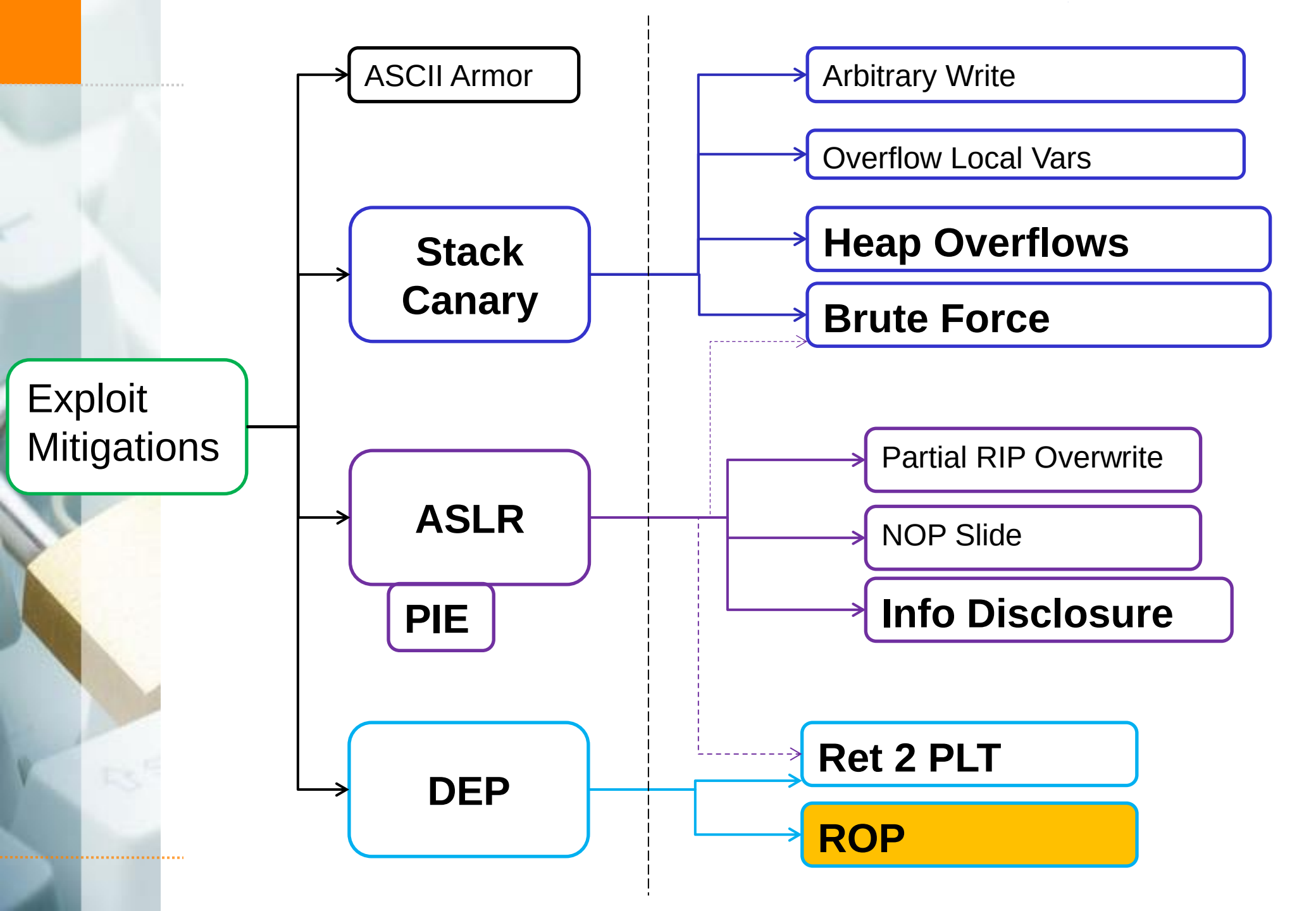
Return Oriented Programming

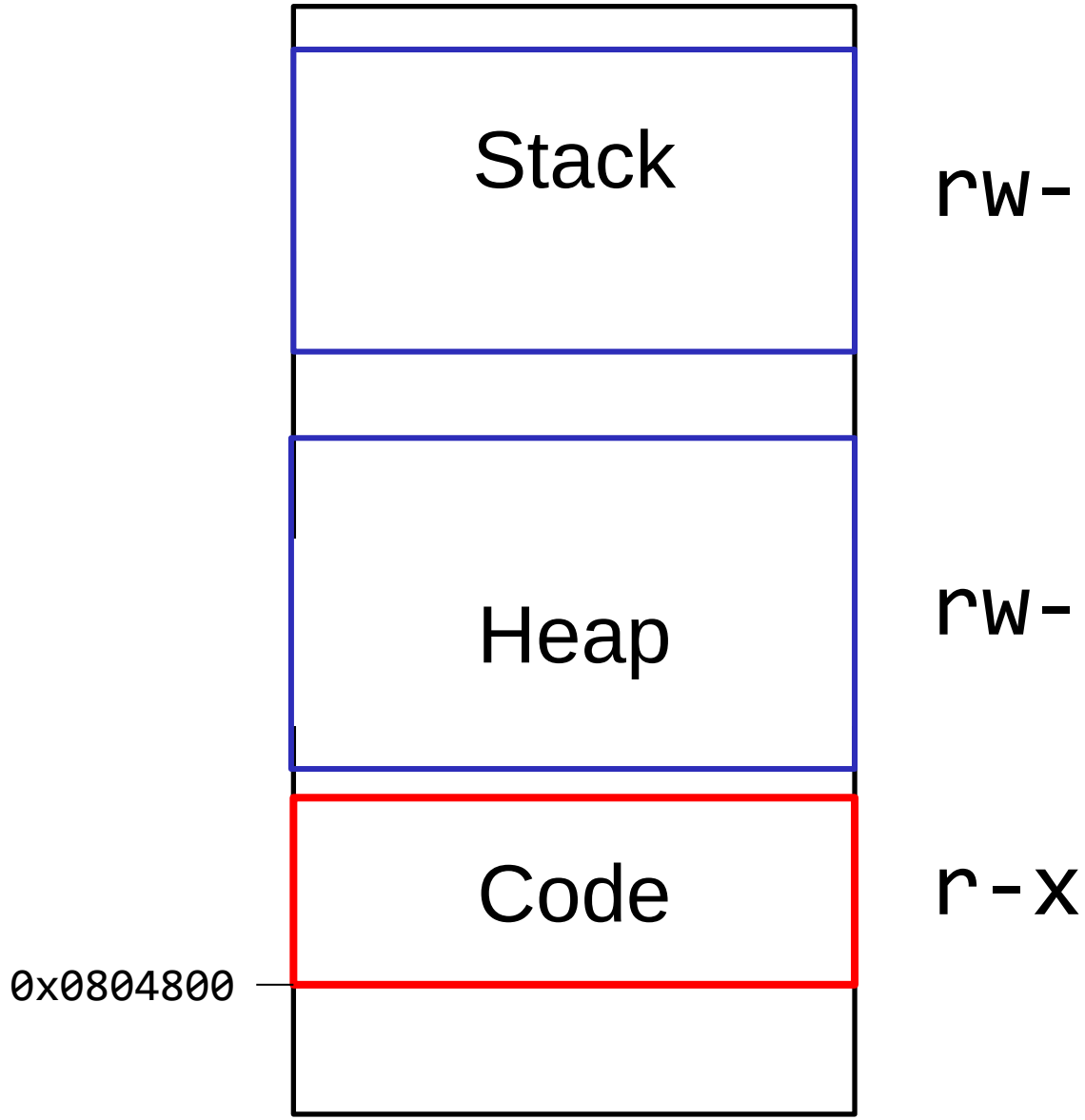
ROP

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch


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



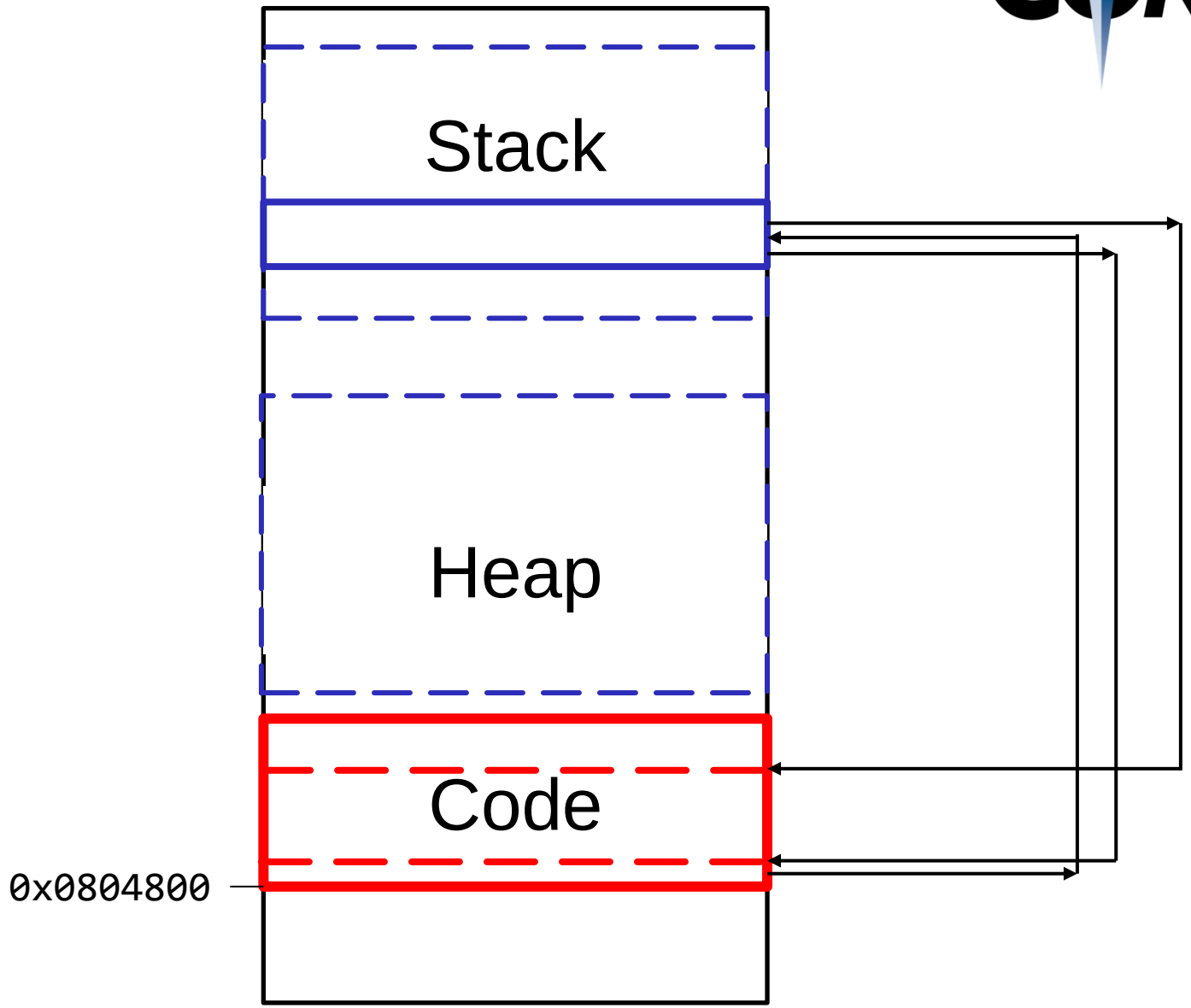
Exploiting: DEP - ROP

DEP does not allow execution of uploaded code

But what about **existing code**?

ROP: smartly put together existing code

Exploiting: DEP - Memory Layout

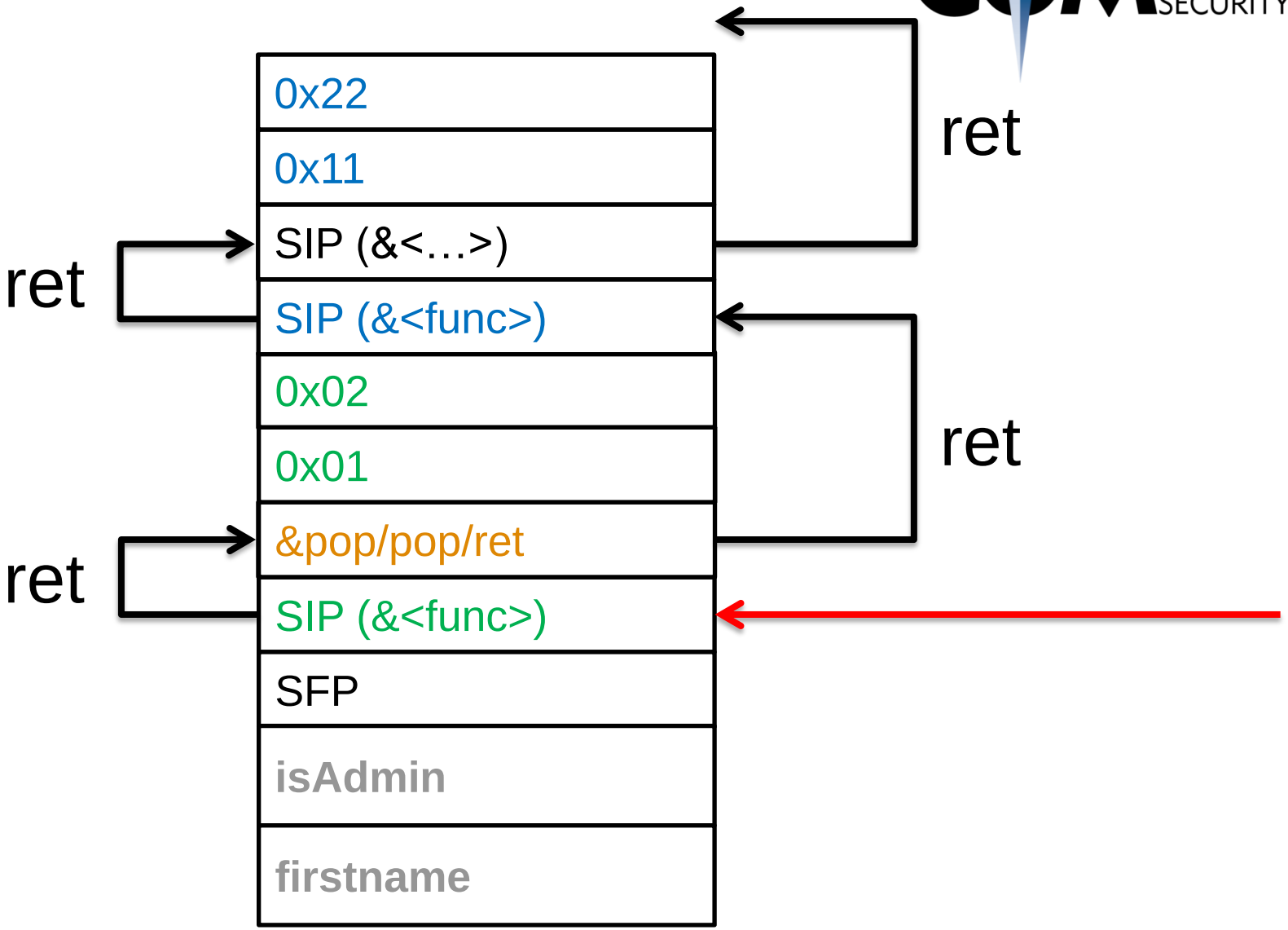


ROP In One Slide

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

ROP Preview


0x22
0x11
SIP (&<…>)
SIP (&<func>)
0x02
0x01
&pop/pop/ret
SIP (&<func>)
SFP
isAdmin
firstname
ret
ret
ret
ret

ROP

Gadgets

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

Exploiting DEP - ROP

What is ROP?

