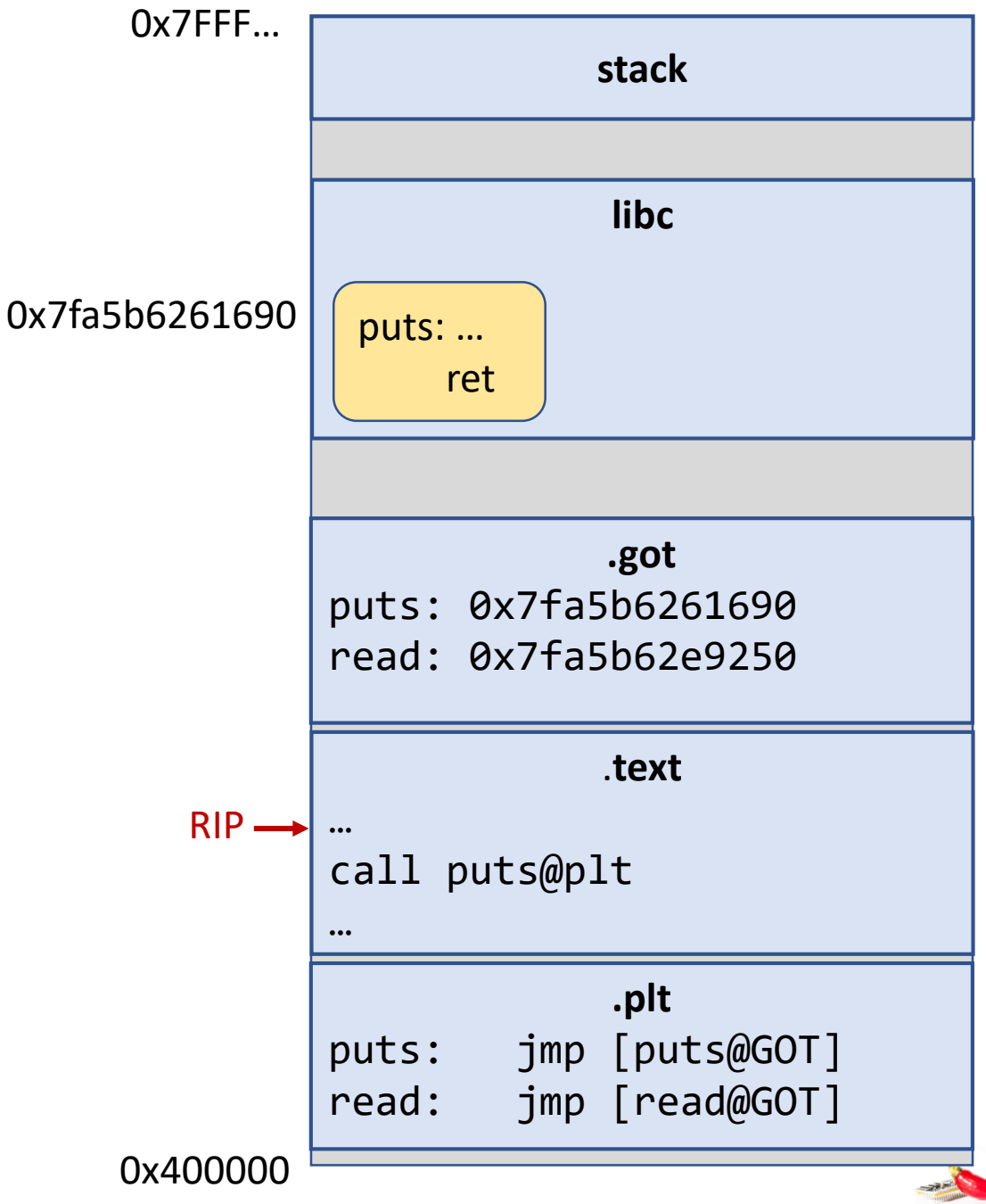
- Every library function that is called from inside the binary has a corresponding entry in those tables



