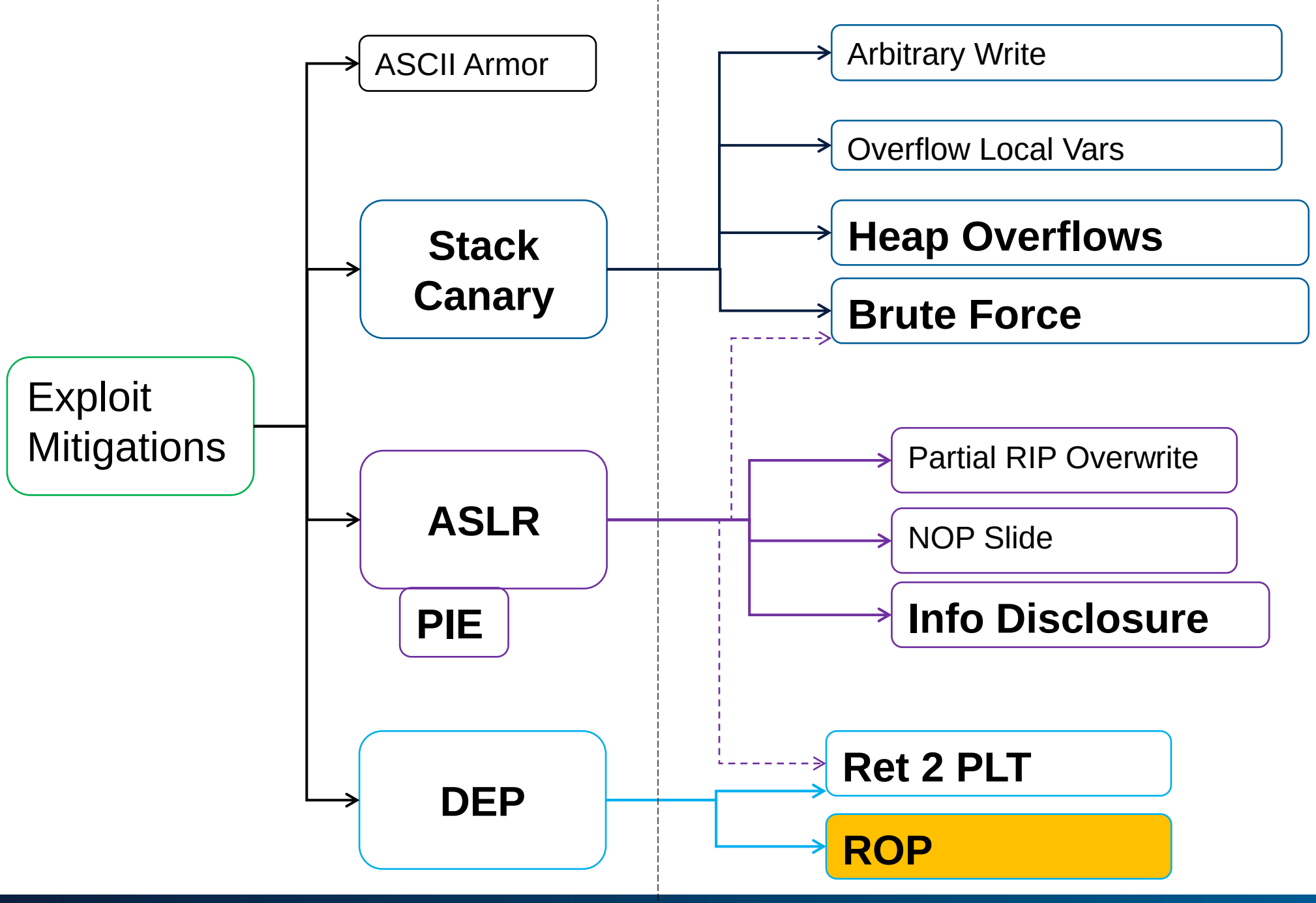
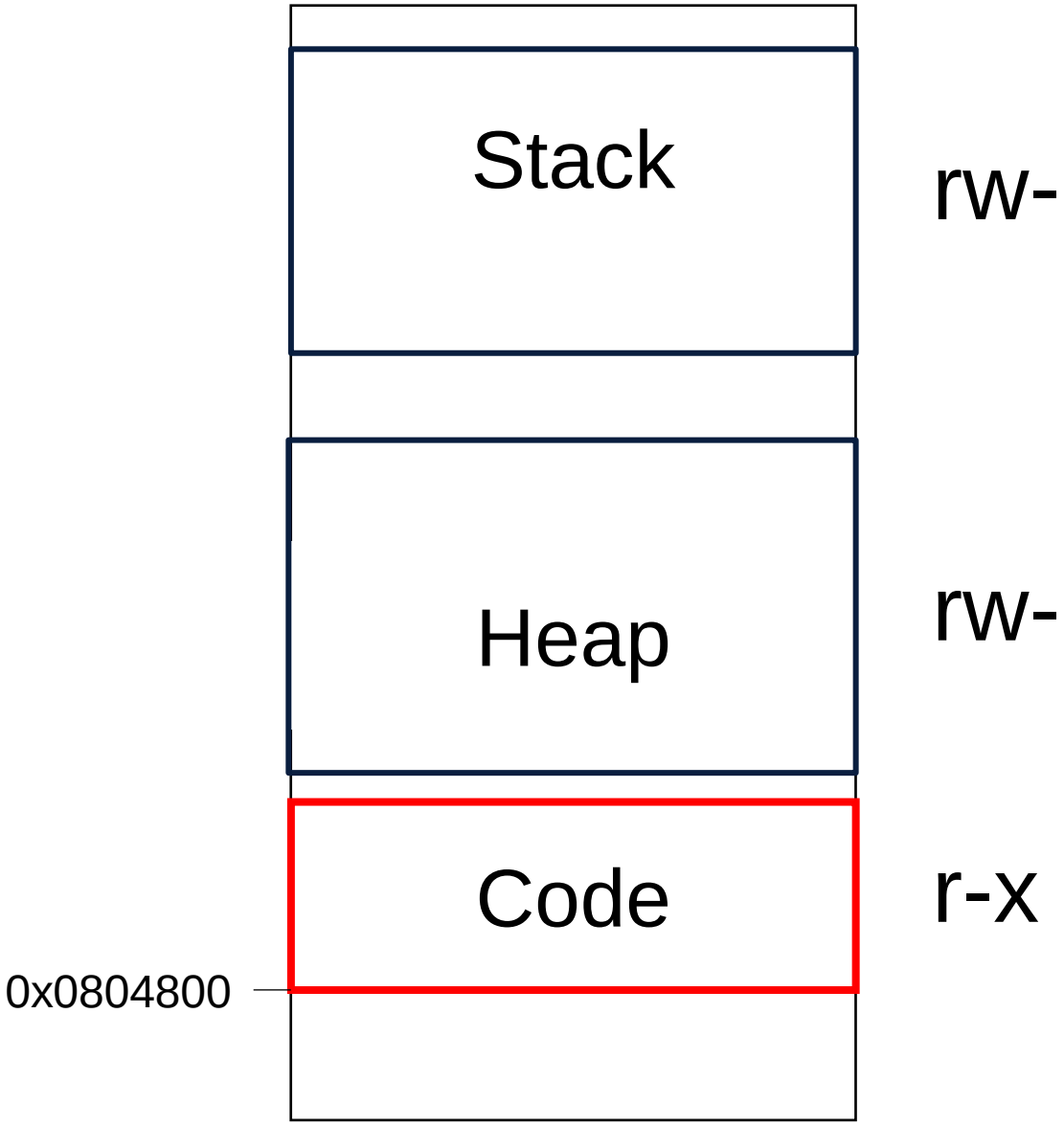
Return Oriented Programming

