Questions & Answers 13/10/23

https://bit.ly/ODC_QA_1310

- Challenge To Review
- Topics to Review
    - Try to be specific
- Strange Stuff That you Do not Understand
- Complex Questions
    - Attach Example on GDocs

Offensive and Defensive CyberSecurity

–

Returned Oriented Programming

23-24

Some Credits to: Marco Bonelli, Alessandro Bertani

What's ROP?

We all know that overwriting a saved return address on the stack can be very interesting… we can make the program jump wherever we want, but that's it more or less.

- How can we call a function *passing arguments*?
- And what about calling *multiple* functions one after another?

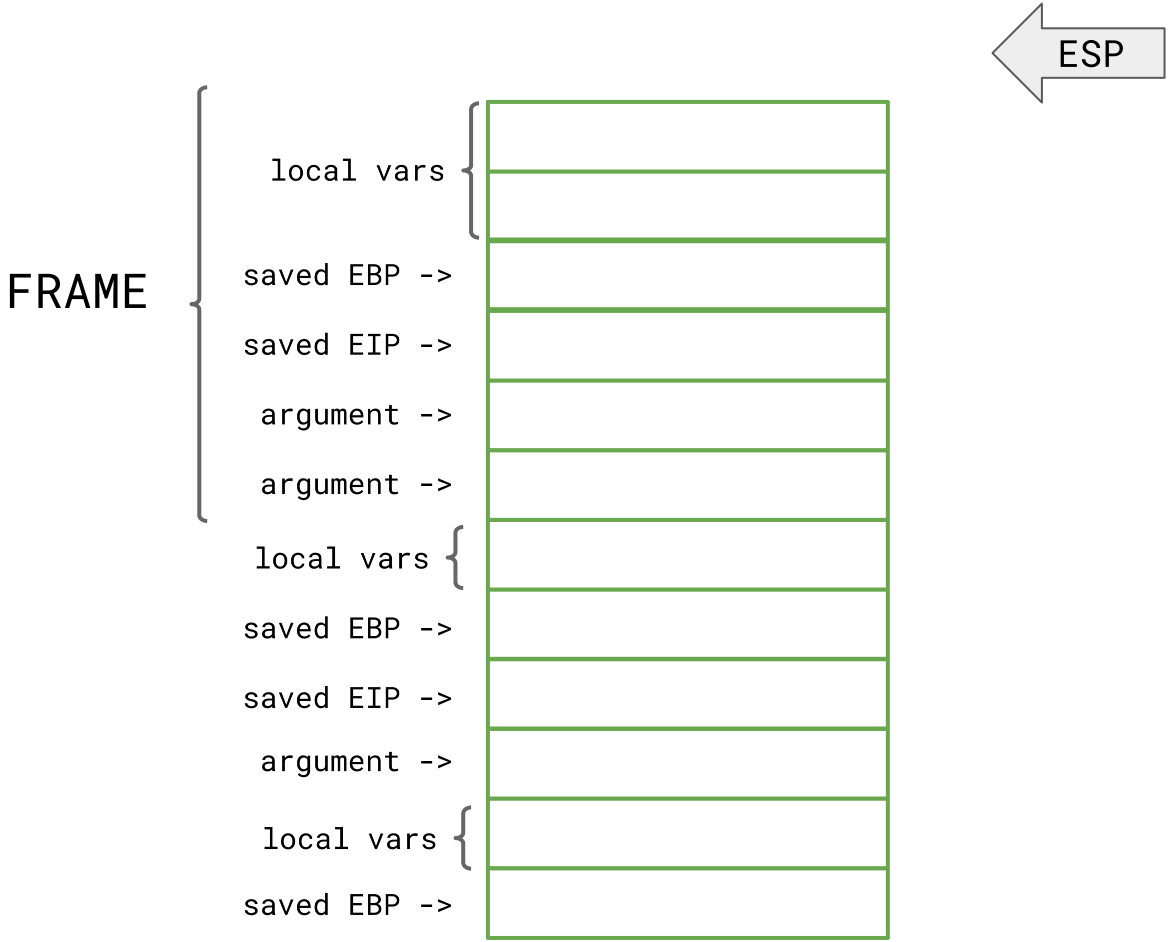
Here's where Return Oriented Programming comes in handy!