Smartly chain gadgets together to execute arbitrary code

Gadgets:

- Some sequence of code, followed by a RET

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
```

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
```

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
```

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

ROP Introduction

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

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

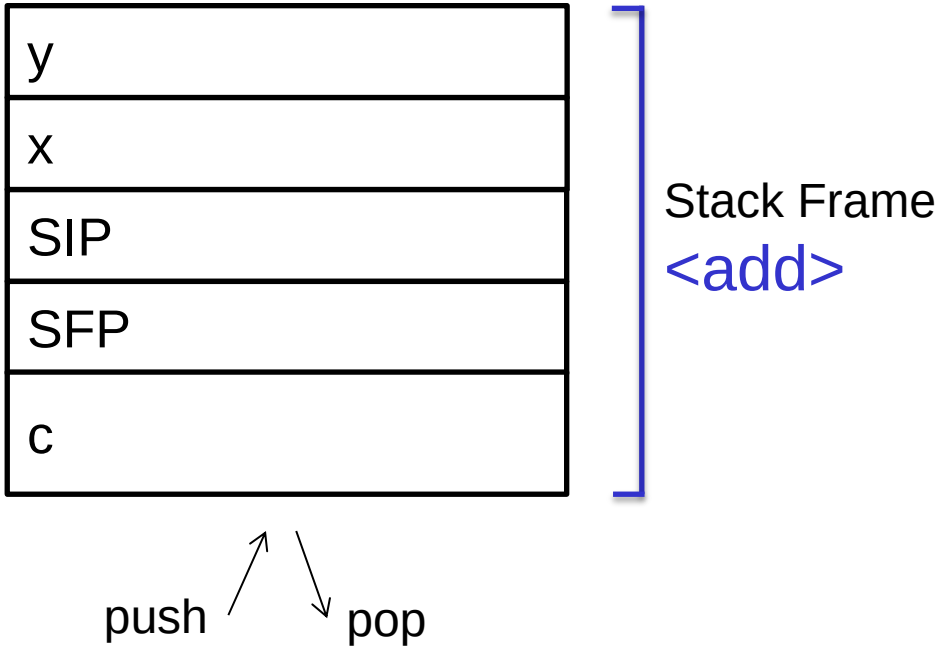
Argument 2 for <add>

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… fuck that, 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?

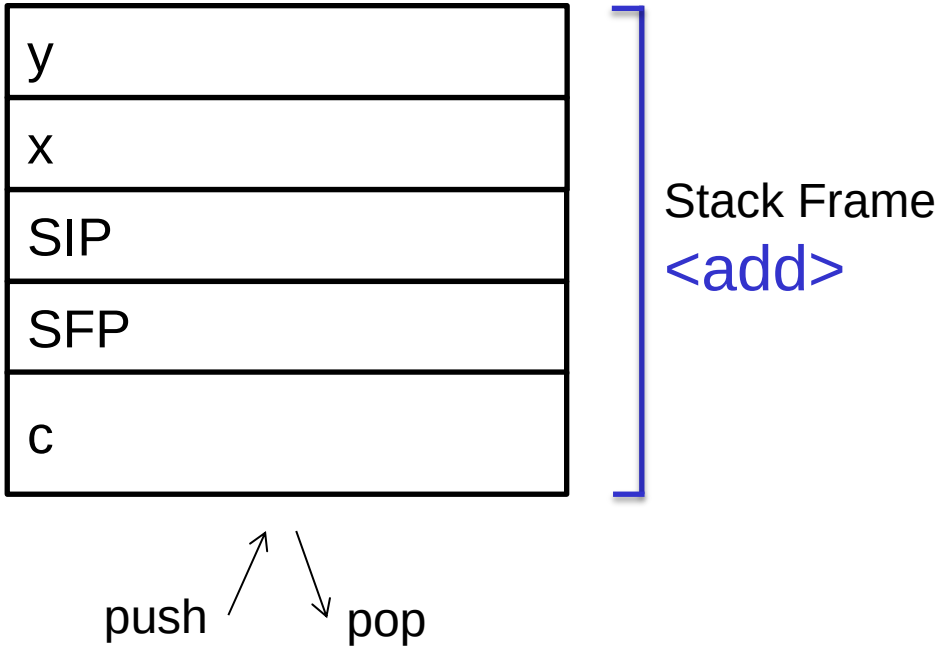
Argument 2 for <add>

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?

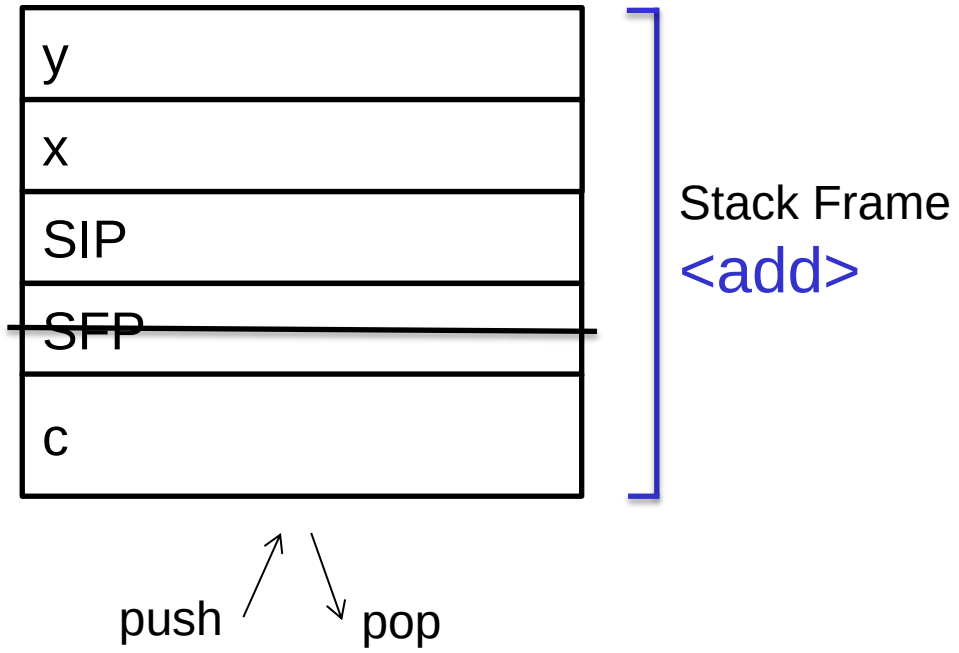
Argument 2 for <add>

Argument 1 for <add>

Saved IP (&return)

~~Saved Frame Pointer~~

Local Variables <add>



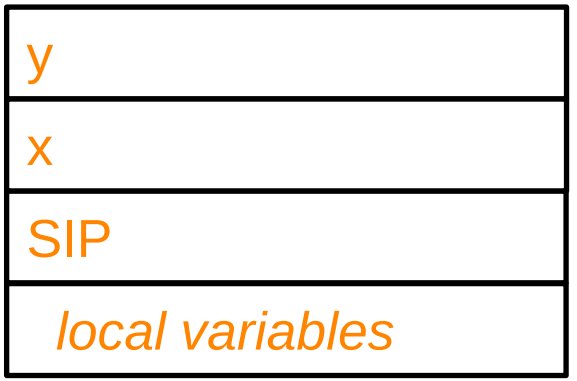
Why does ROP work

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

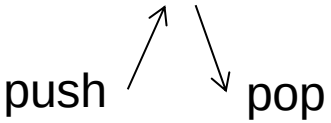
Argument 2

Argument 1

Saved IP (&next instruction)



Note: SIP gets pushed by “call”

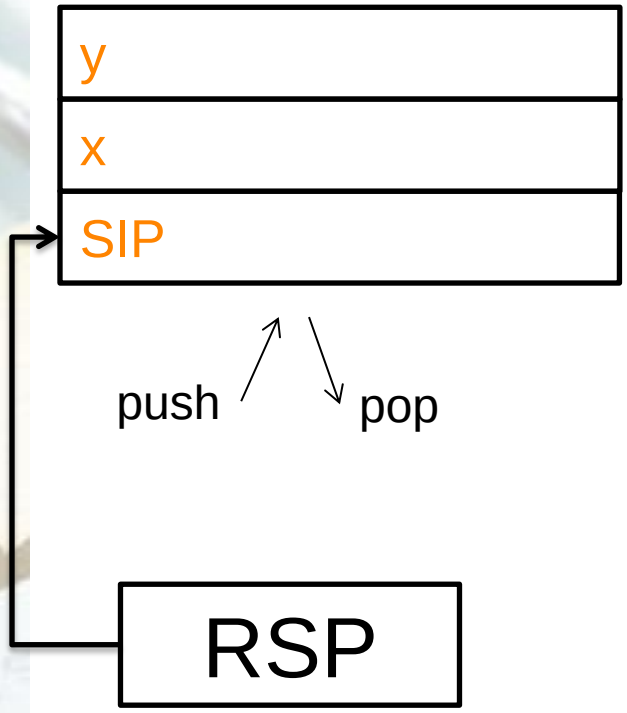


Why does ROP work

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

(after "call", inside the function)

Stack:



Function:

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

ROP: Remainder: Normal Call

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

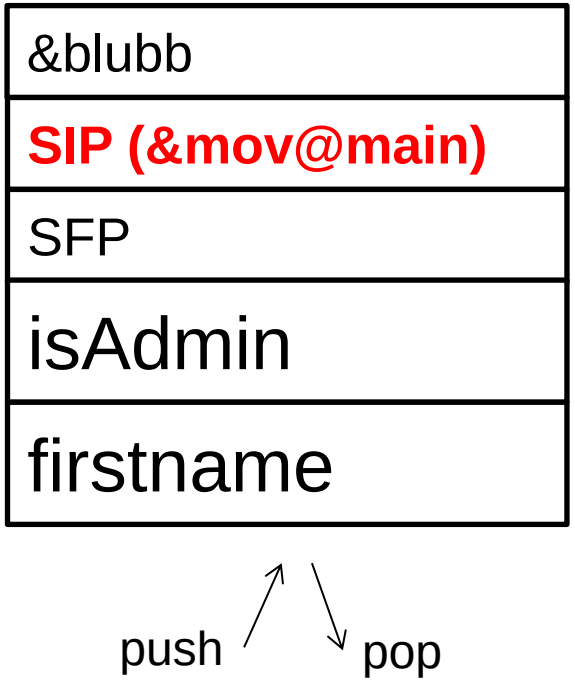
Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

ROP: Remainder: Normal Call

Lets check again the normal call convention process

ROP: Remainder: Normal Call

Reminder: Buffer Overflow, Pre-Overflow:



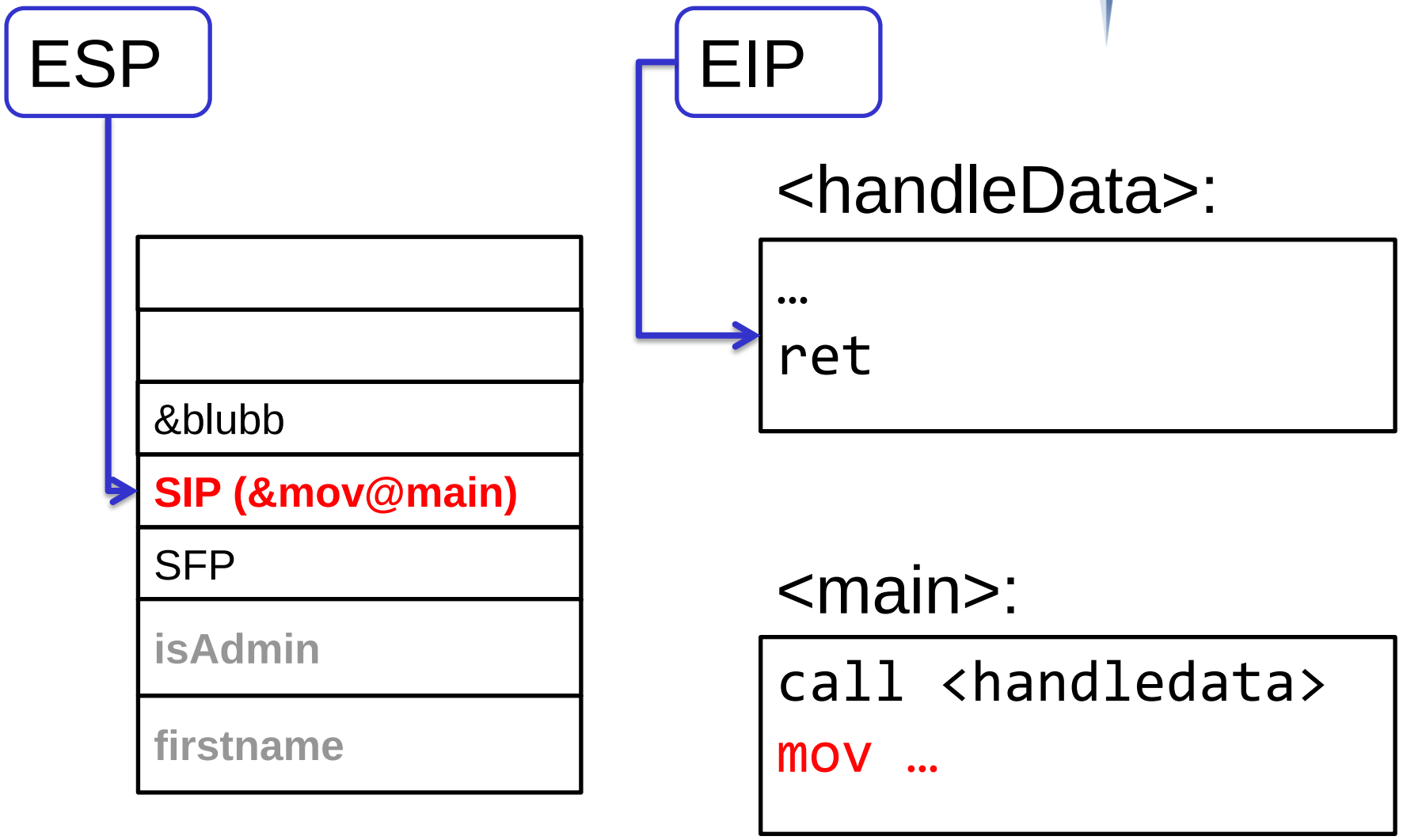
Argument arg1 for <handleData>

Saved IP

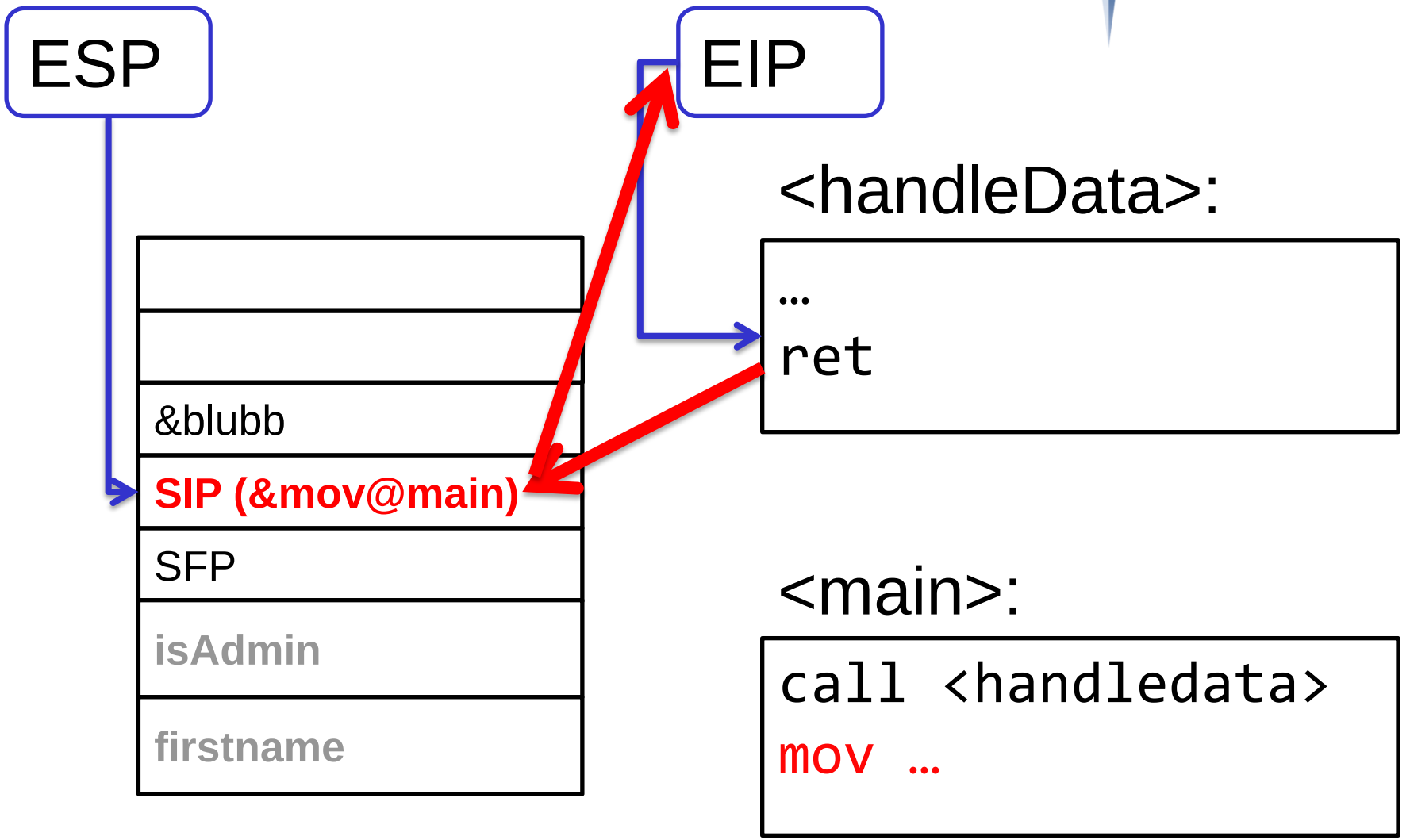
Saved Frame Pointer

Local Variable 1

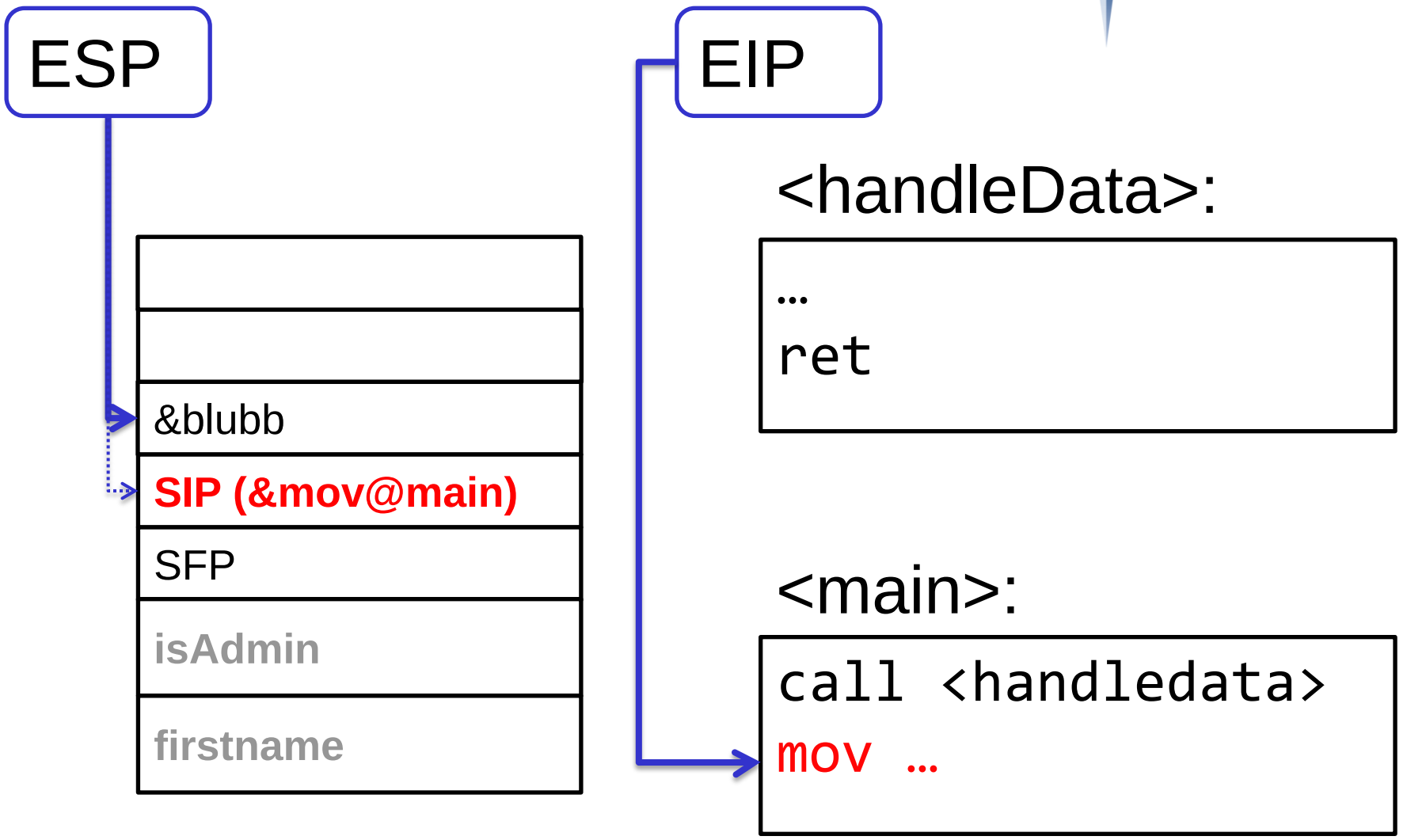
ROP: Remainder: Normal Call



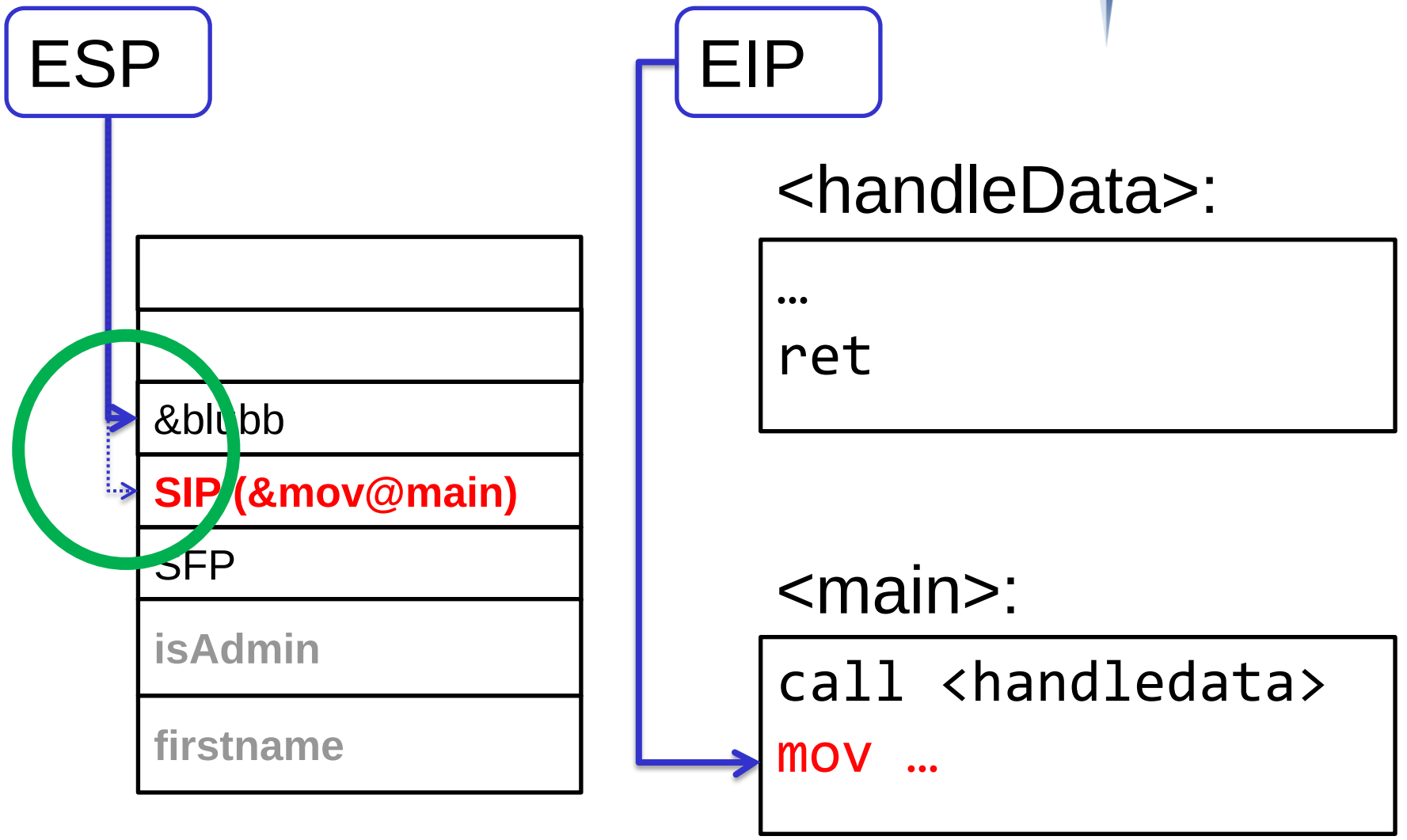
ROP: Remainder: Normal Call



ROP: Remainder: Normal Call



ROP: Remainder: Normal Call



Exploiting: DEP - ROP

Now, lets add the overflow

ROP By Example

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

ROP By Example

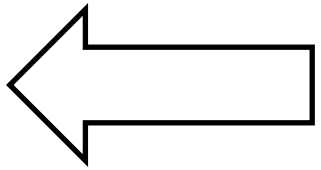
Lets assume we have a nice little “add(int a, int b)” function

Hand written assembly, no standard call convention

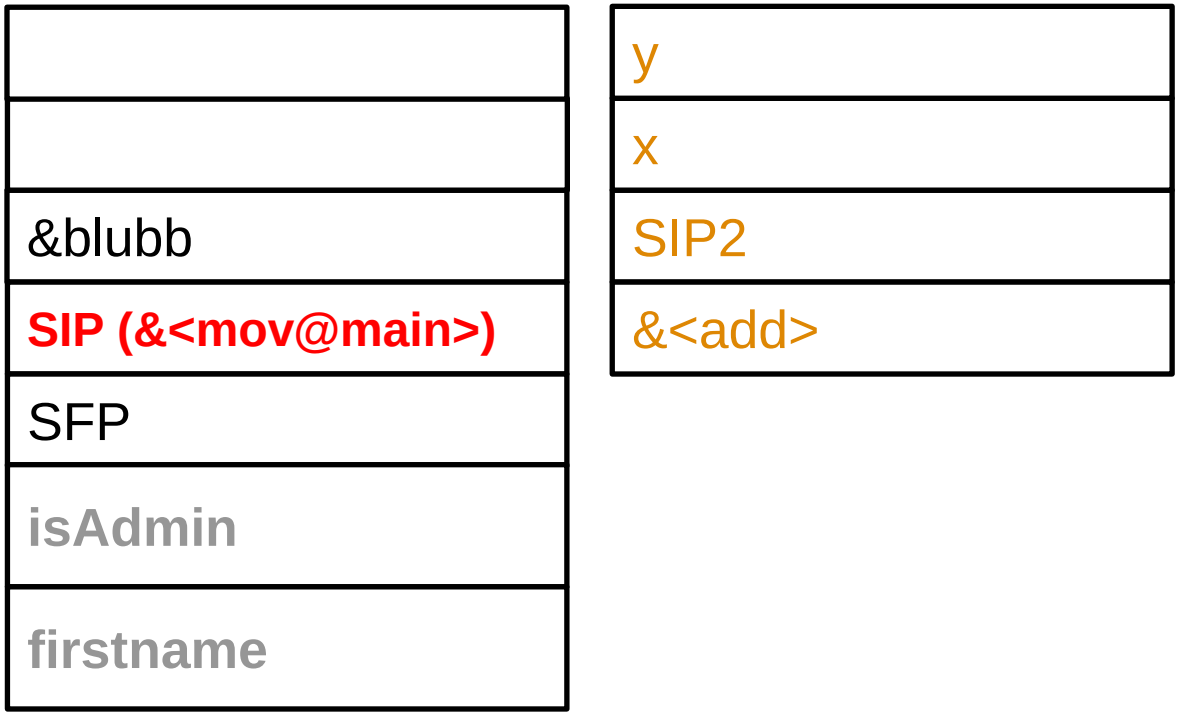
```
add:
  mov     0x8(%esp),%eax
  add     0x4(%esp),%eax
  ret
```

Lets call it…

handleData() Stack:



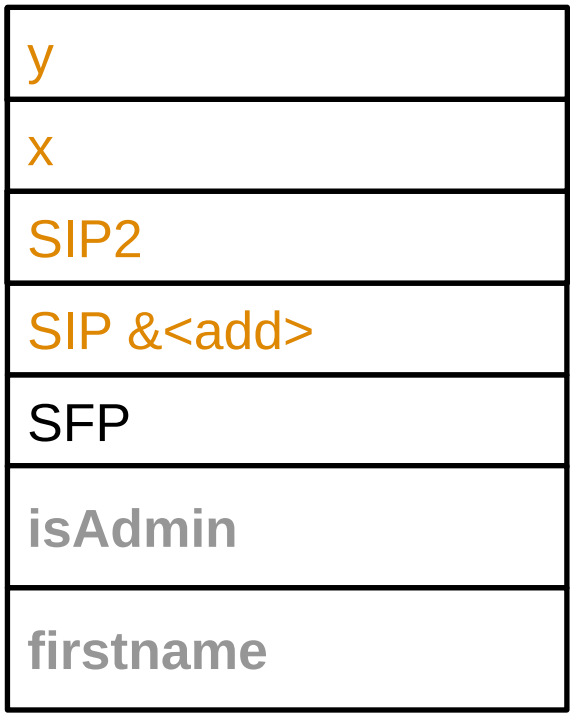
Overflow



Original Stack

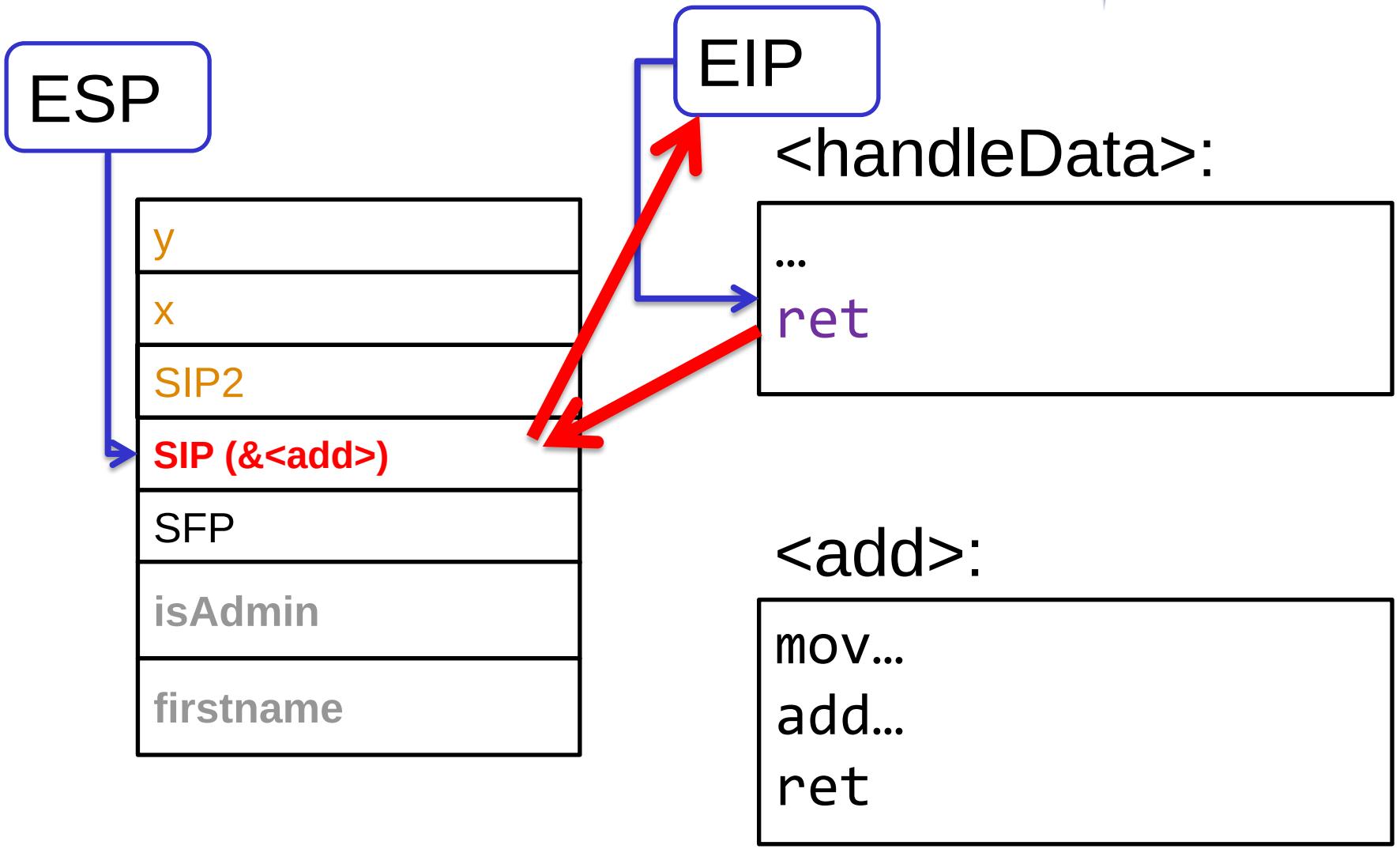
Overflow Data

handleData() Stack:

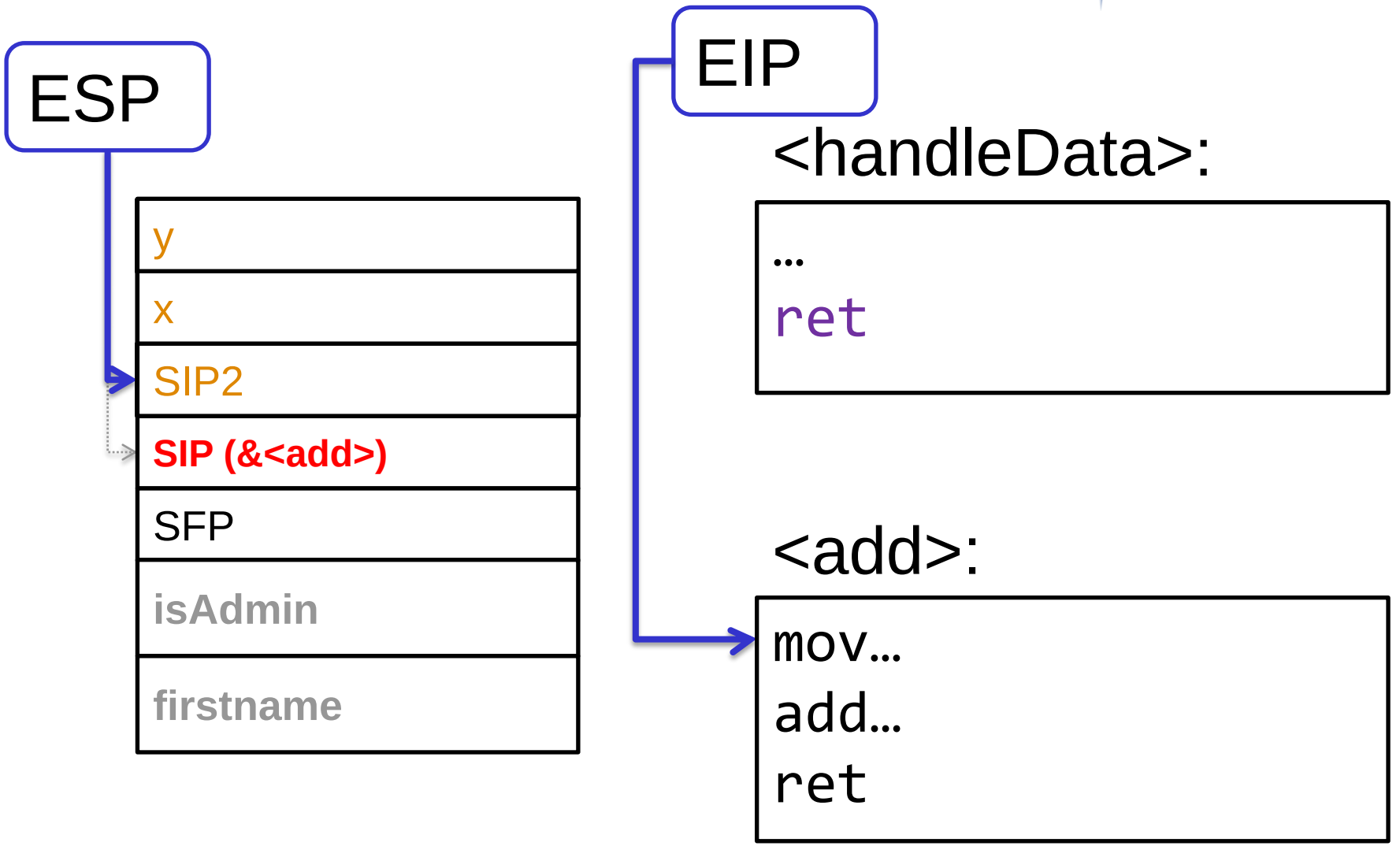


Stack after Overflow

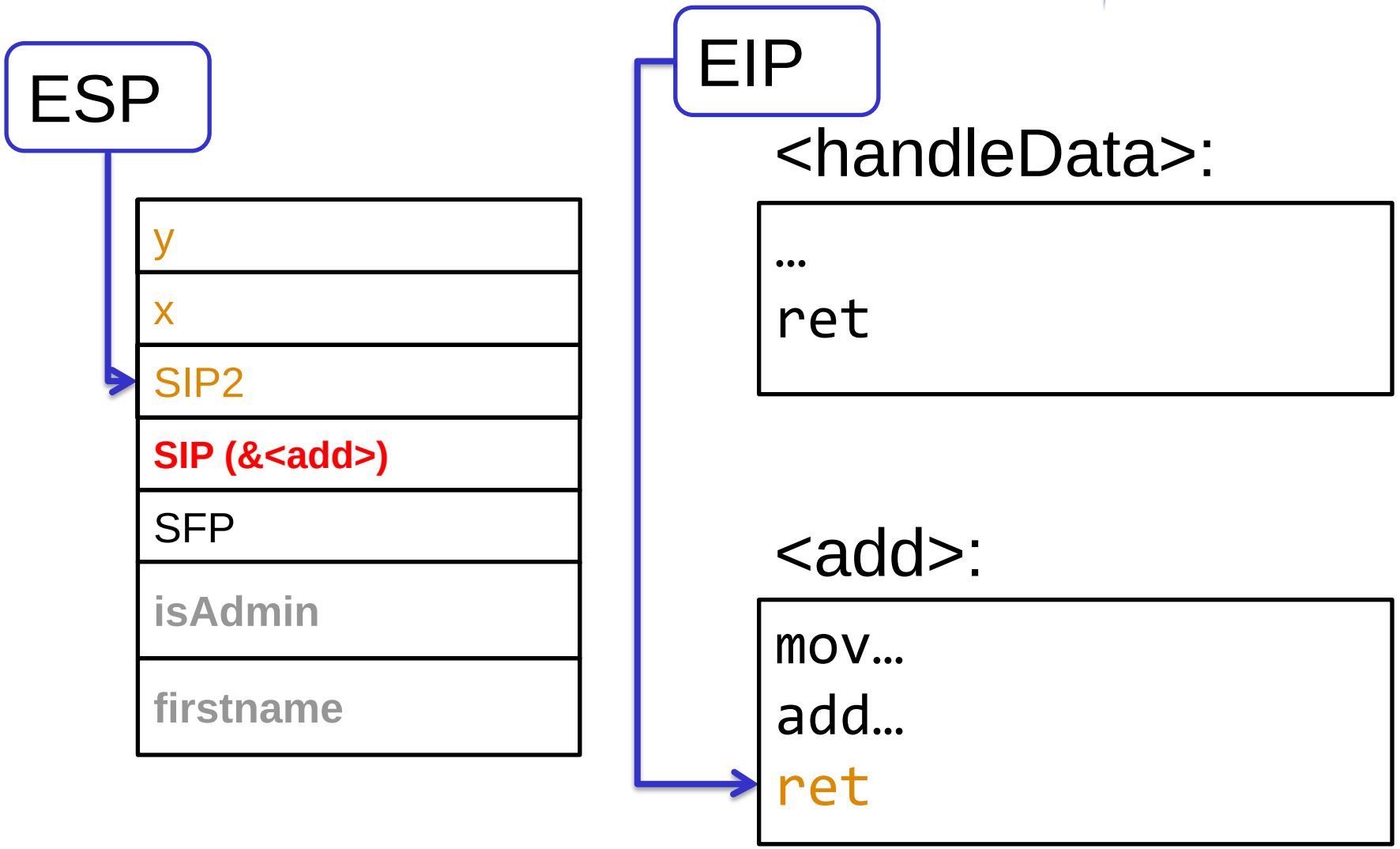
handleData() Stack: On ret@handleData



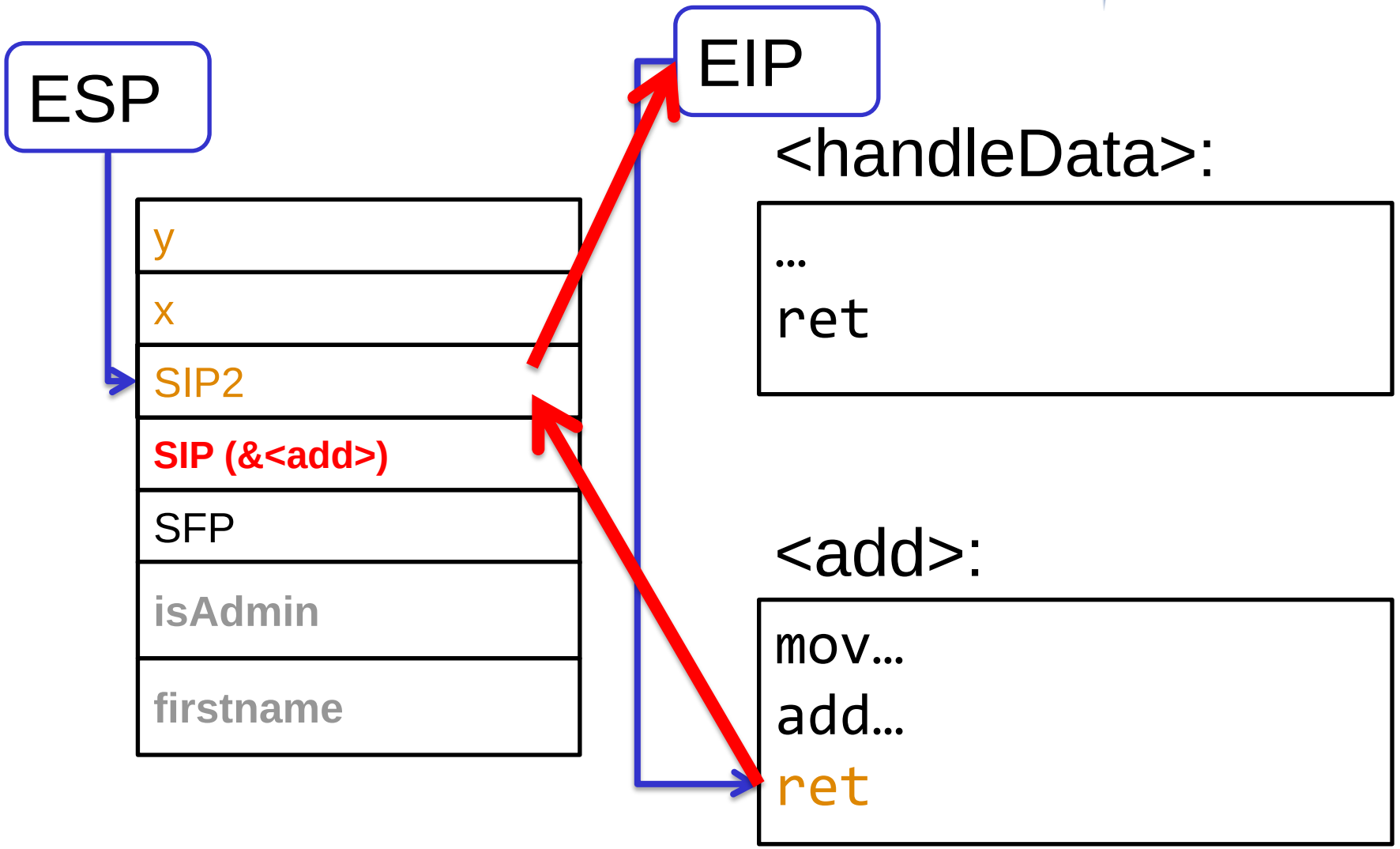
handleData() Stack: After ret@handleData



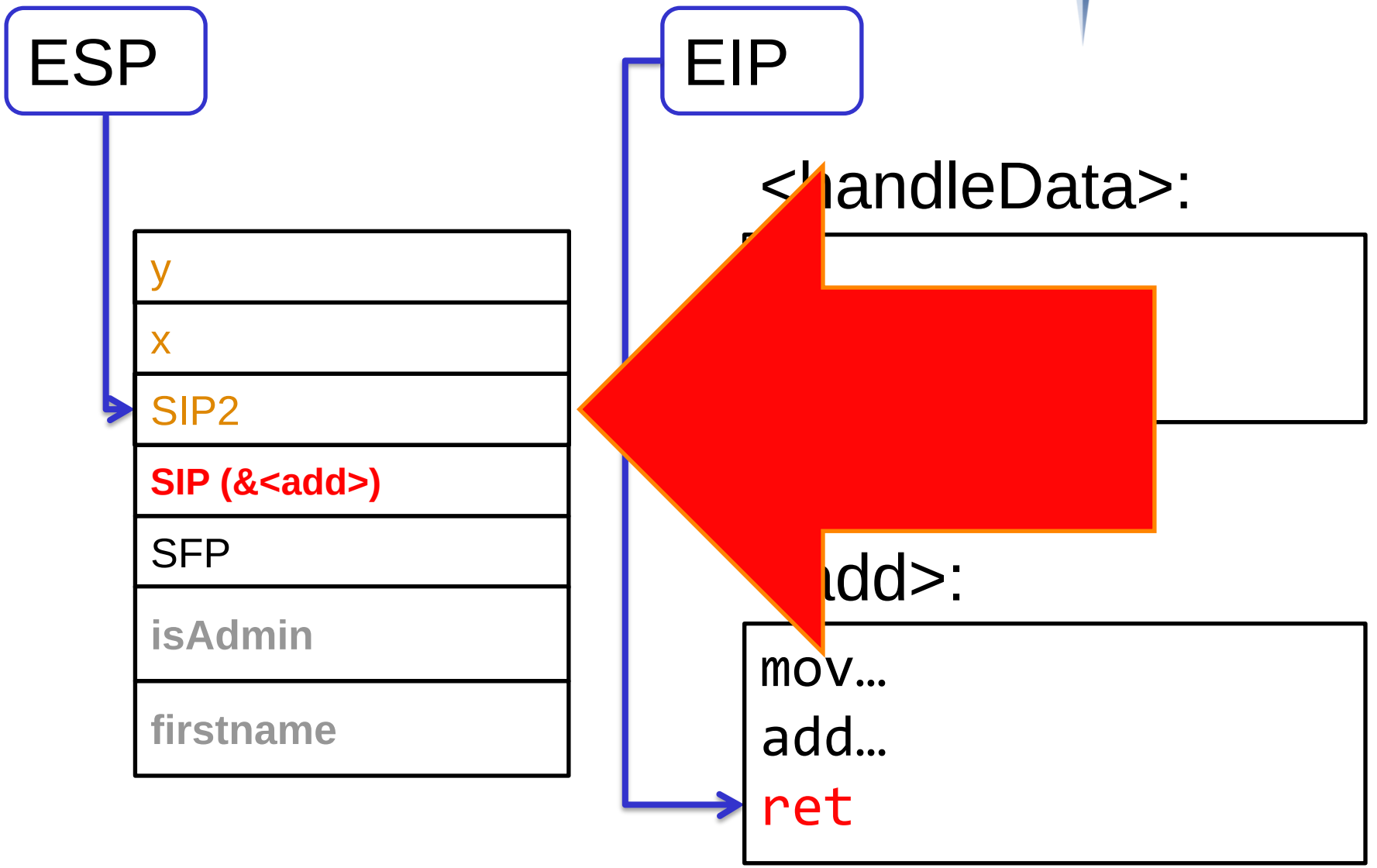
handleData() Stack: On ret@add



handleData() Stack: On ret@add



ROP By Example



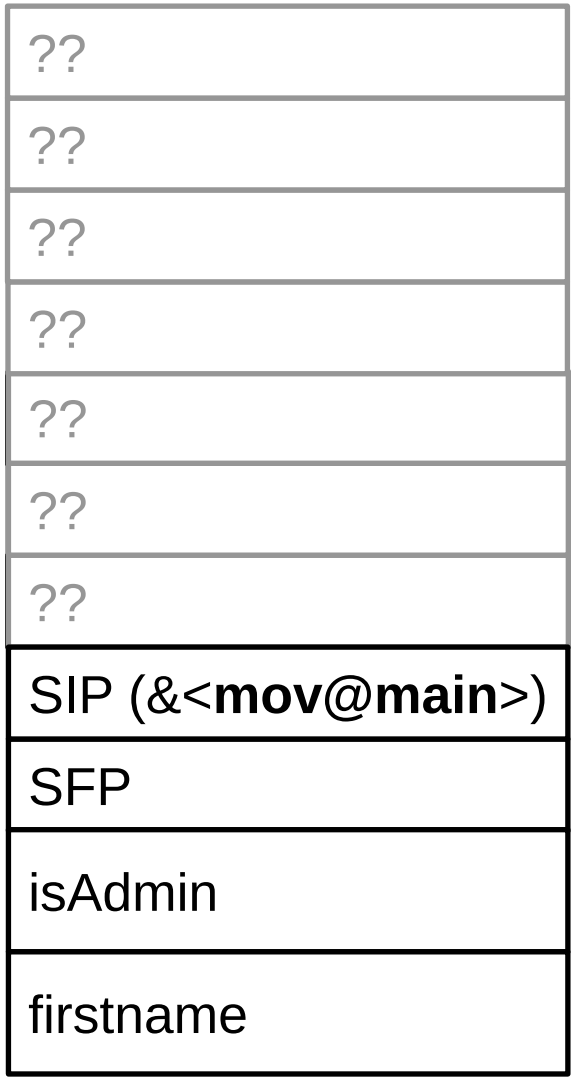
What does this mean?

- 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

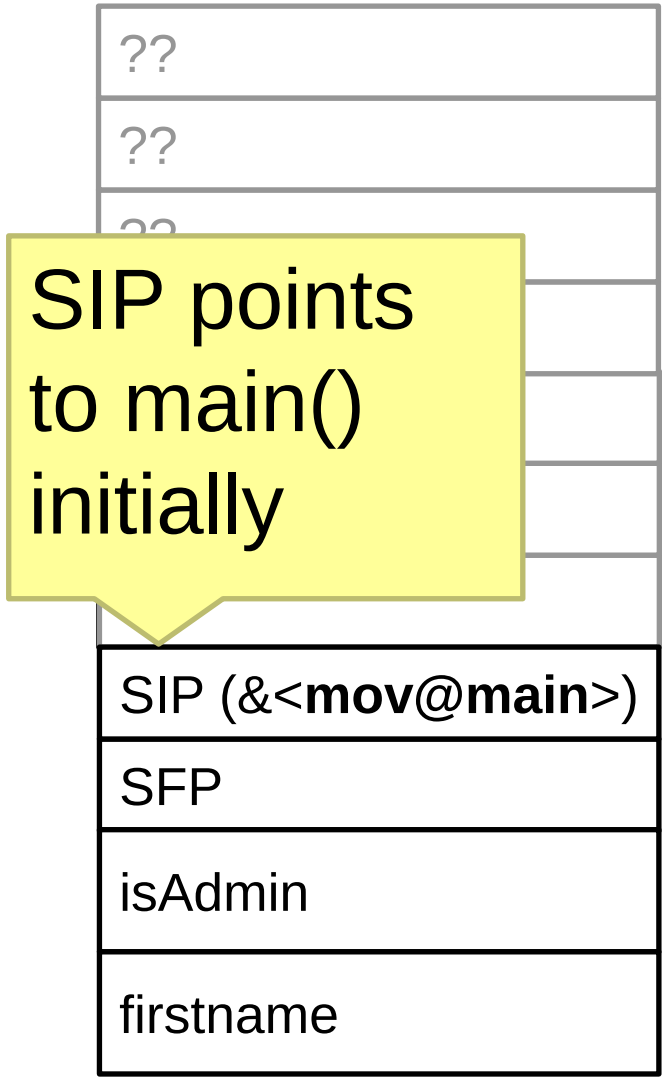


Previous Function Stack Frame

(*handleData*() doesn't/can't know)

Regular *handleData*() Stack Frame

ROP By Example

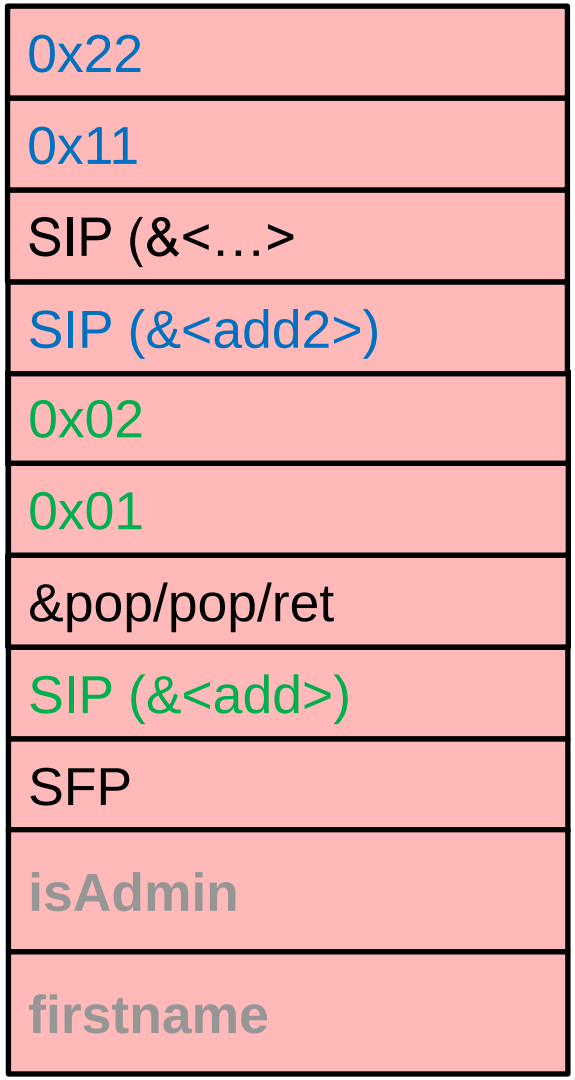


Previous Function Stack Frame

(*handleData*() doesn't/can't know)

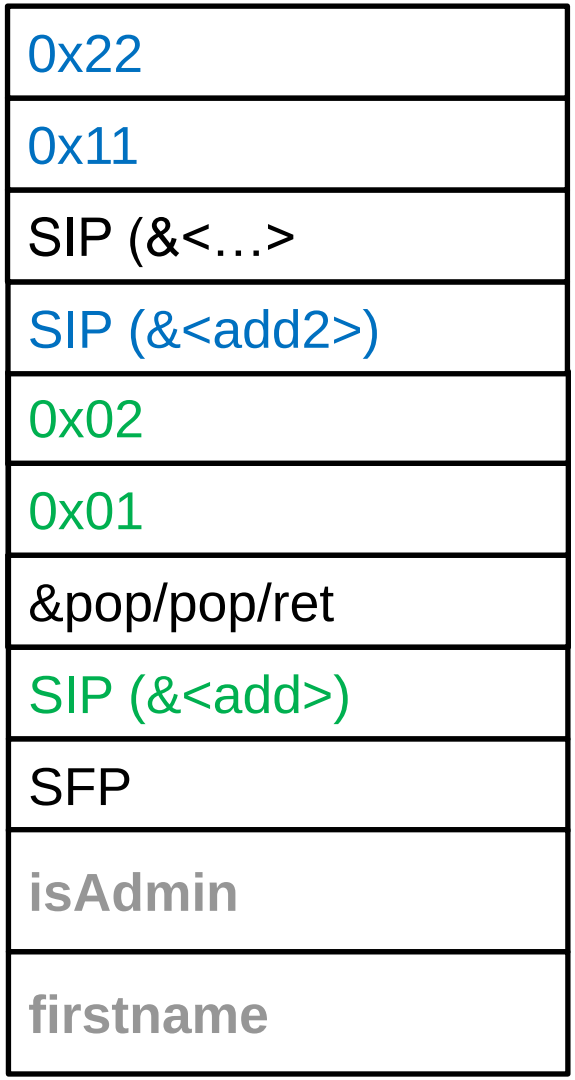
Regular *handleData*() Stack Frame

ROP By Example



The Data we wrote via overflow (red)

ROP By Example

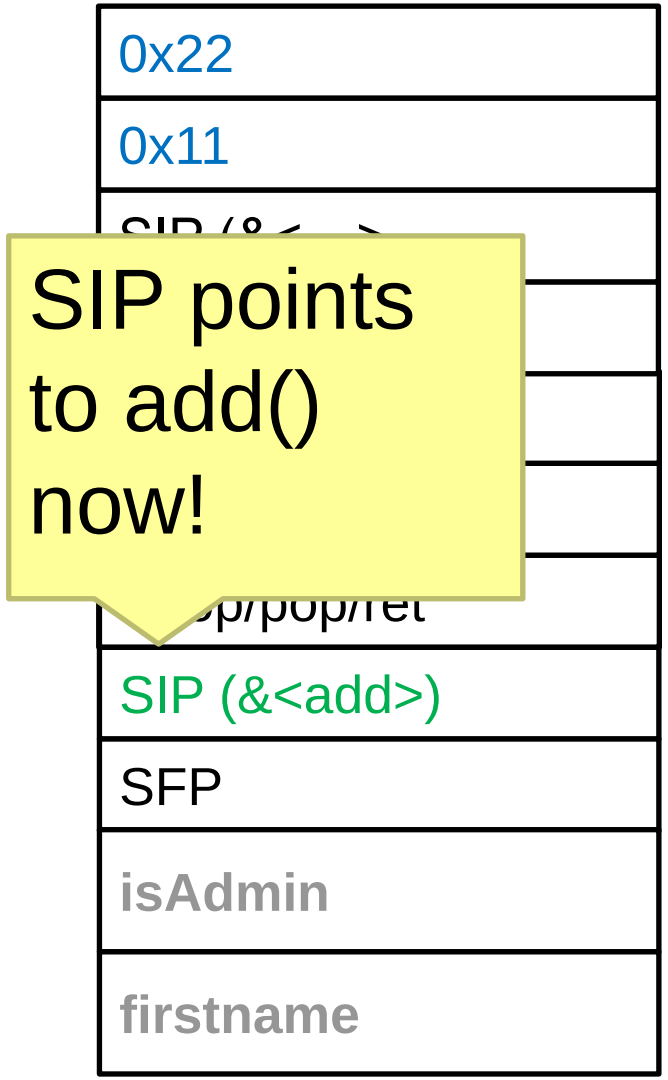


add2 Stuff

add Stuff

Stack Frame <handleData>

ROP By Example

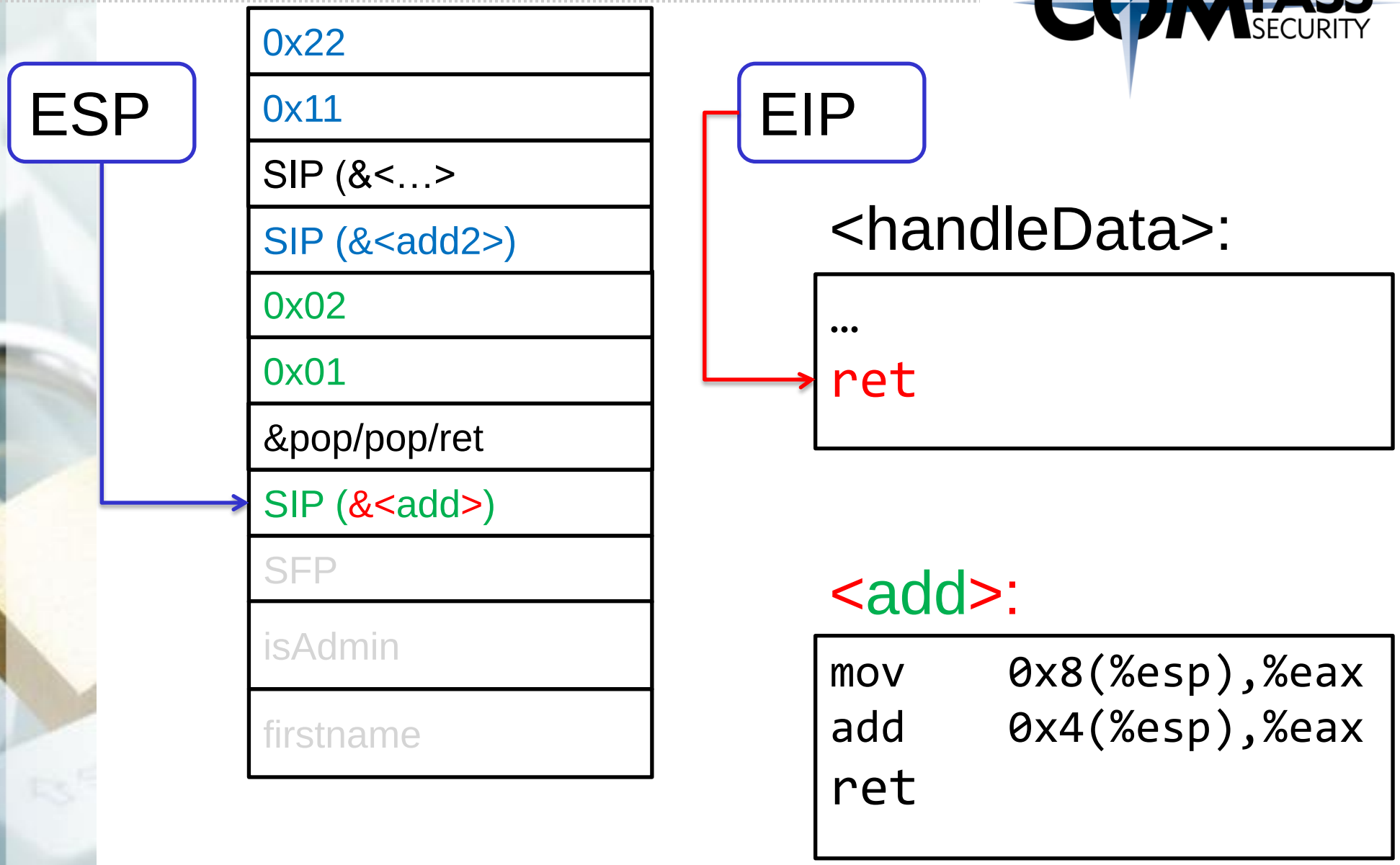


add2 Stuff

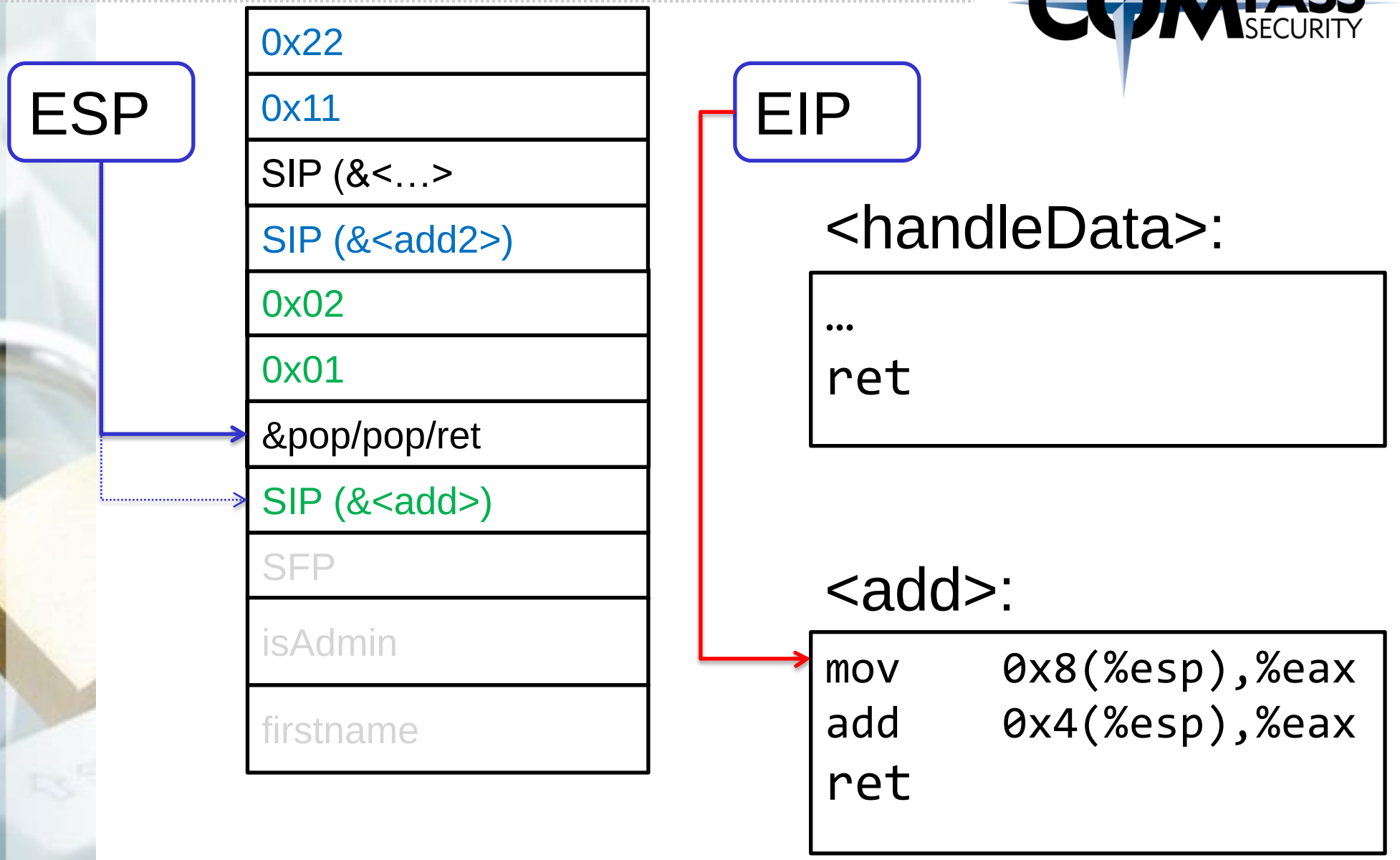
add Stuff

Stack Frame <handleData>

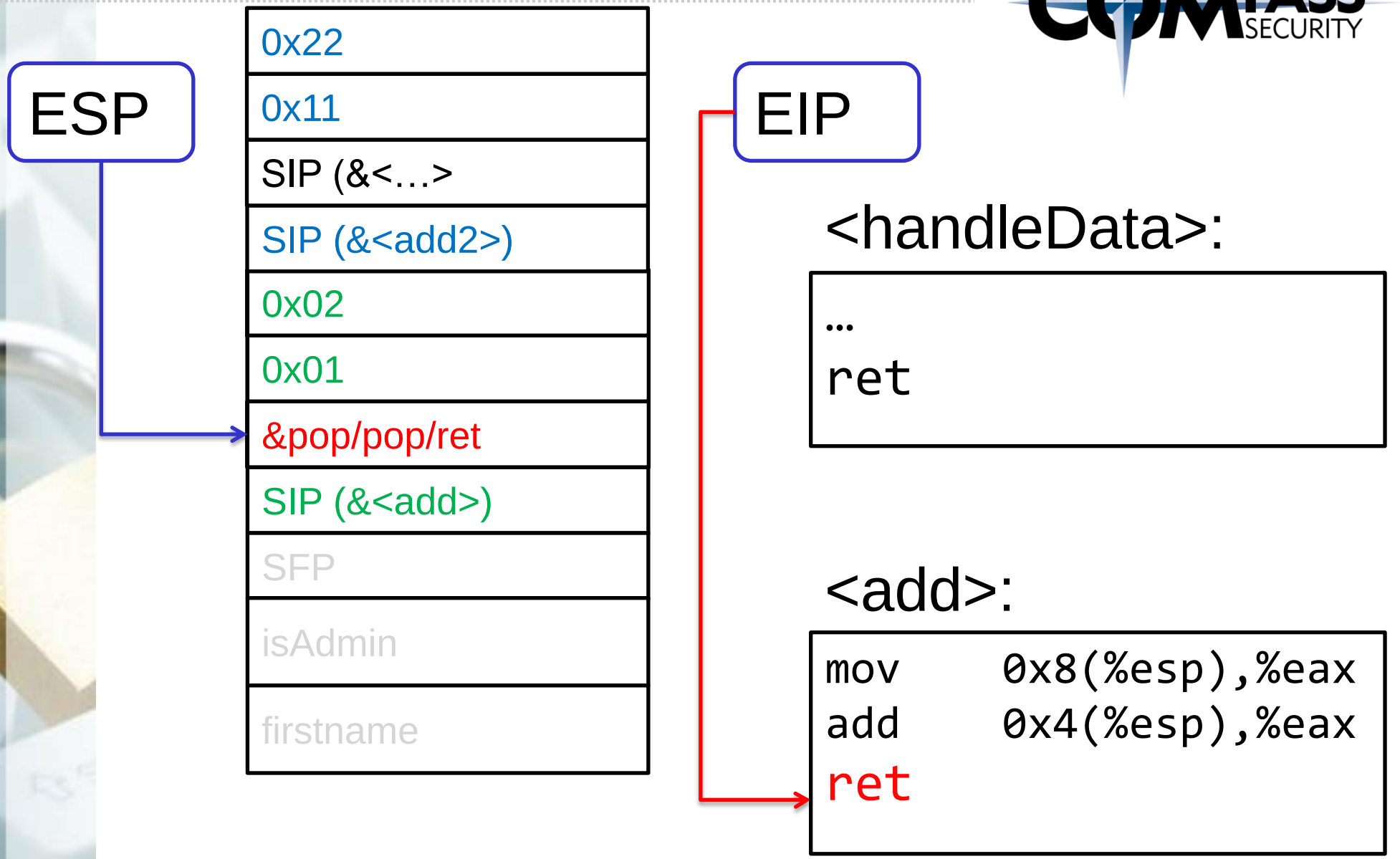
ROP By Example



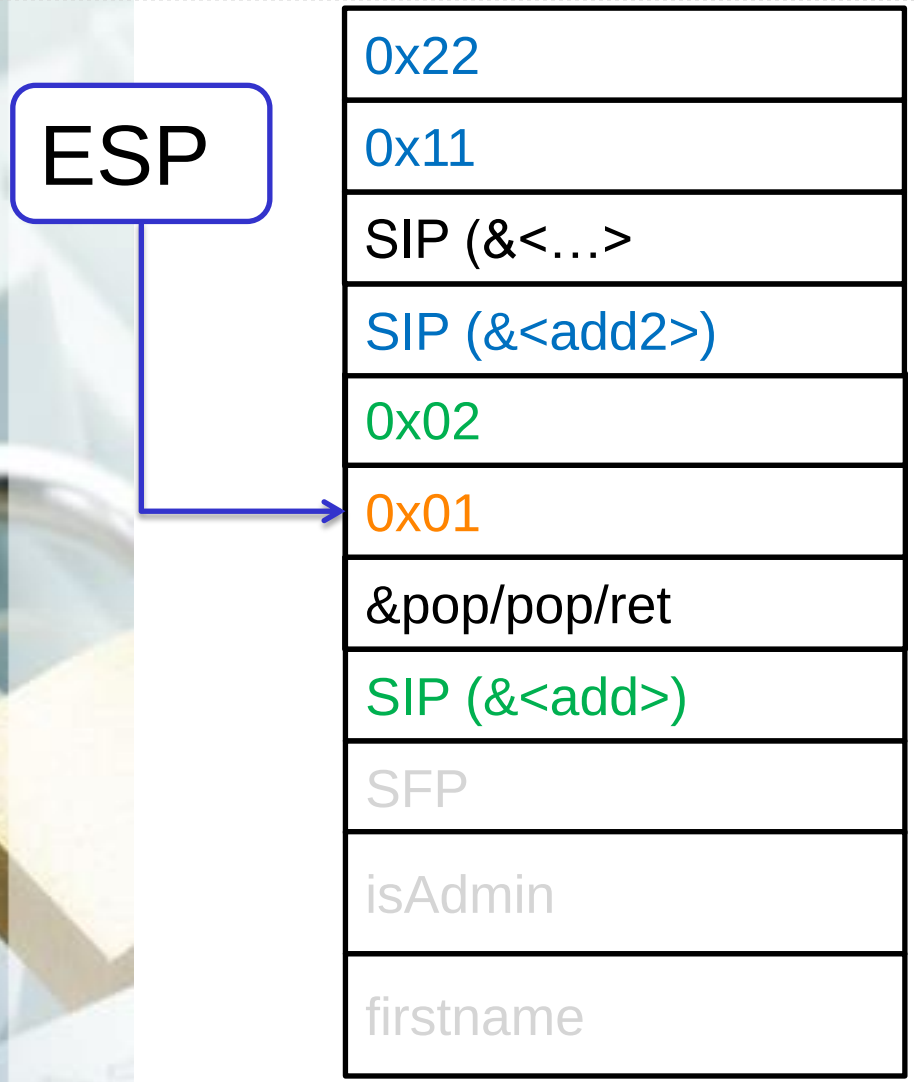
ROP By Example



ROP By Example



ROP By Example



EIP

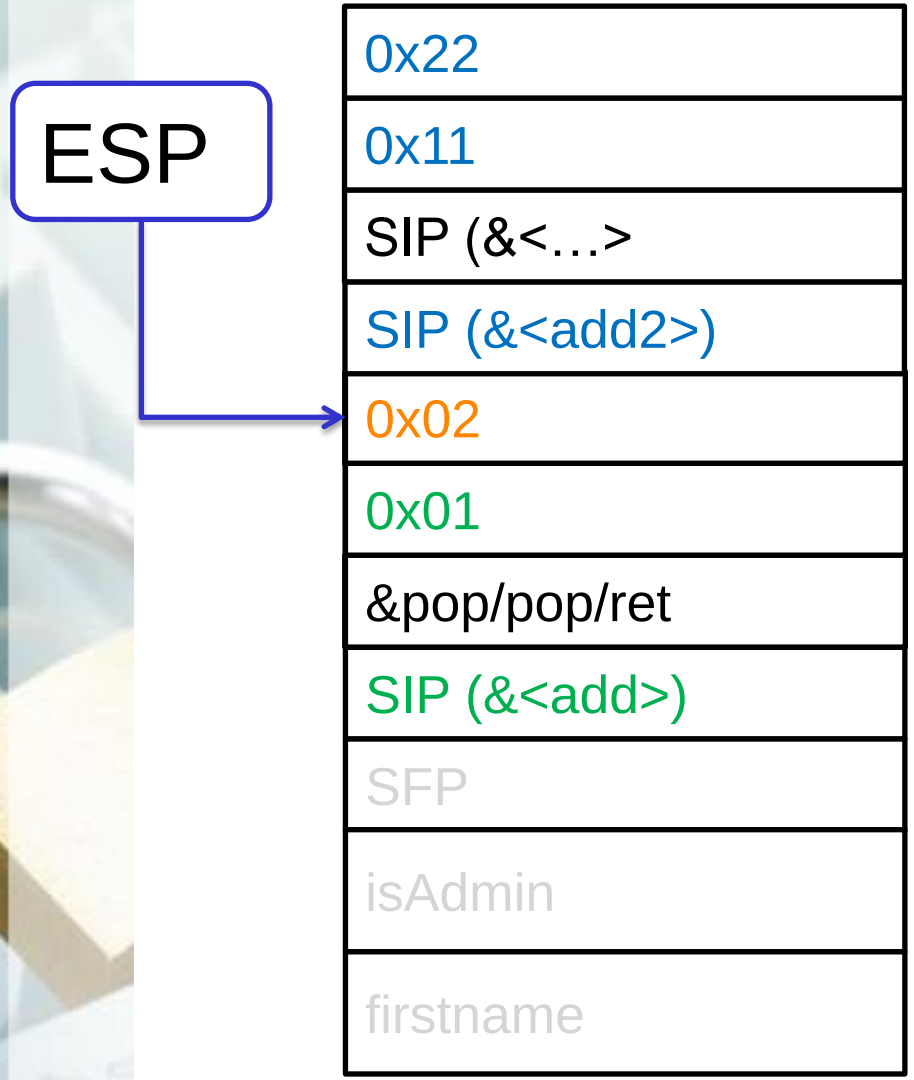
<0xaabb>:

```
pop eax
pop ebx
ret
```

<add>:

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

ROP By Example



EIP

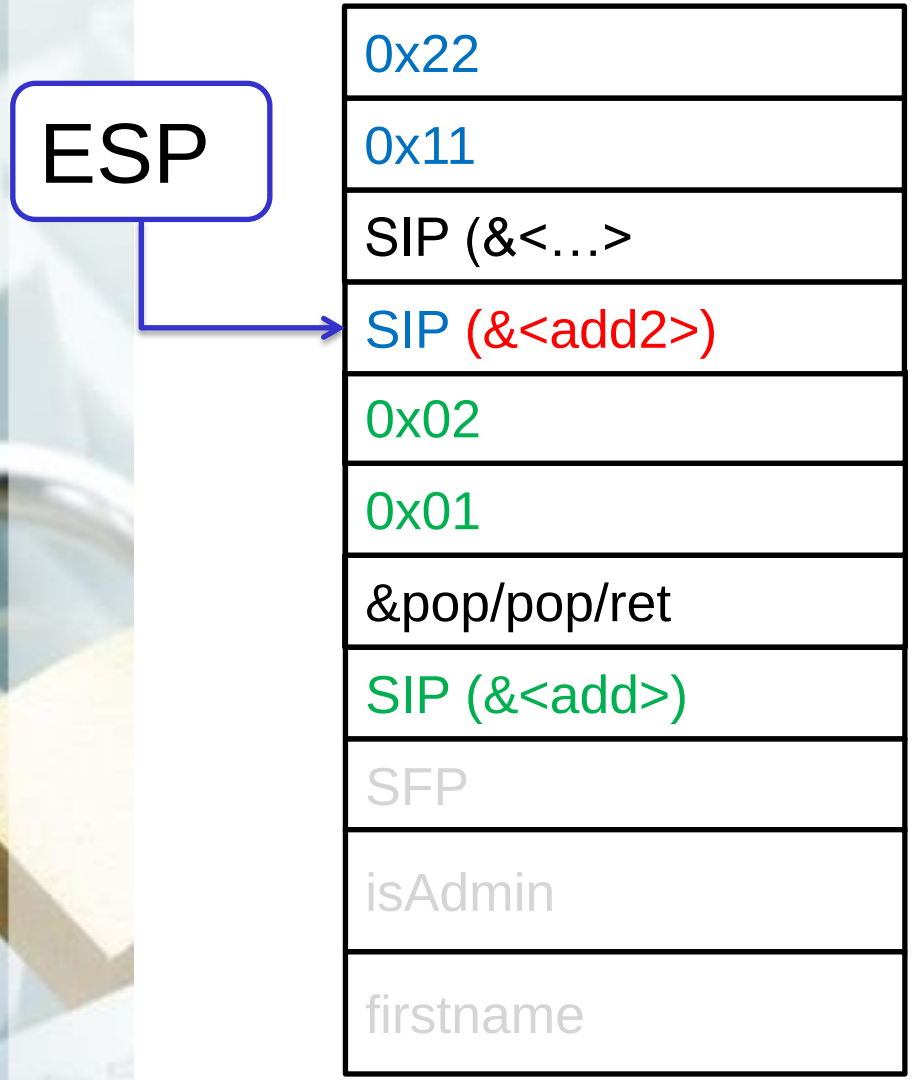
<0xaabb>:

```
pop eax
pop ebx
ret
```

<add>:

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

ROP By Example



EIP

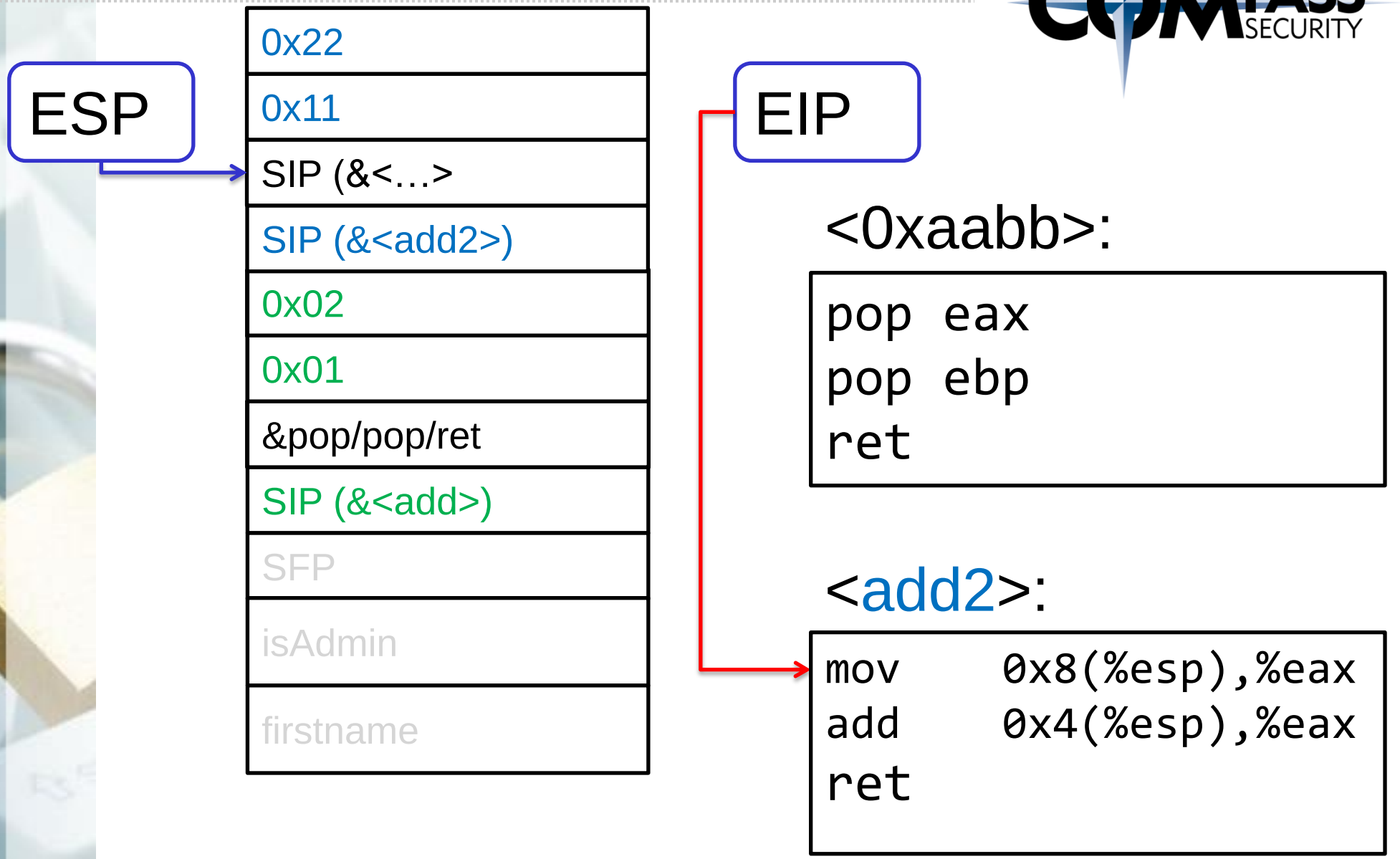
<0xaabb>:

```
pop eax
pop ebp
ret
```

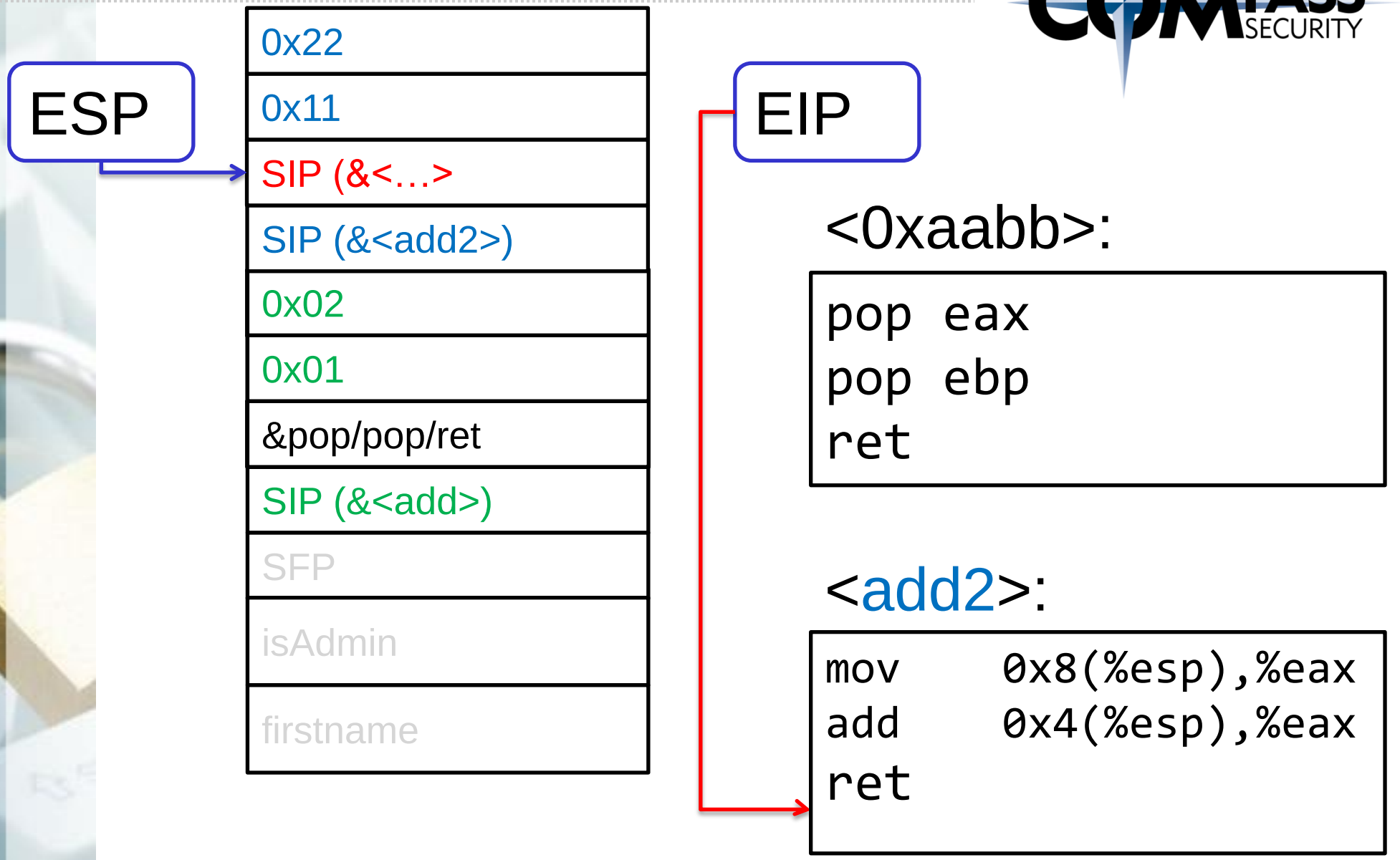
<add2>:

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

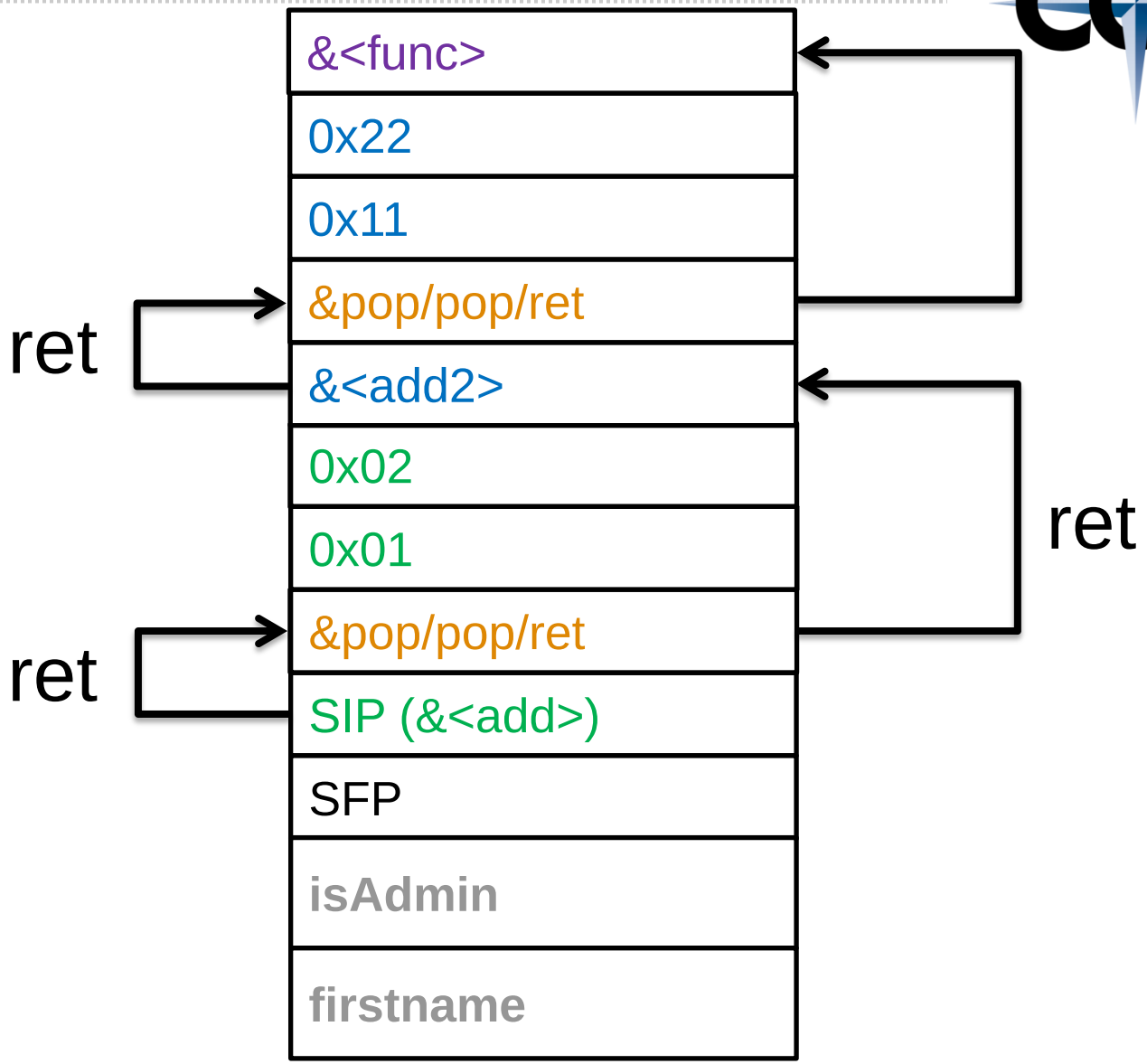
ROP By Example



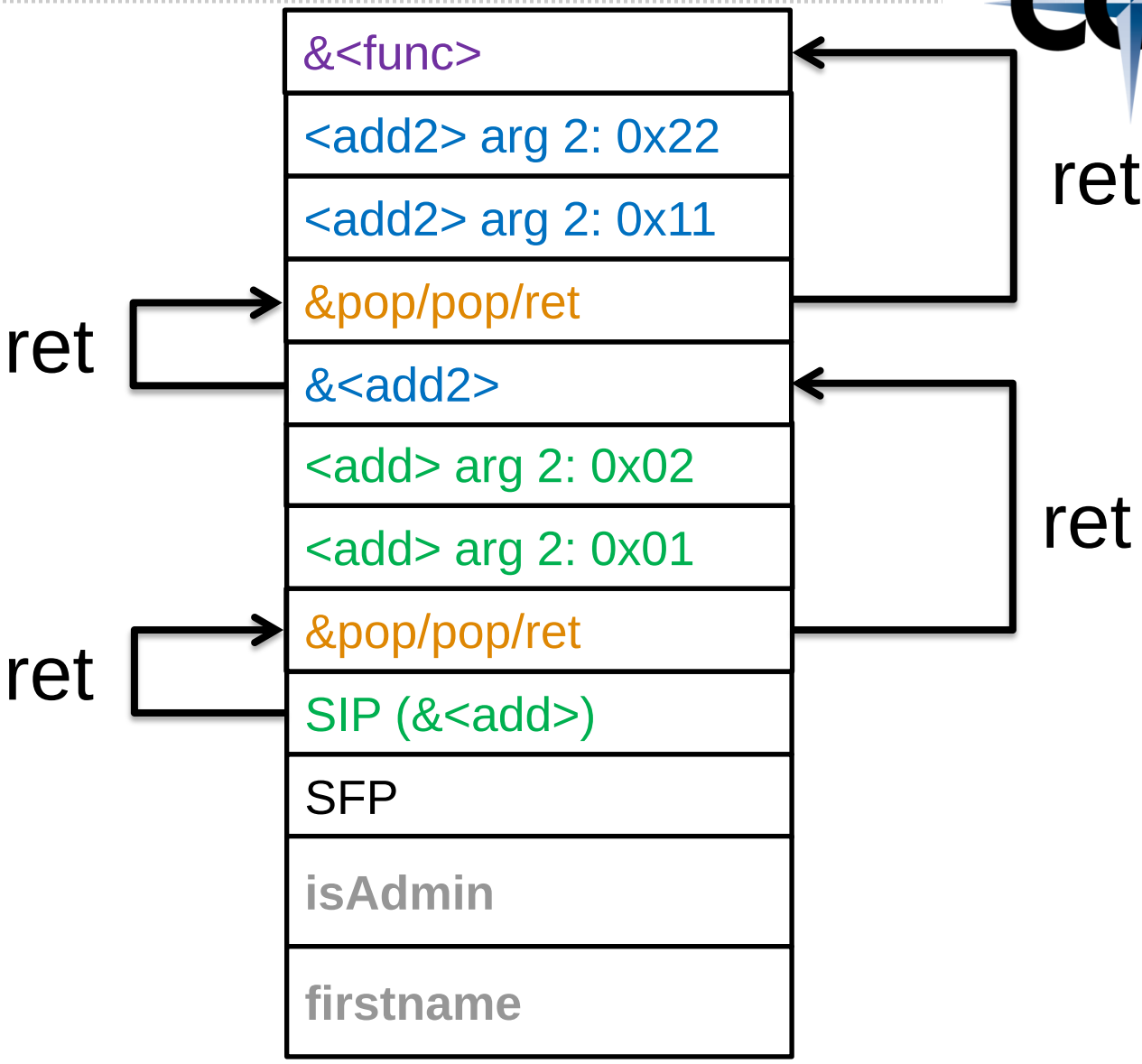
ROP By Example



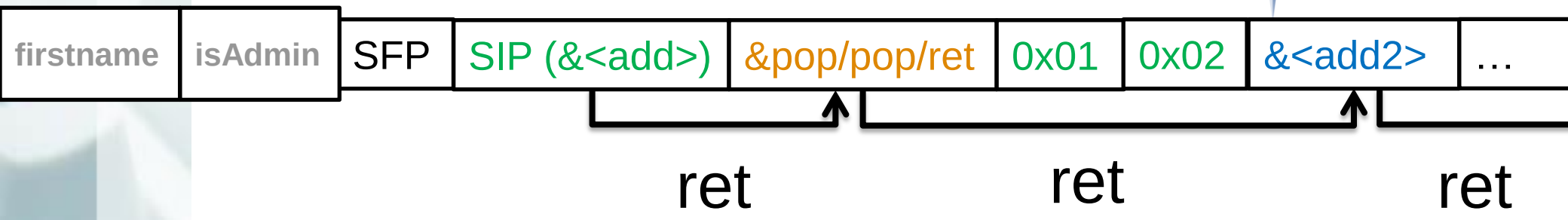
ROP By Example



ROP By Example



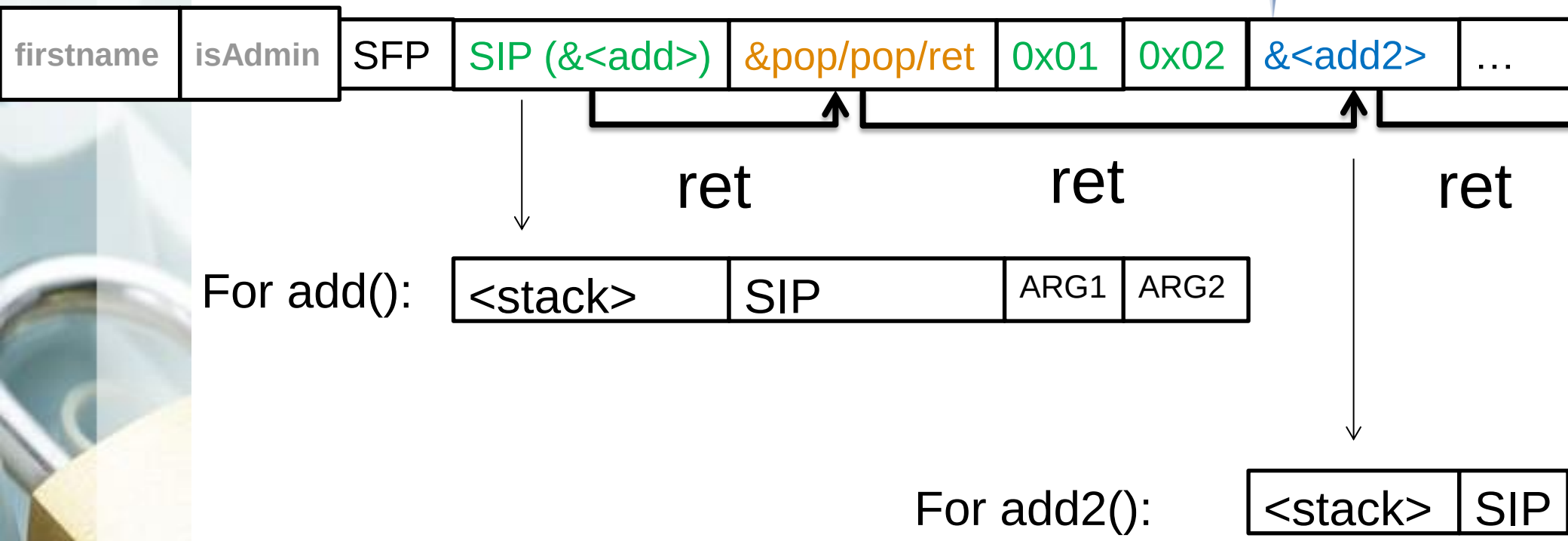
ROP By Example



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

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

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'
```