ROP


Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

Exploiting: DEP - Memory Layout



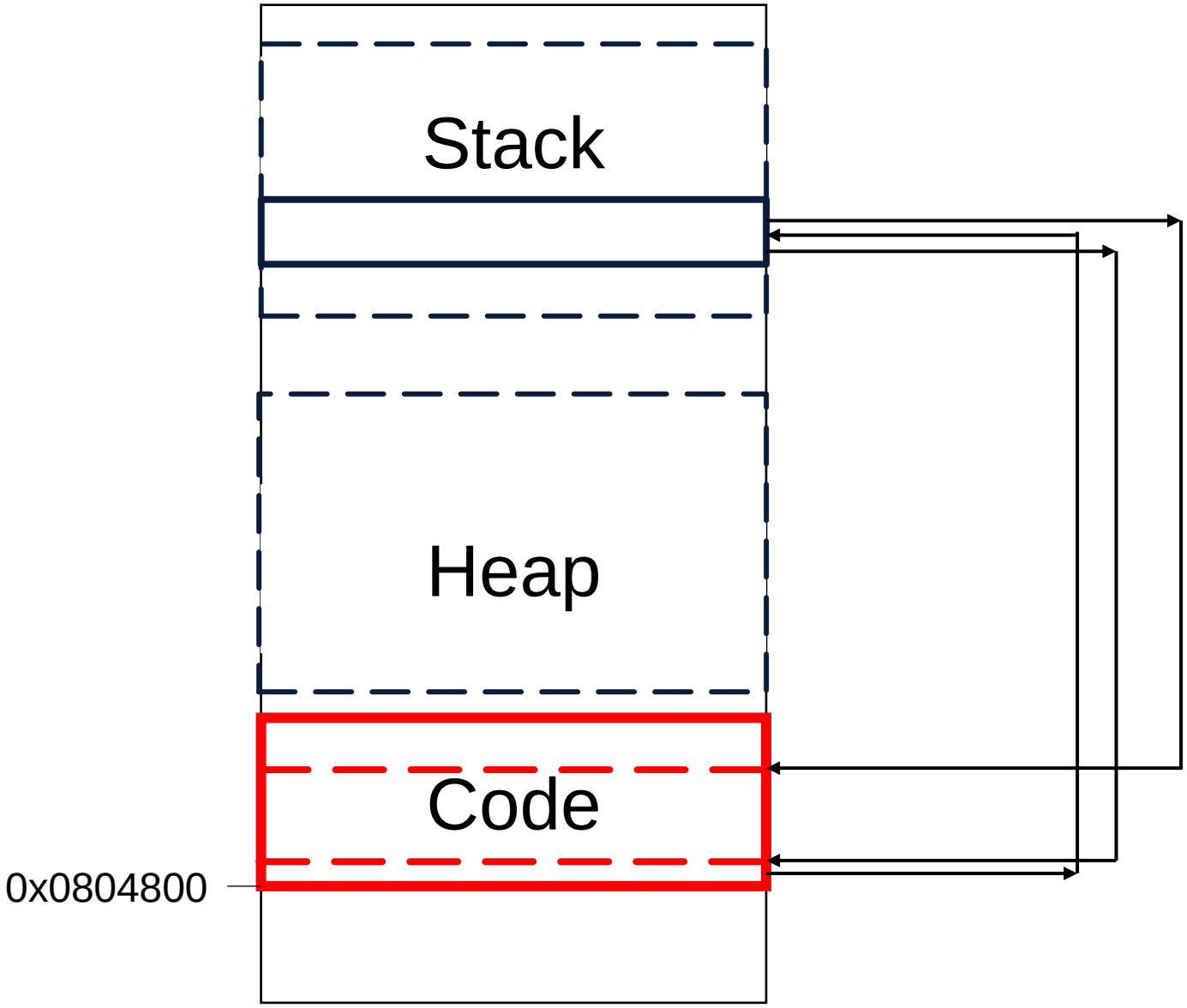
Exploiting: DEP - ROP

DEP does not allow execution of uploaded code

But what about **existing code**?