What's ROP?

```
0xffff00xx  <... locals of func_3>
0xffff0004  <saved $ebp>
0xffff0008  <saved return addr>
0xffff000c  <func_3's arg1>
0xffff0010  <func_3's arg2>
0xffff0014  <func_3's arg3>
0xffff00xx  <... locals of func_2>
0xffff001c  <saved $ebp>
0xffff0020  <saved return addr>
0xffff0024  <func_2's arg1>
0xffff0028  <func_2's arg2>
0xffff00xx  <... locals of func_1>
0xffff0030  <saved $ebp>
0xffff0034  <saved return addr>
0xffff0038  <func_1's arg1>
0xffff003c  <... locals of main>
```

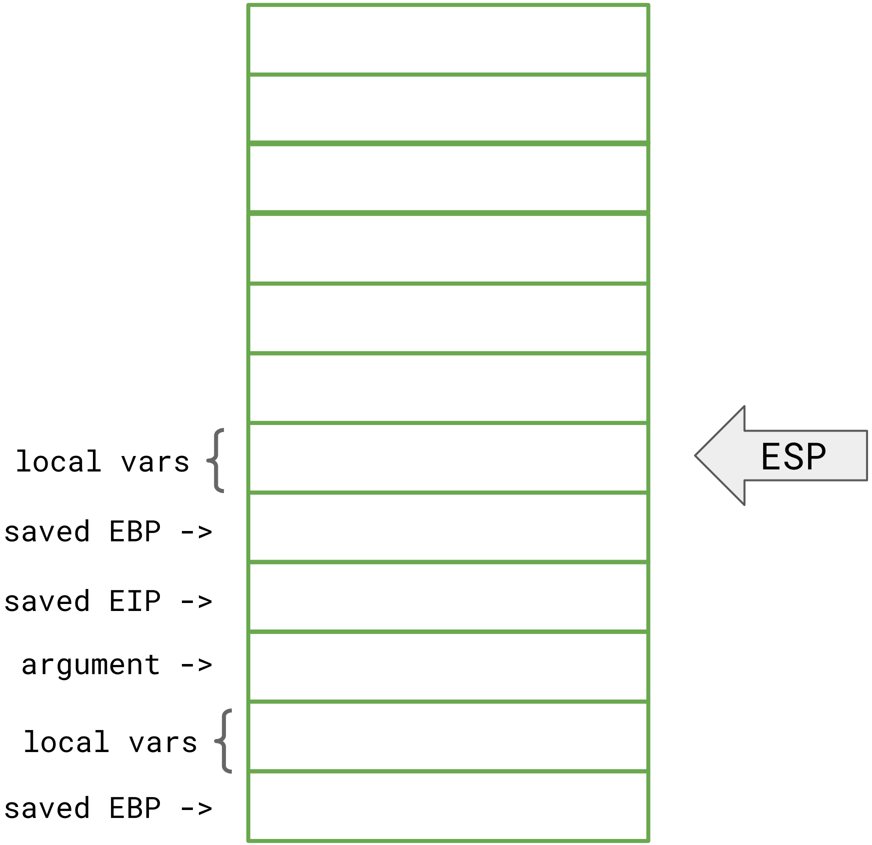
If we're careful enough about how we place stuff on the stack, we can *simulate fake stack frames* and chain the execution of multiple arbitrary functions with arbitrary arguments.