# GOT and PLT

**GOT**: **G**lobal **O**ffset **T**able
**PLT**: **P**rocedure **L**inkage **T**able

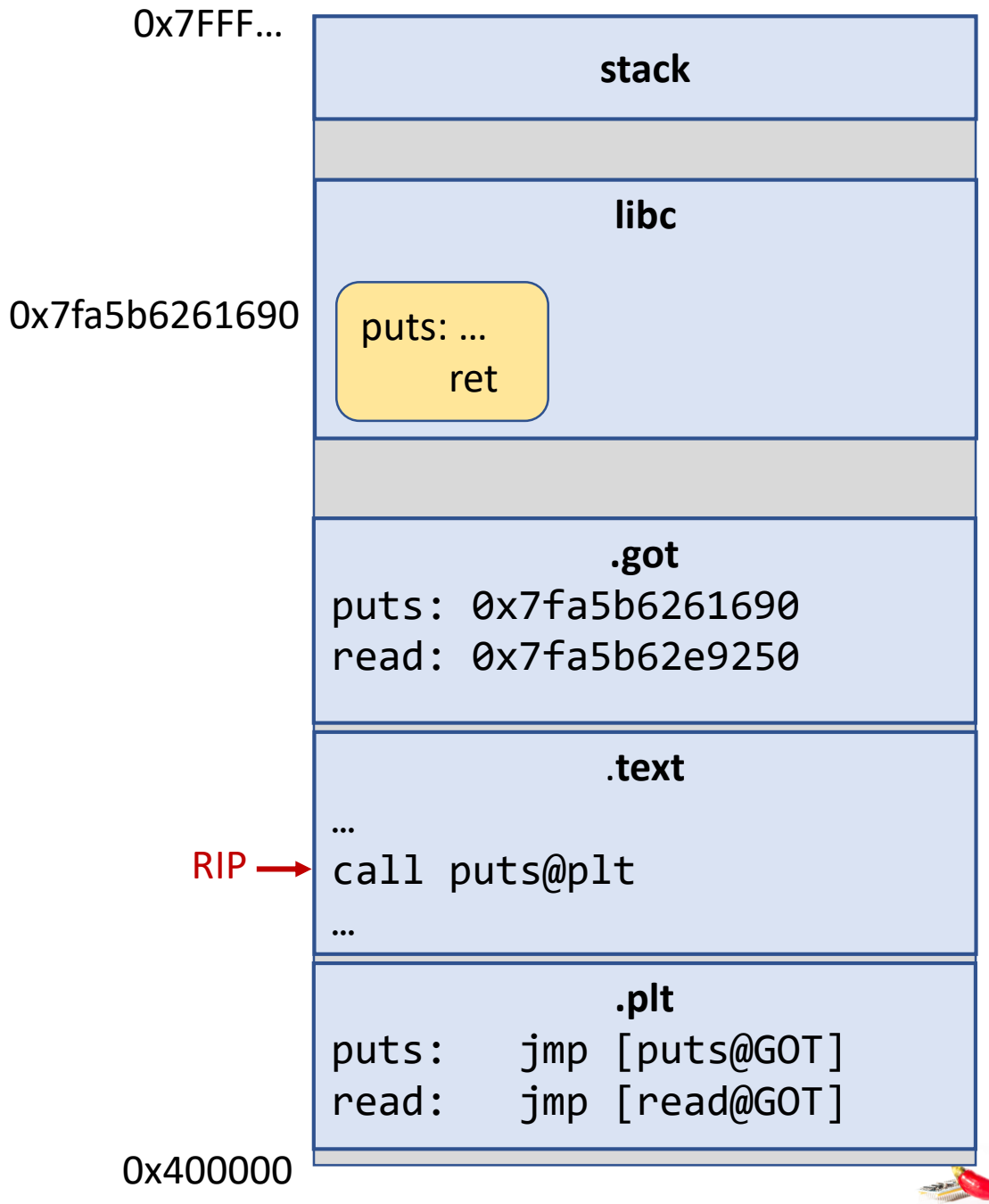
- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables



# GOT and PLT

**GOT**: **G**lobal **O**ffset **T**able
**PLT**: **P**rocedure **L**inkage **T**able

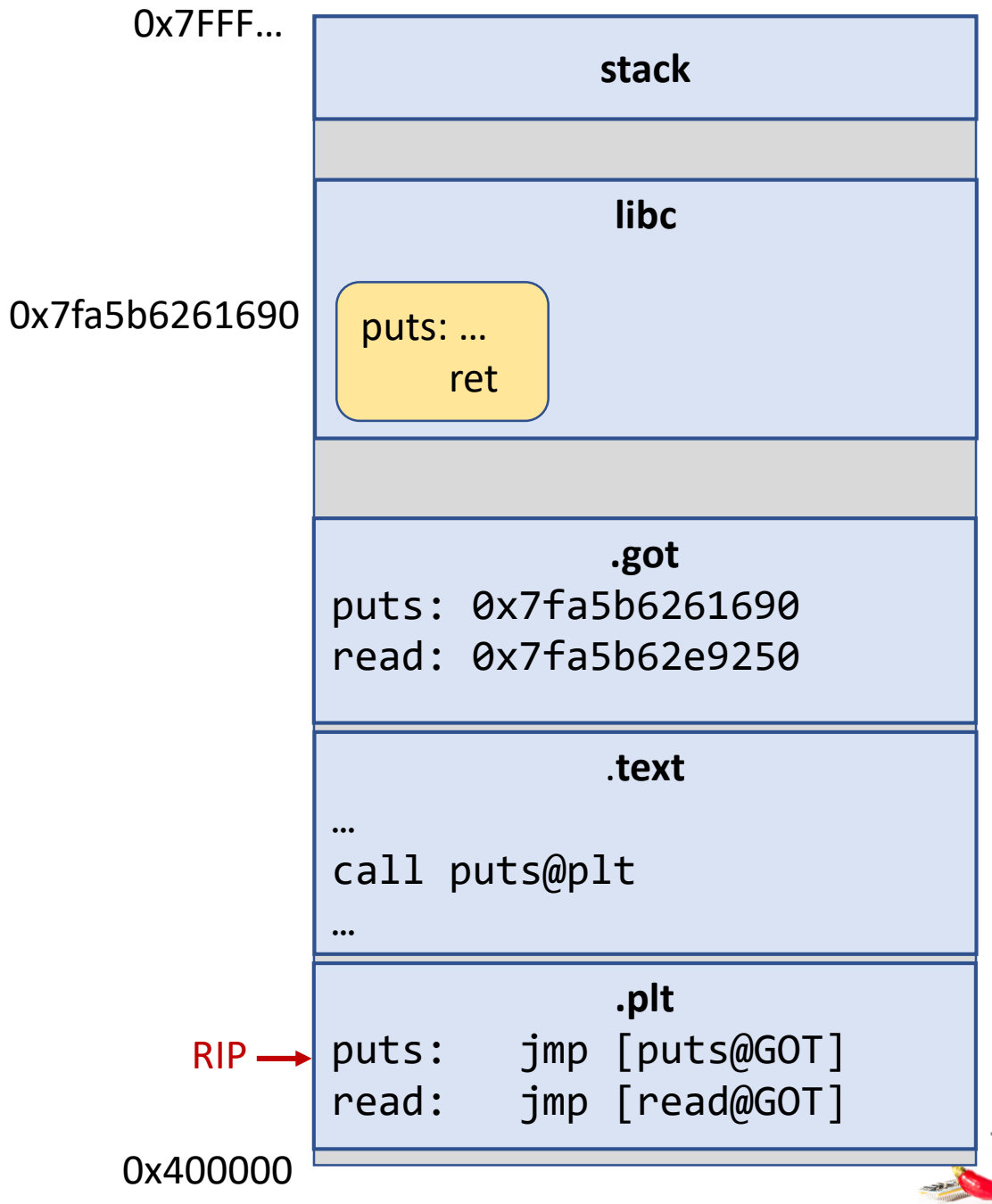
- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables



# GOT and PLT

**GOT**: **G**lobal **O**ffset **T**able
**PLT**: **P**rocedure **L**inkage **T**able

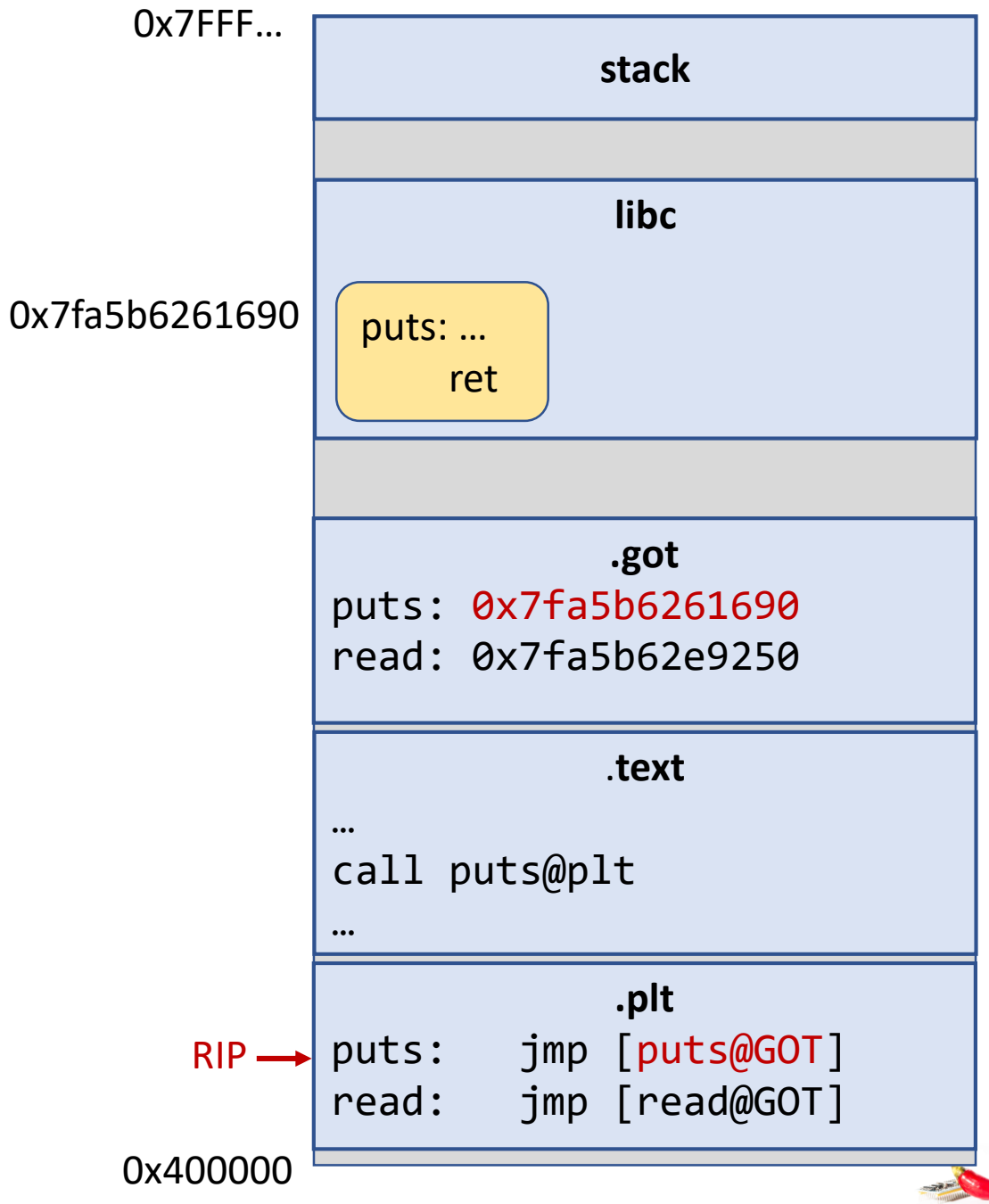
- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables



# GOT and PLT

**GOT**: **G**lobal **O**ffset **T**able
**PLT**: **P**rocedure **L**inkage **T**able

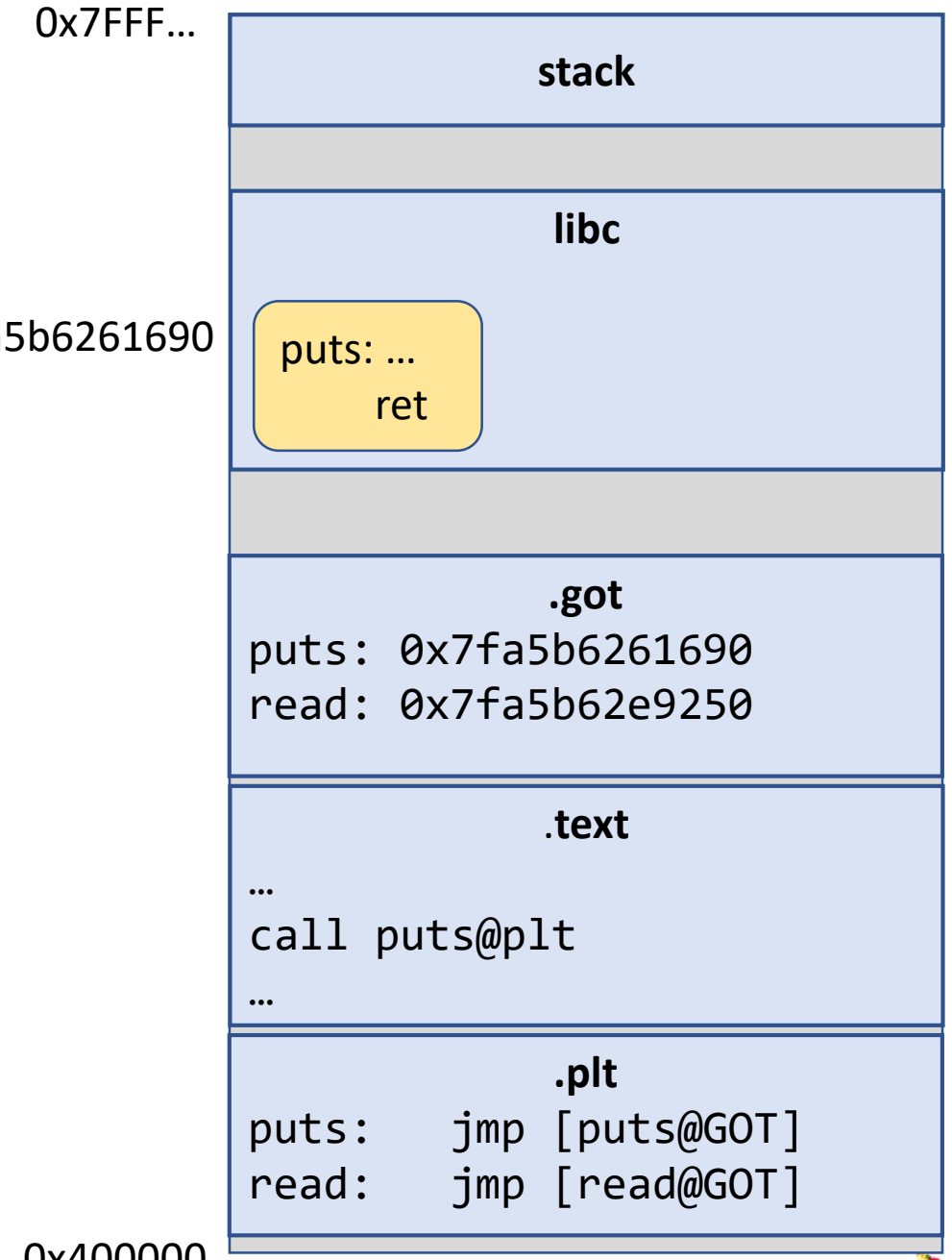
- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables



# GOT and PLT

**GOT**: **G**lobal **O**ffset **T**able
**PLT**: **P**rocedure **L**inkage **T**able

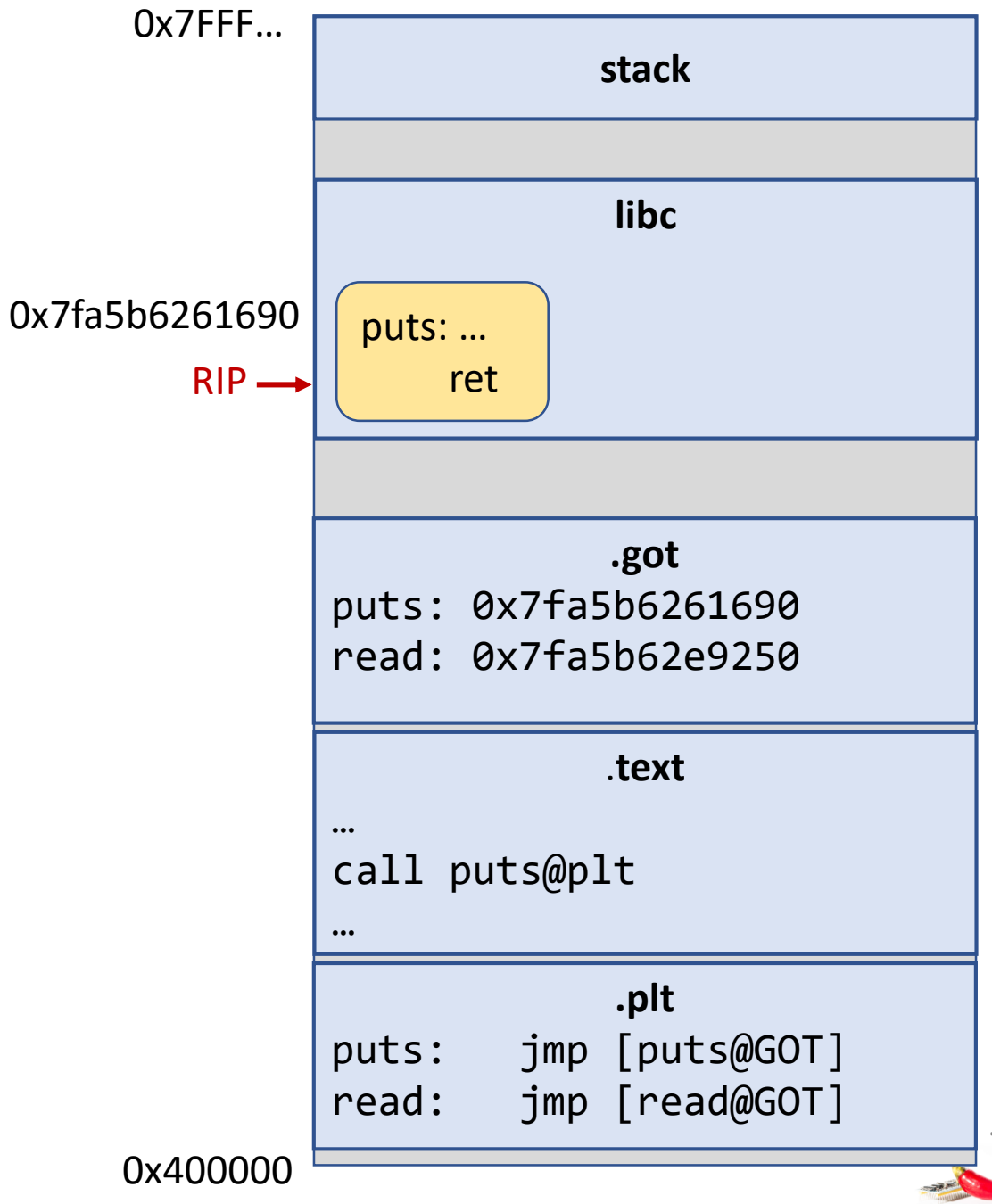
- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables



# GOT and PLT

**GOT**: **G**lobal **O**ffset **T**able
**PLT**: **P**rocedure **L**inkage **T**able

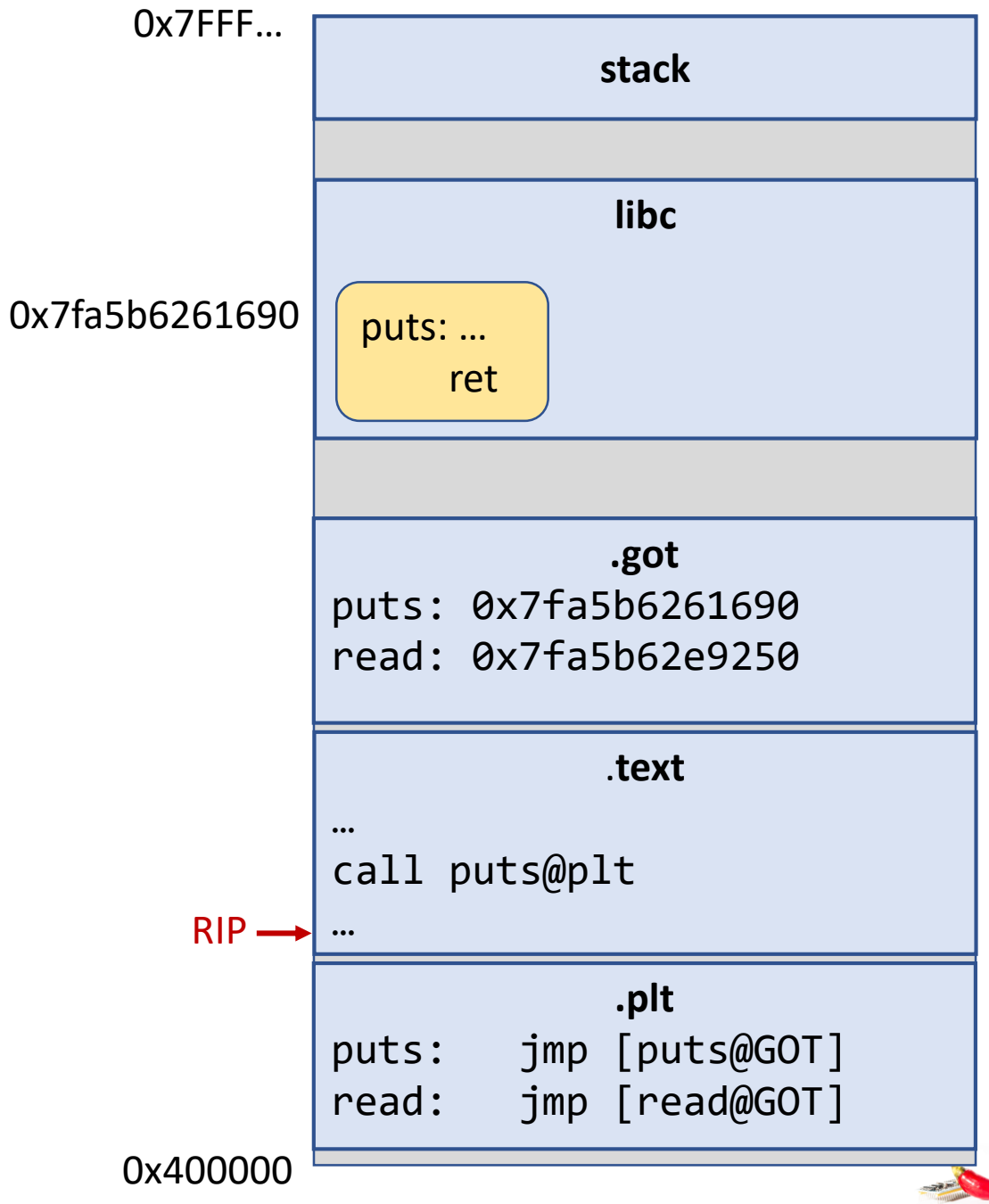
- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables



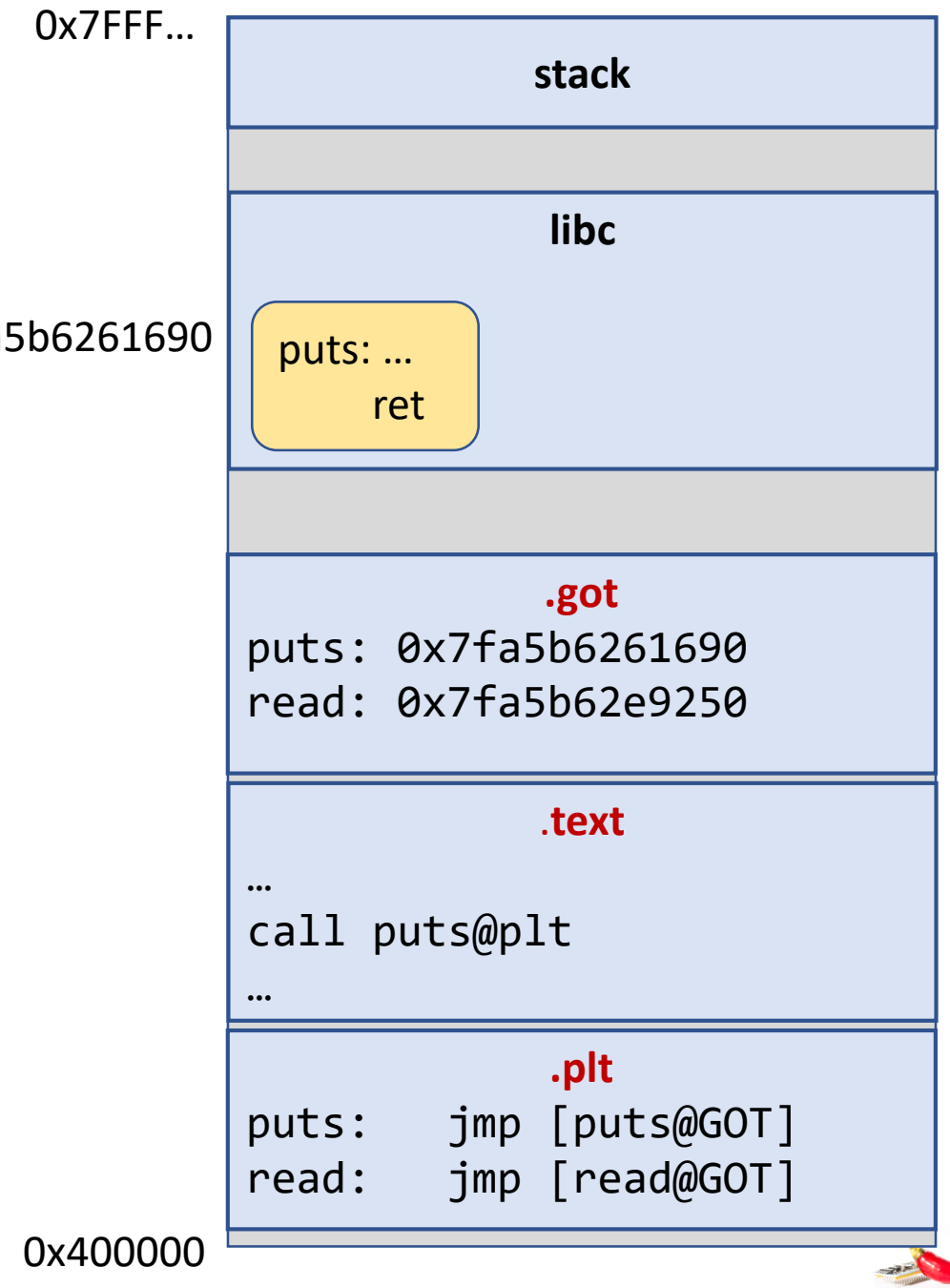
# GOT and PLT

**GOT**: **G**lobal **O**ffset **T**able
**PLT**: **P**rocedure **L**inkage **T**able

- 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

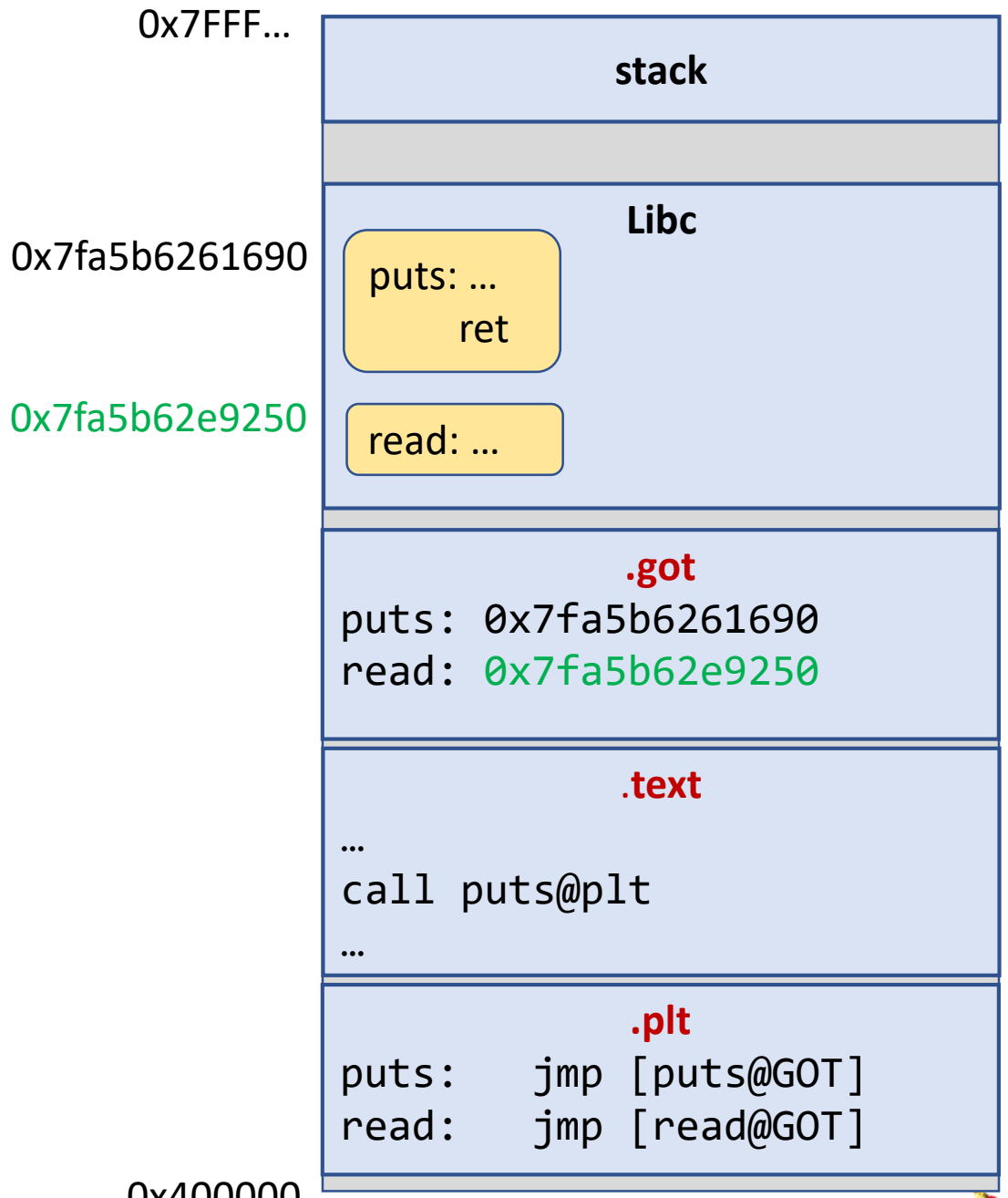


# GOT and PLT




Not randomized if not compiled as PIE

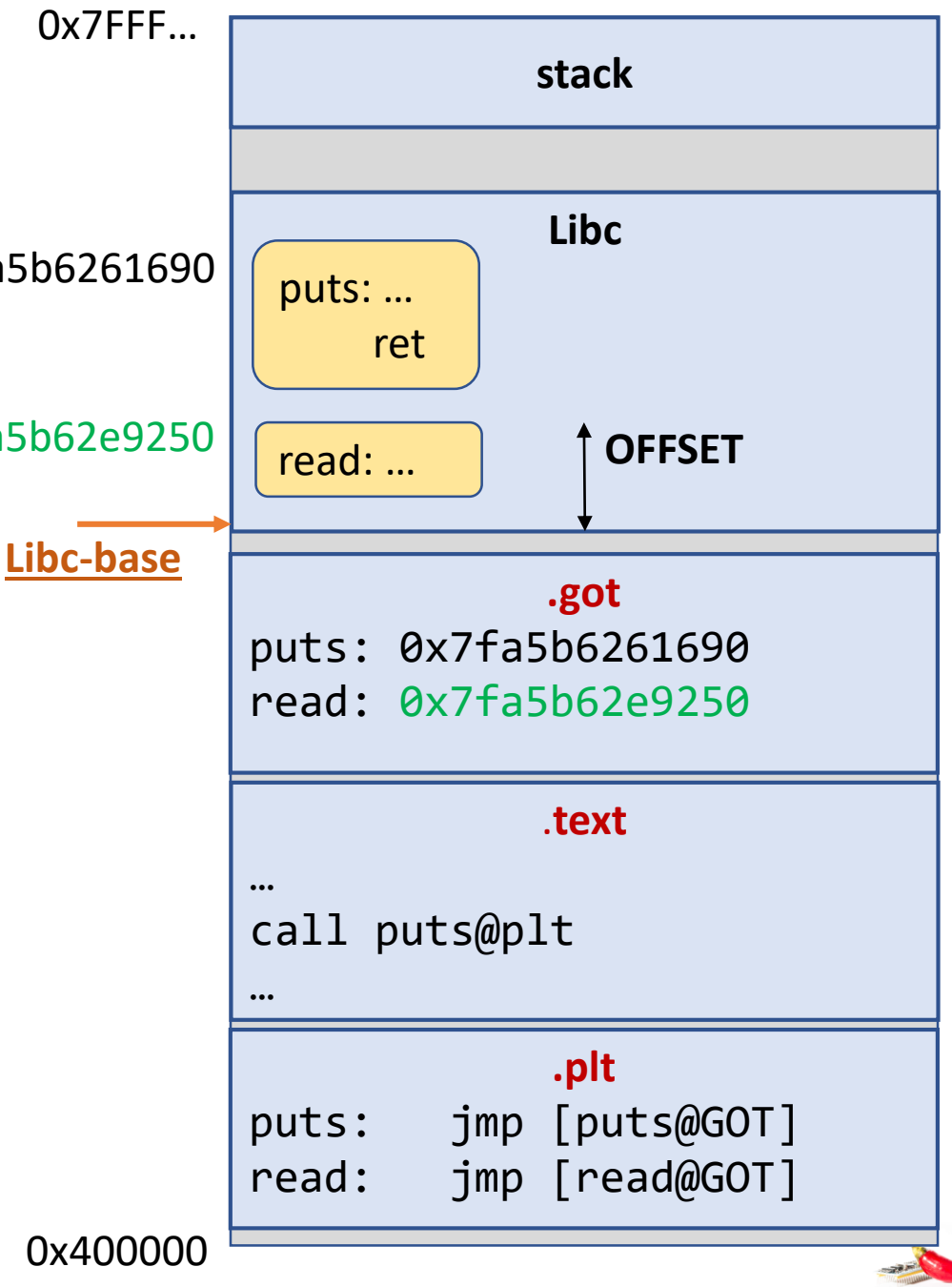
# GOT and PLT


0x7FFF…
stack
Libc
0x7fa5b6261690
puts: …
ret
0x7fa5b62e9250
read: …
.got
puts: 0x7fa5b6261690
read: 0x7fa5b62e9250
.text
…
call puts@plt
…
.plt
puts: jmp [puts@GOT]
read: jmp [read@GOT]
0x400000

# GOT and PLT

libcbase = [leaked address] – OFFSET

`libcbase = 0x7fa5b62e9250 - 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. 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