ROP: smartly put together existing code

Exploiting: DEP - Memory Layout

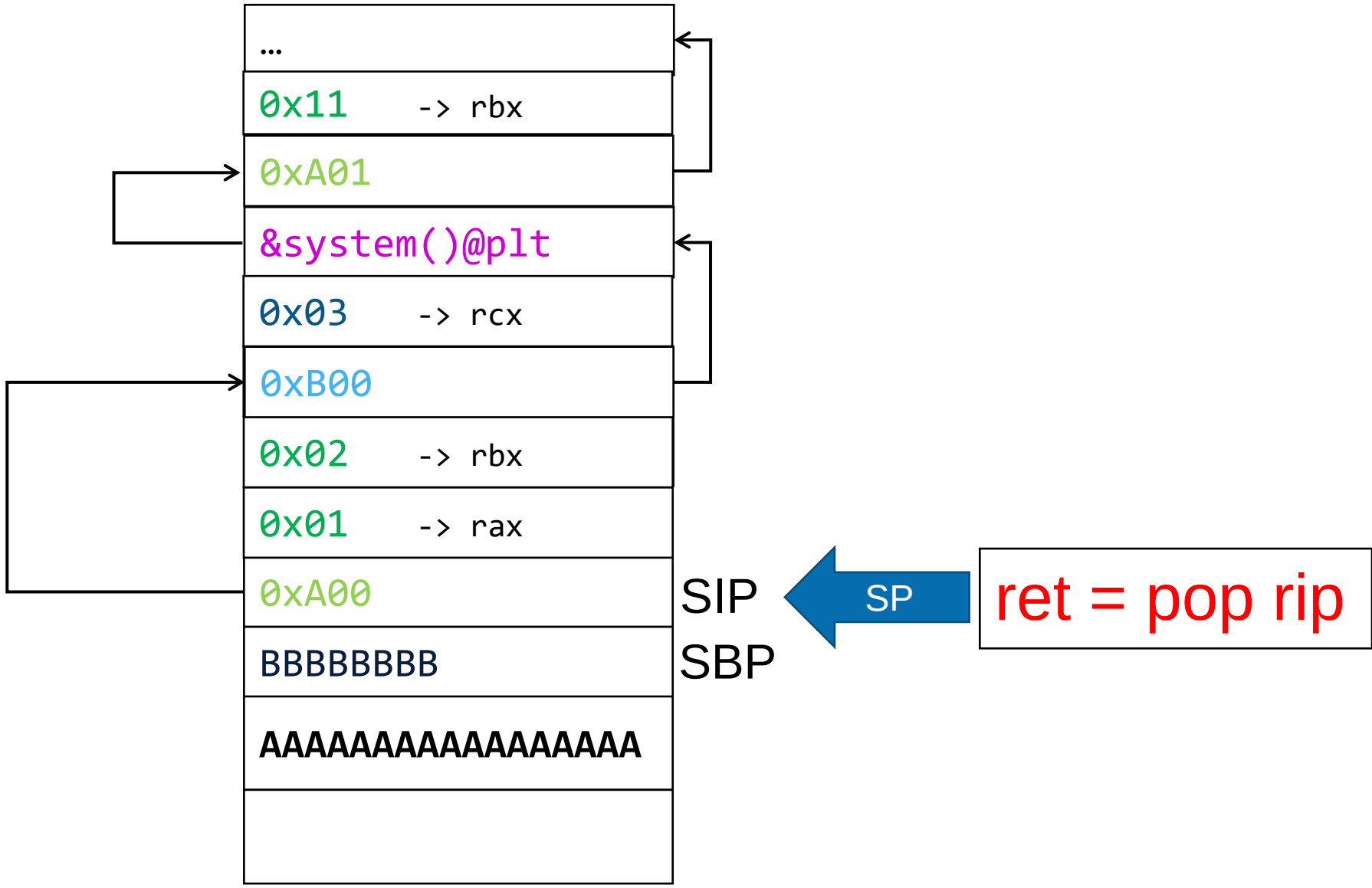


ROP In One Slide

ROP in 2 slides

```
0xB00: pop rcx
0xB01: ret
```

```
0xA00: pop rax
0xA01: pop rbx
0xA02: ret
```



ROP in 2 slides

```
payload = "AAAAAAAAAAAAAAAA"

payload += p64 (0xBBBBBBBB)
payload += p64 (0xA00)   # written on SIP
payload += p64 (0x01)
payload += p64 (0x02)
payload += p64 (0xB00)
payload += p64 (0x03)
payload += p64 (&system()@plt)
payload += p64 (0xA01)
payload += p64 (0x11)
payload += ...

print(payload)
```

ROP

Gadgets

Exploiting DEP - ROP

What is ROP?

Smartly chain gadgets together to execute arbitrary code

Gadgets:

- Some sequence of code, followed by a RET

Exploiting: DEP – ROP - Gadgets

So, what is are gadgets?

- Code sequence followed by a “ret”

```
pop r15 ; ret

add byte ptr [rcx], al ; ret

dec ecx ; ret
```

Exploiting: DEP – ROP - Gadgets

```
add byte ptr [rax], al ; add bl, dh ; ret
add byte ptr [rax], al ; add byte ptr [rax], al ; ret
add byte ptr [rax], al ; add cl, cl ; ret
add byte ptr [rax], al ; add rsp, 8 ; ret
add byte ptr [rax], al ; jmp 0x400839
add byte ptr [rax], al ; leave ; ret
add byte ptr [rax], al ; pop rbp ; ret
add byte ptr [rax], al ; ret
add byte ptr [rcx], al ; ret
add cl, cl ; ret
add eax, 0x20087e ; add ebx, esi ; ret
add eax, 0xb8 ; add cl, cl ; ret
add ebx, esi ; ret
```

Exploiting: DEP – ROP - Gadgets

How to find gadgets?

- Search in code section for byte 0xc3 (=ret)
- Go backwards, and decode each byte
- For each byte:
  - Check if it is a valid x32 instruction
  - If yes: add gadget, and continue
  - If no: continue

80 00 51 02 80 31 60 00 0e 05 **c3** 20 07 dd da 23

Exploiting: DEP – ROP - Gadgets

How to find gadgets?