Call Frame


ESP
FRAME
local vars
saved EBP ->
saved EIP ->
argument ->
argument ->
local vars
saved EBP ->
saved EIP ->
argument ->
local vars
saved EBP ->

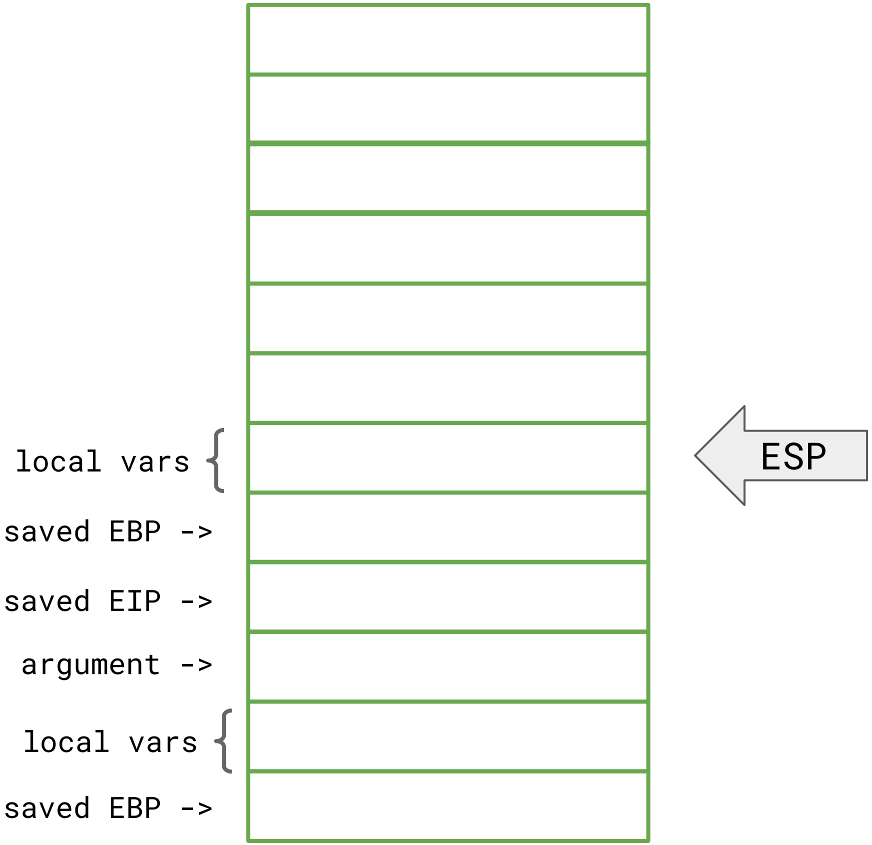
Caller

- PUSH EAX
- PUSH EBX
- CALL