Practical ROP Exploiting

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

Practical ROP Exploiting

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)

Practical ROP Exploiting

ROP is very inefficient

Needs a lot of gadgets

Not suitable to implement complete shellcode in it

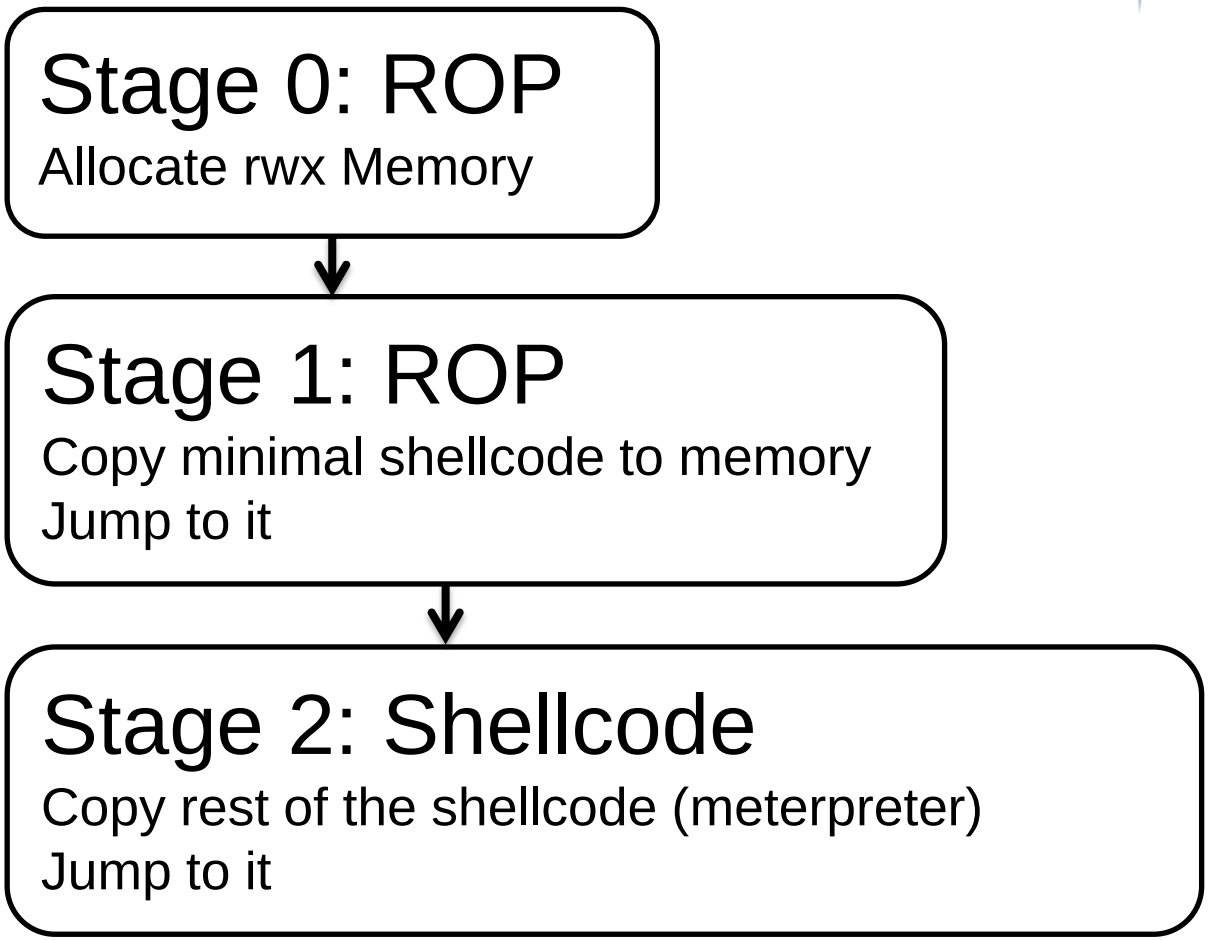
Hello: Multi Stage Shellcode

Stager: Change permission

- Set Stack executable
- Execute it (jmp)
- Profit

Stager: Allocator

- Allocate new RWX memory
- Copy rest of shellcode to newly allocated memory
- Execute it (jmp)
- Profit

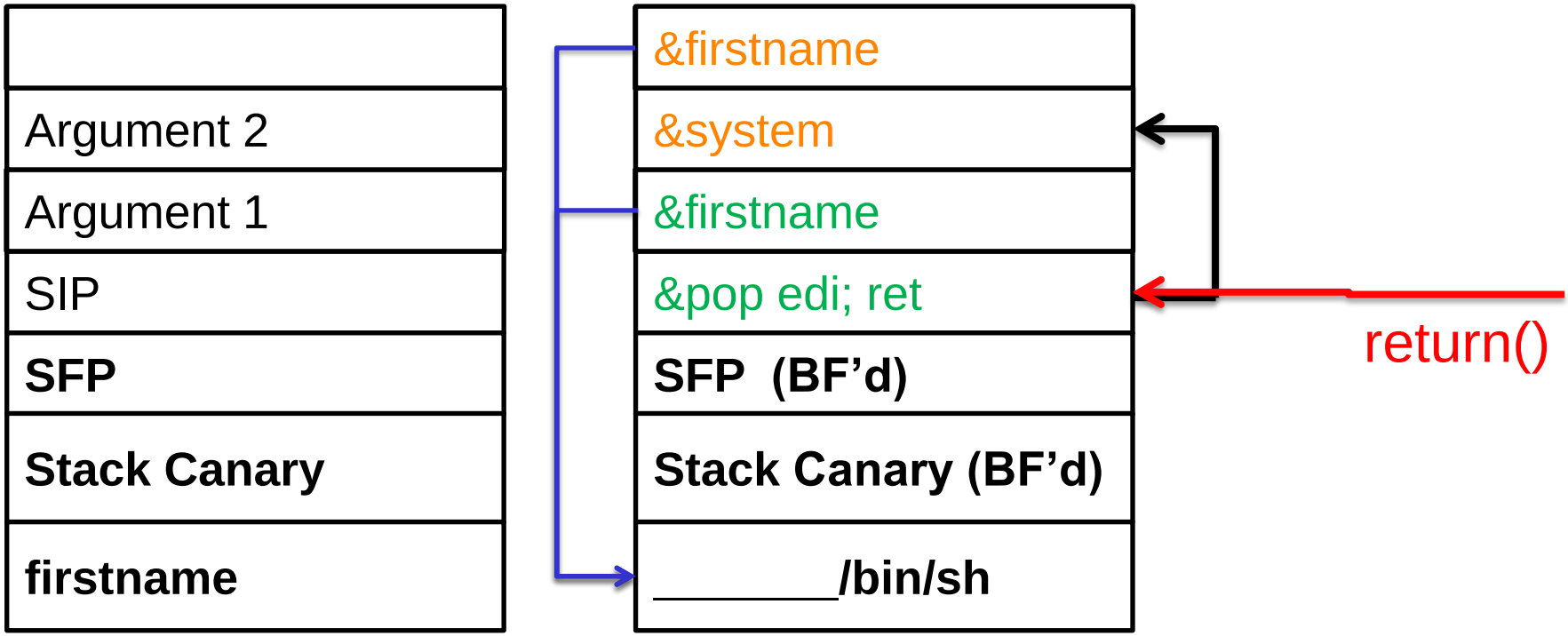

Stage 0: ROP
Allocate rwx Memory
Stage 1: ROP
Copy minimal shellcode to memory
Jump to it
Stage 2: Shellcode
Copy rest of the shellcode (meterpreter)
Jump to it

Practical ROP

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

Practical ROP



Insomnihack Teaser

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

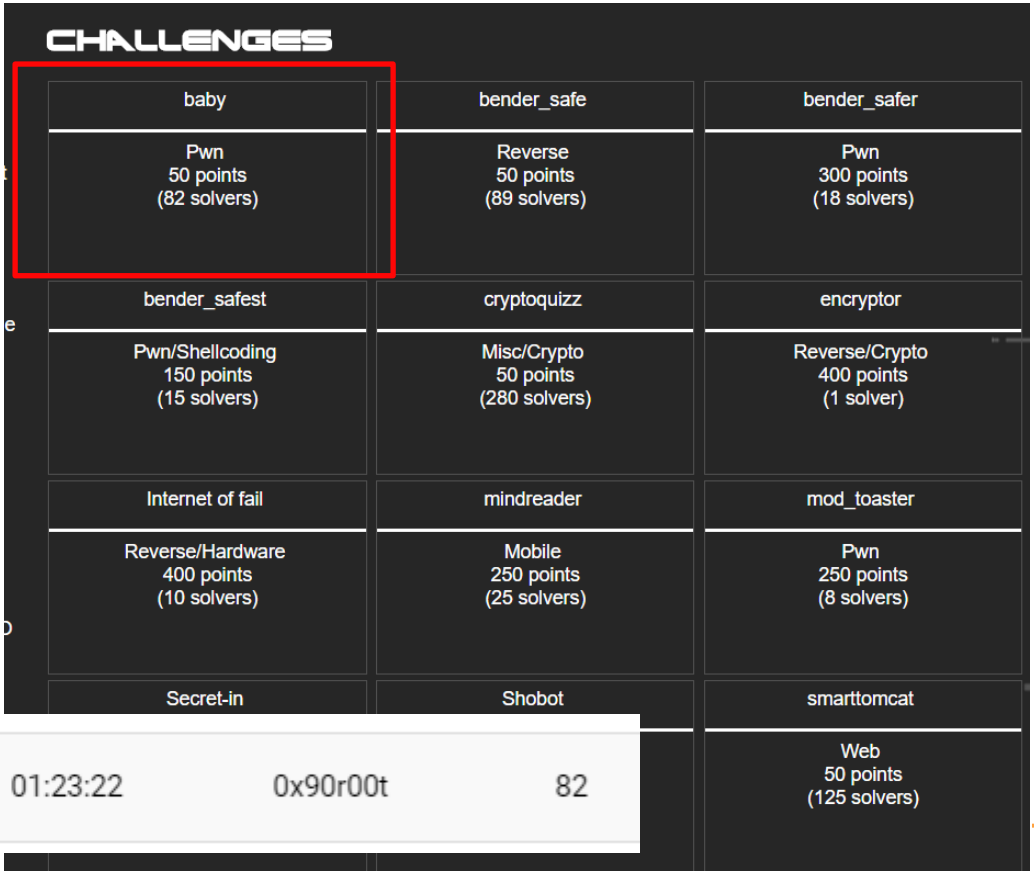
Insomnihack Teaser

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

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

ROP: Conclusion

Ret2libc / ret2got / ret2plt

- Is able to execute arbitrary library functions

ROP

- Turing complete
- Can execute arbitrary code

Targets:

- Prepare for full generic shellcode by using RWX memory