- Search in code section for byte 0xc3 (=ret)
- Go backwards, and decode each byte
- For each byte:
  - Check if it is a valid x32 instruction
  - If yes: add gadget, and continue
  - If no: continue

80 00 51 02 80 31 60 00 0e **05 c3** 20 07 dd da 23

Exploiting: DEP – ROP - Gadgets

How to find gadgets?

- Search in code section for byte 0xc3 (=ret)
- Go backwards, and decode each byte
- For each byte:
  - Check if it is a valid x32 instruction
  - If yes: add gadget, and continue
  - If no: continue

80 00 51 02 80 31 60 00 **0e 05 c3** 20 07 dd da 23

Exploiting: DEP – ROP - Gadgets

There will be gadgets which were not created by the compiler

- x86 instructions are not static size
- 1-15bytes
  - Unlike RISC (usually 4 byte size)
- Start parsing at the “wrong offset”

Exploiting: DEP – ROP - Gadgets

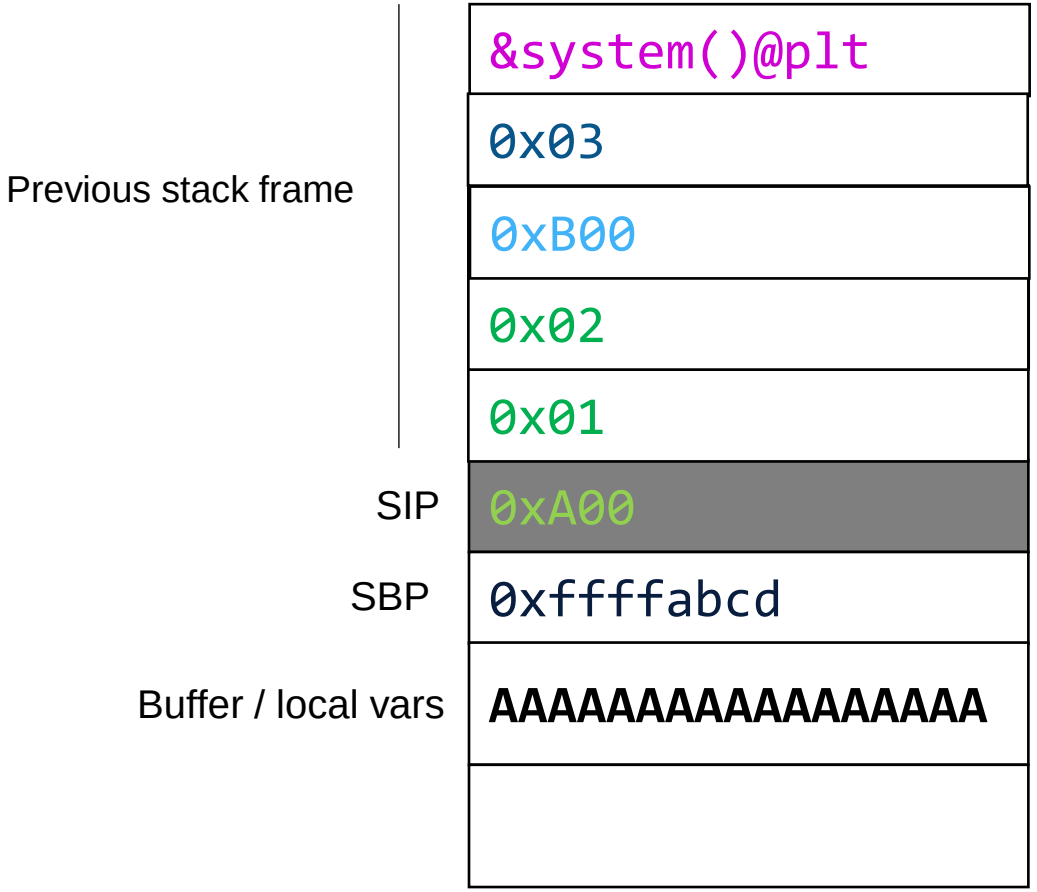
Why ret?

ret = pop eip

It "jumps" to the address at the stack location %rsp
It removes that element from the stack

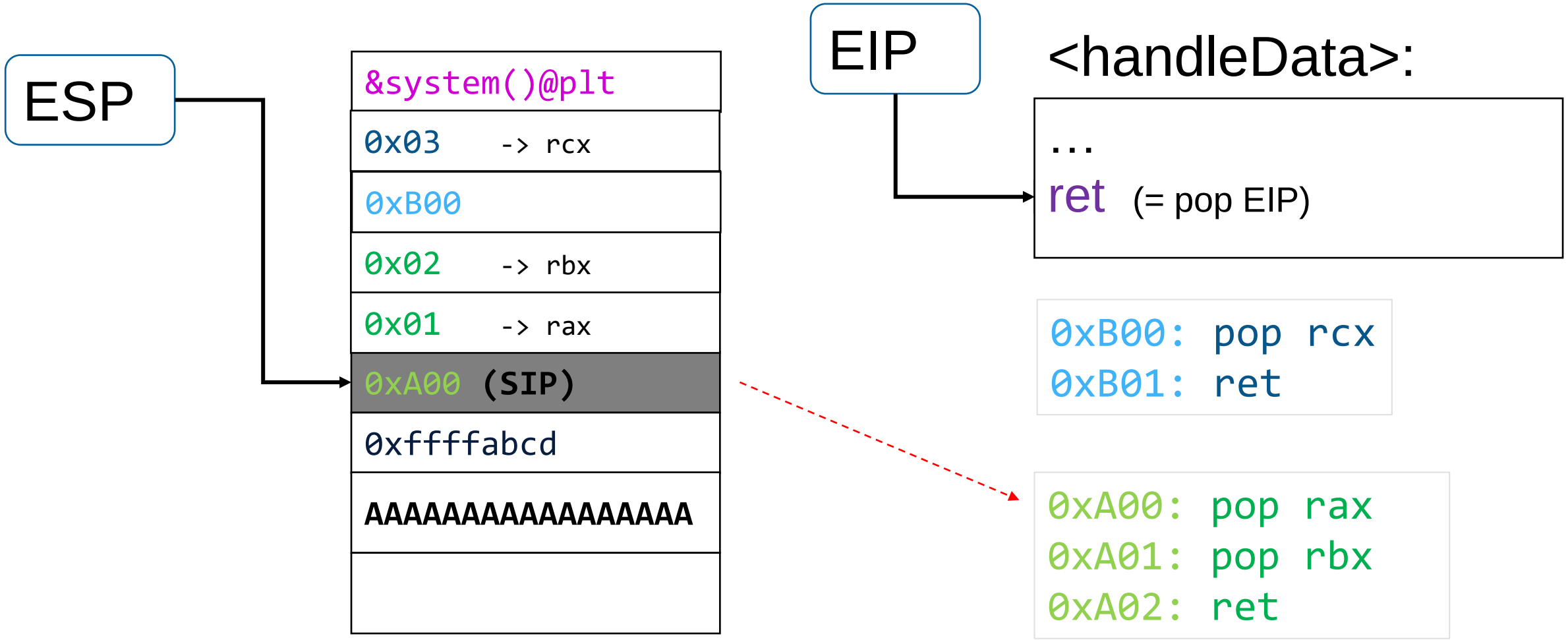
64 bit ROP By Example

64 bit ROP By Example

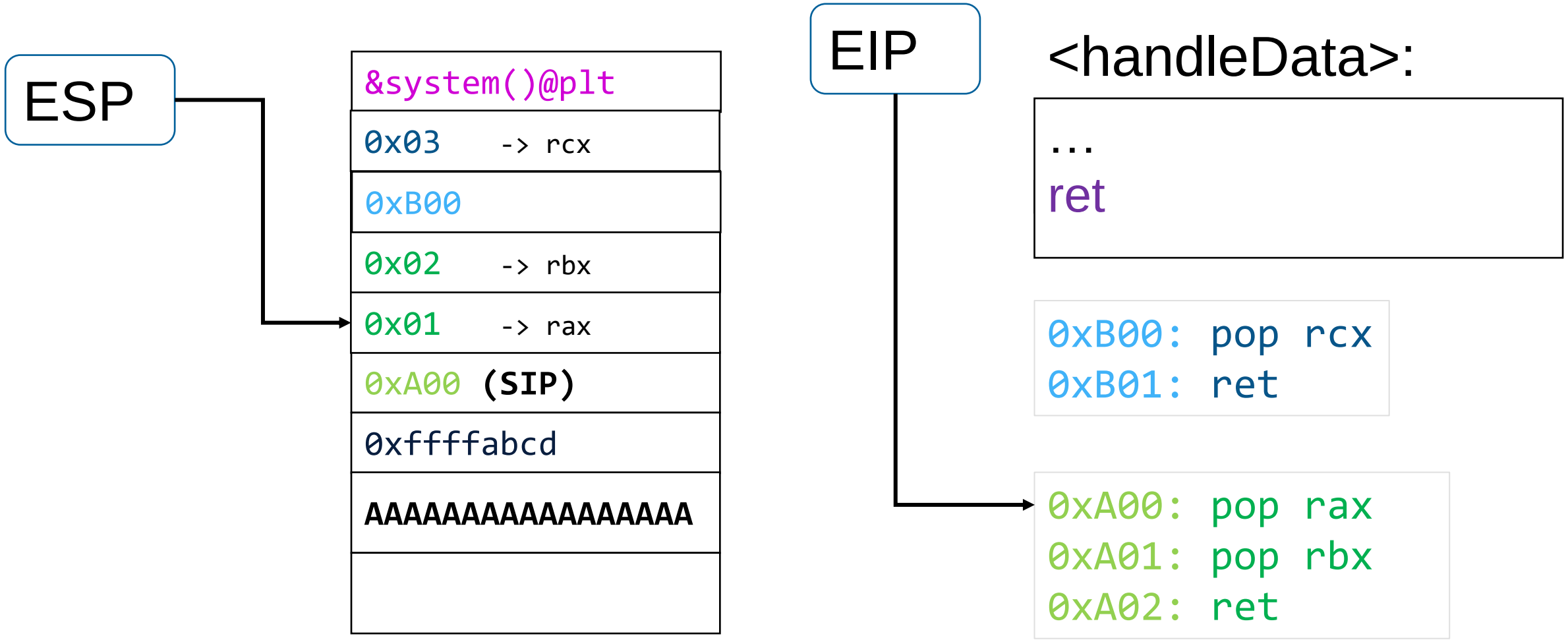


View of the stack
After stack overflow
Before ret

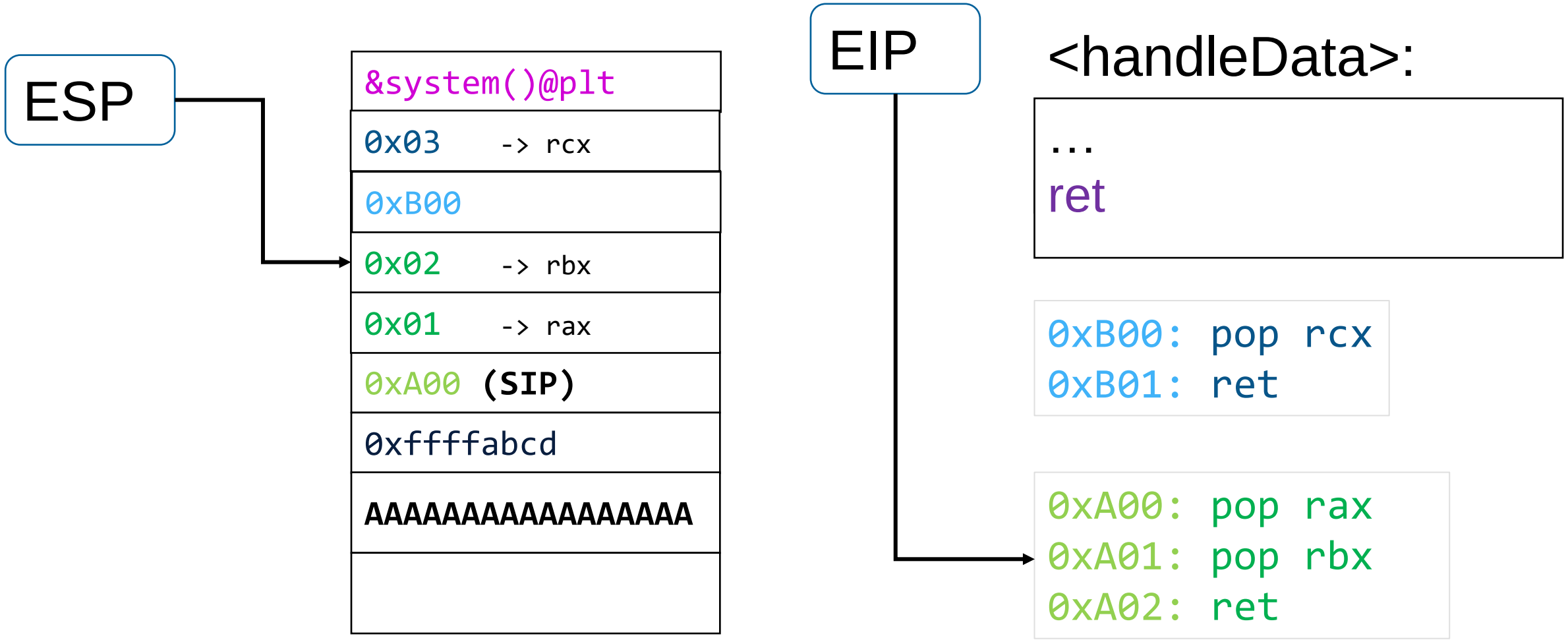
64 bit ROP By Example



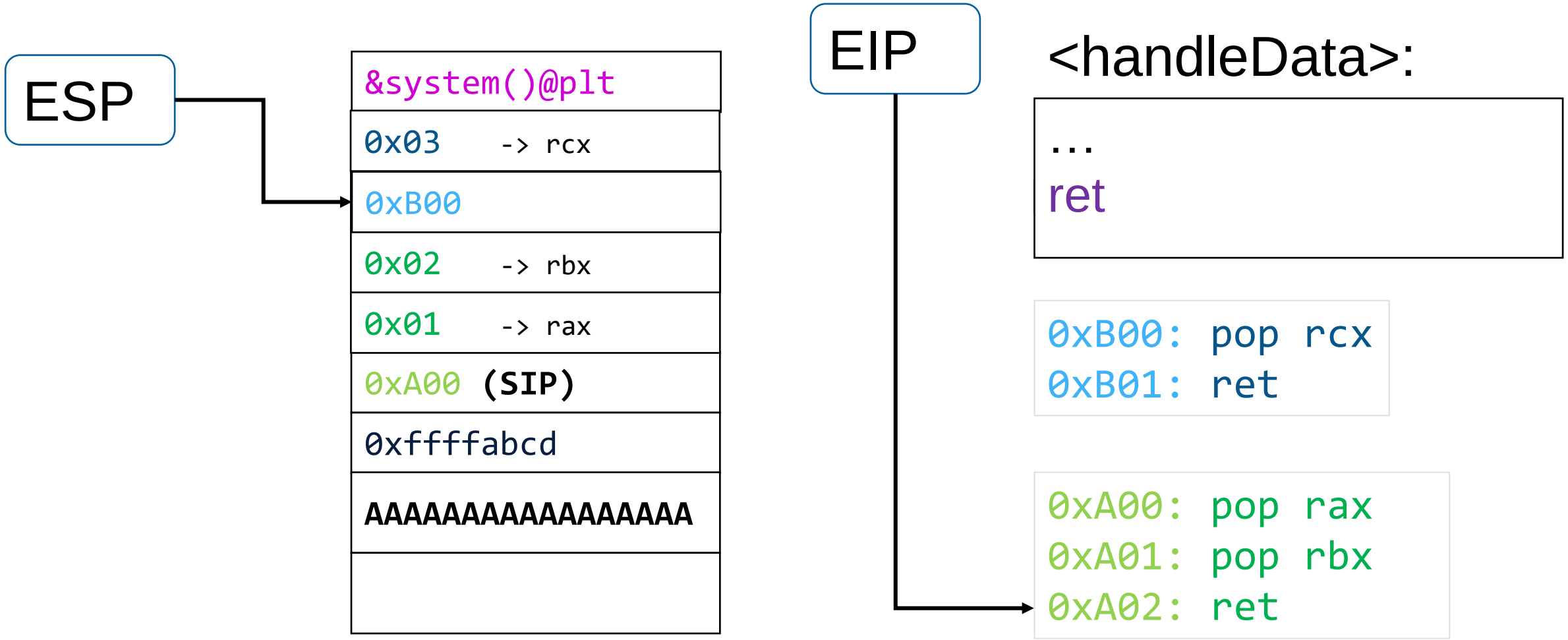
64 bit ROP By Example



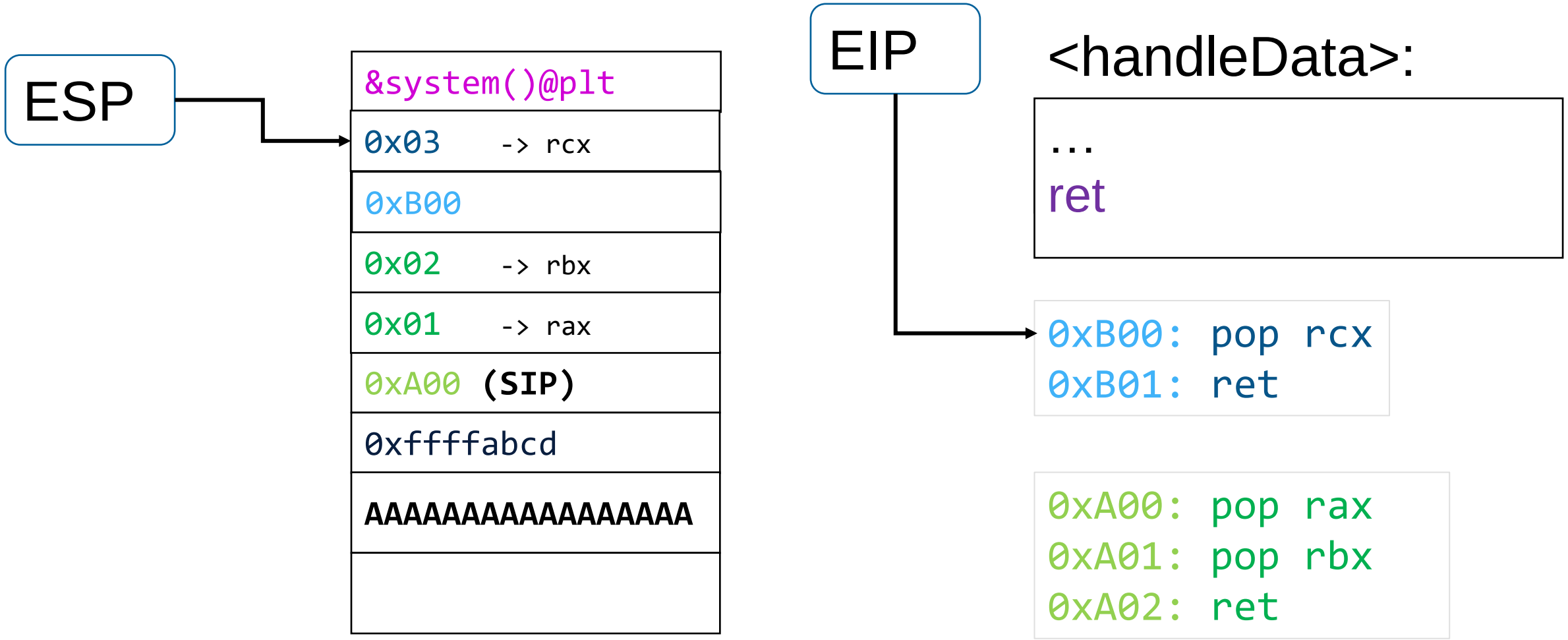
64 bit ROP By Example



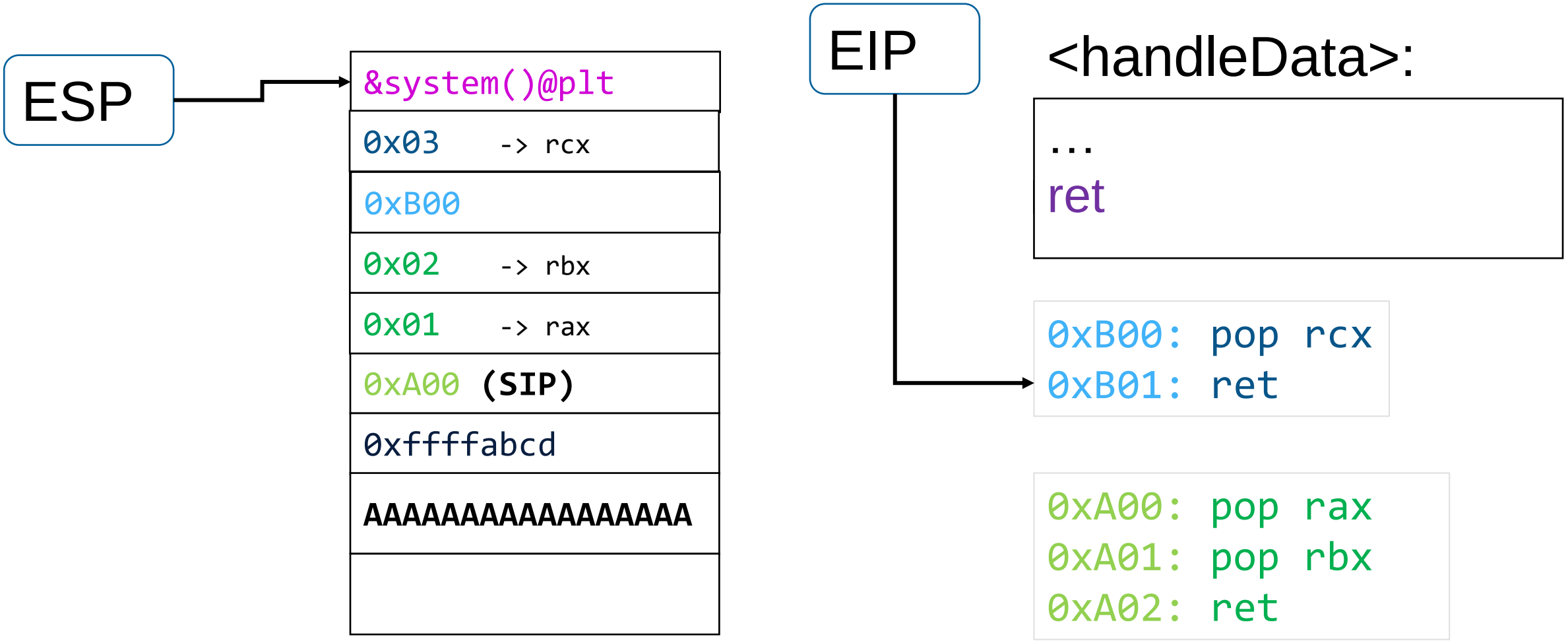
64 bit ROP By Example



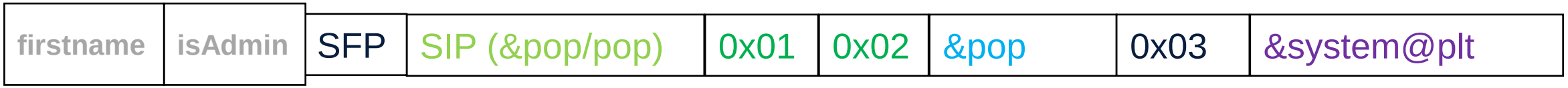
64 bit ROP By Example



64 bit ROP By Example



64 bit ROP By Example



Stack grows down

Writes go up

ROP By Example

call/ret’s can be chained!

Arbitrary code execution with no code uploaded

“Shellcode” consists of:

- Addresses of gadgets
- Arguments for gadgets (addresses, or immediates)
- NOT: assembler instructions

Finding gadgets: ropper

$ **ropper**
(ropper)> **file challenge16**
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] File loaded.

(challenge16/ELF/x86_64)> **search pop rbx;**
[INFO] Searching for gadgets: pop rbx;

[INFO] File: challenge16
0x000000000044b431: pop rbx; adc eax, 0xc6e8fffd; ret 0xfffc;
0x000000000047f177: pop rbx; add rax, qword ptr [rdx + 8]; pop rbp; pop r12; jmp rax;
0x00000000004492a6: pop rbx; and eax, 0x7ff80000; ret;
0x000000000046be26: pop rbx; and eax, 0xc; pop rbp; pop r12; ret;
0x00000000004491fb: pop rbx; cmove rax, rdx; ret;
0x000000000044928b: pop rbx; cmovne rax, rdx; ret;
0x000000000415af5: pop rbx; jmp rax;

ROP: Conclusion

ROP: Conclusion

Ret2libc / ret2got / ret2plt

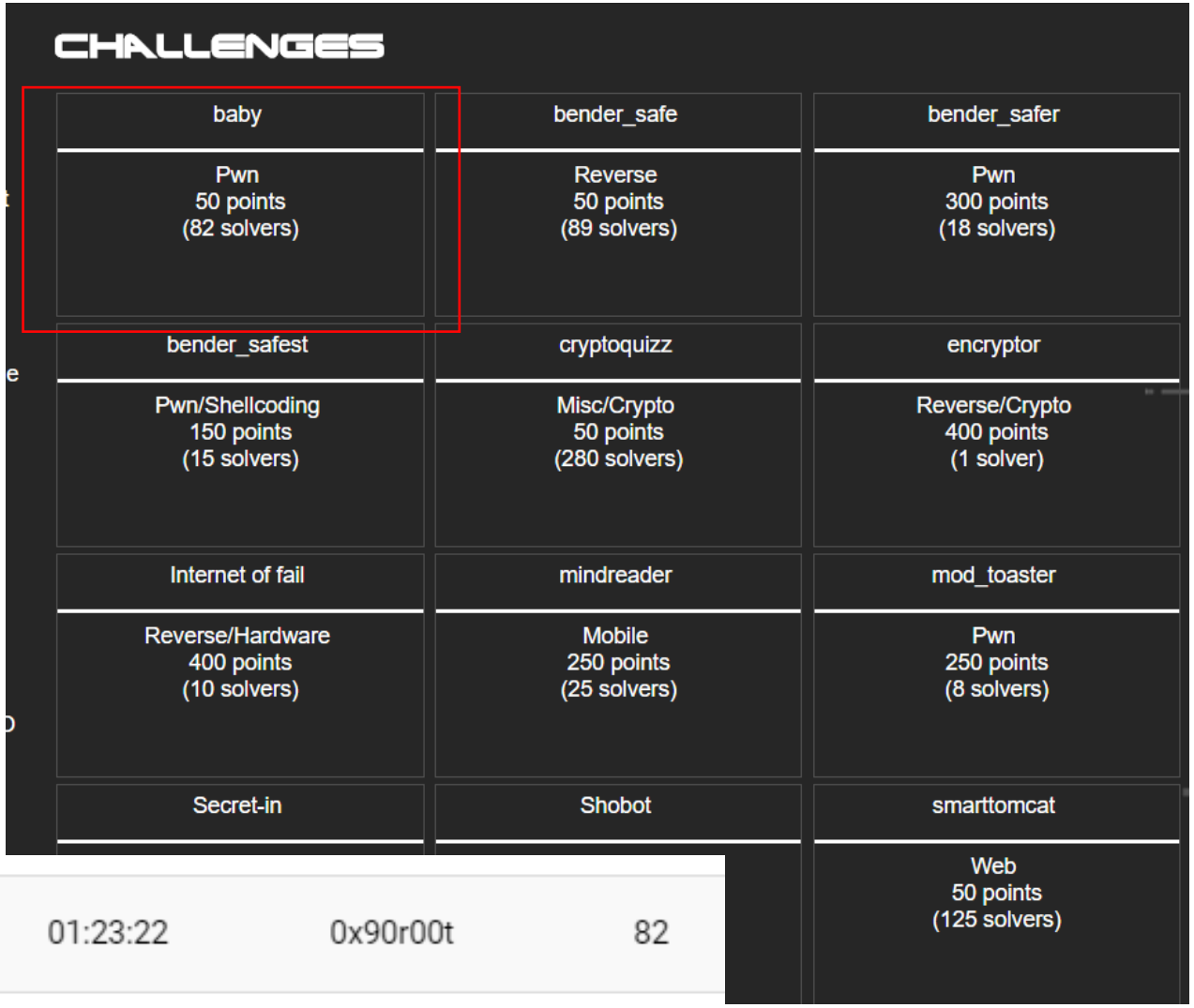
- Is “only” able to execute arbitrary library functions

ROP

- Can execute arbitrary code by re-using existing code from program or shared libraries
- Can defeat often ASLR + DEP
- Can defeat ASLR+DEP+PIE, with information disclosure

Insomnihack Teaser

- Insomnihack: Security Conference in Geneva
- Got a Teaser CTF (Capture the Flag)
- Baby challenge:
  - Forking Server
  - 64 bit
  - ASLR
  - PIE
  - Stack Canary



baby	Pwn	50	01:23:22	0x90r00t	82