- ADD ESP, 0x8



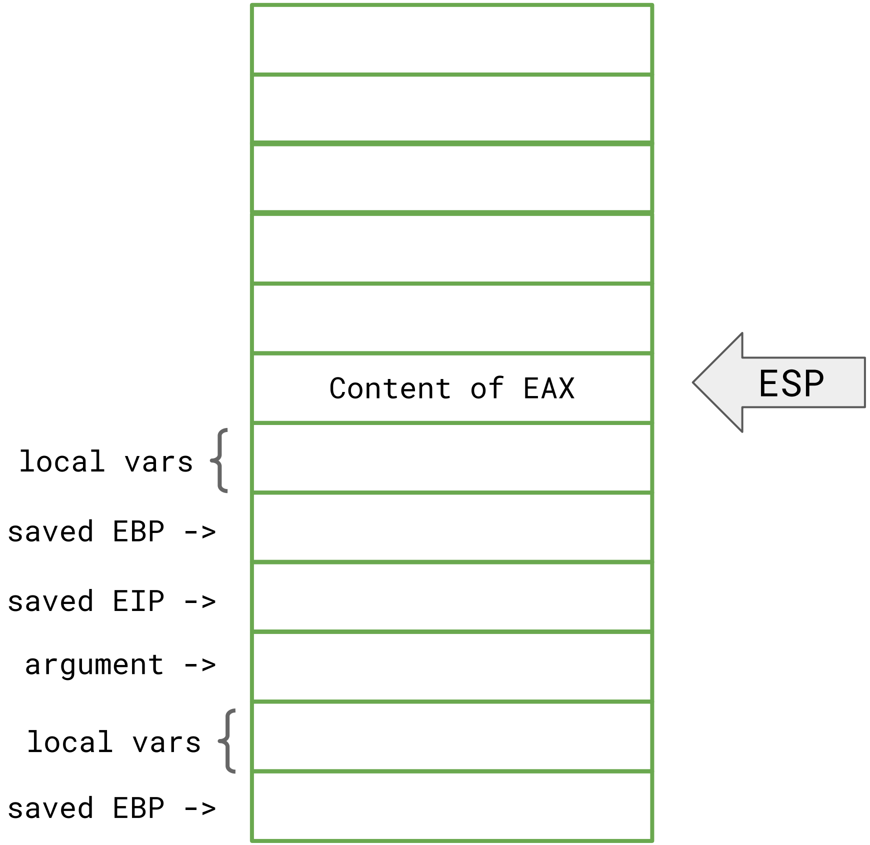
Caller

- **PUSH EAX**
- PUSH EBX
- CALL
- ADD ESP, 0x8



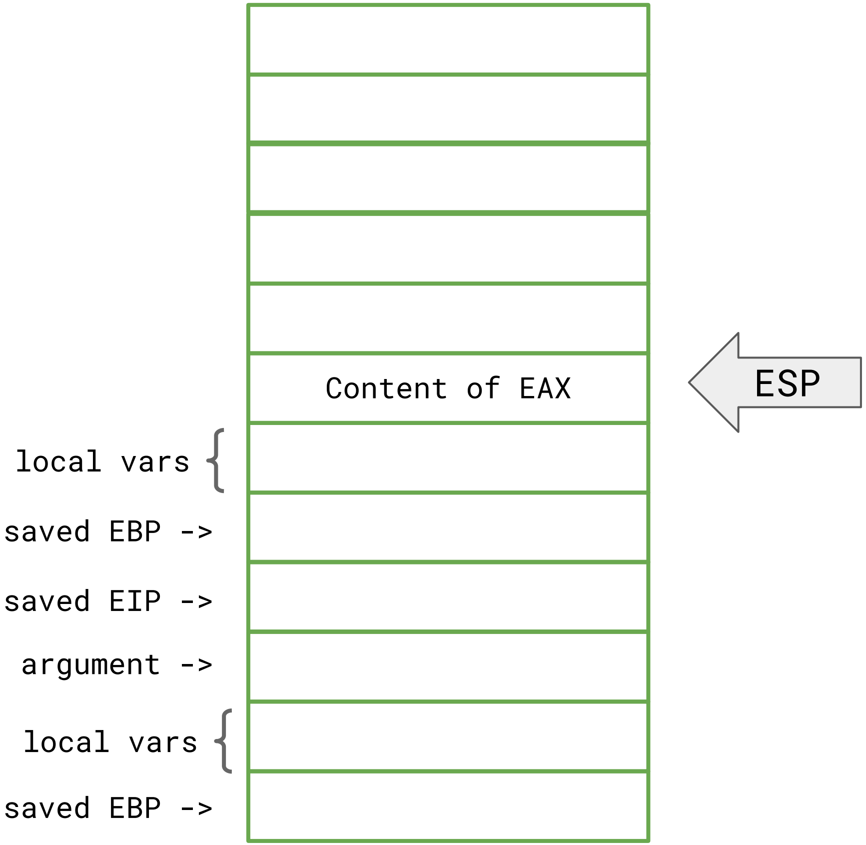
Caller

- **PUSH EAX**
  - SUB ESP, 0x4
  - MOV [ESP], EAX
- PUSH EBX
- CALL
- ADD ESP, 0x8



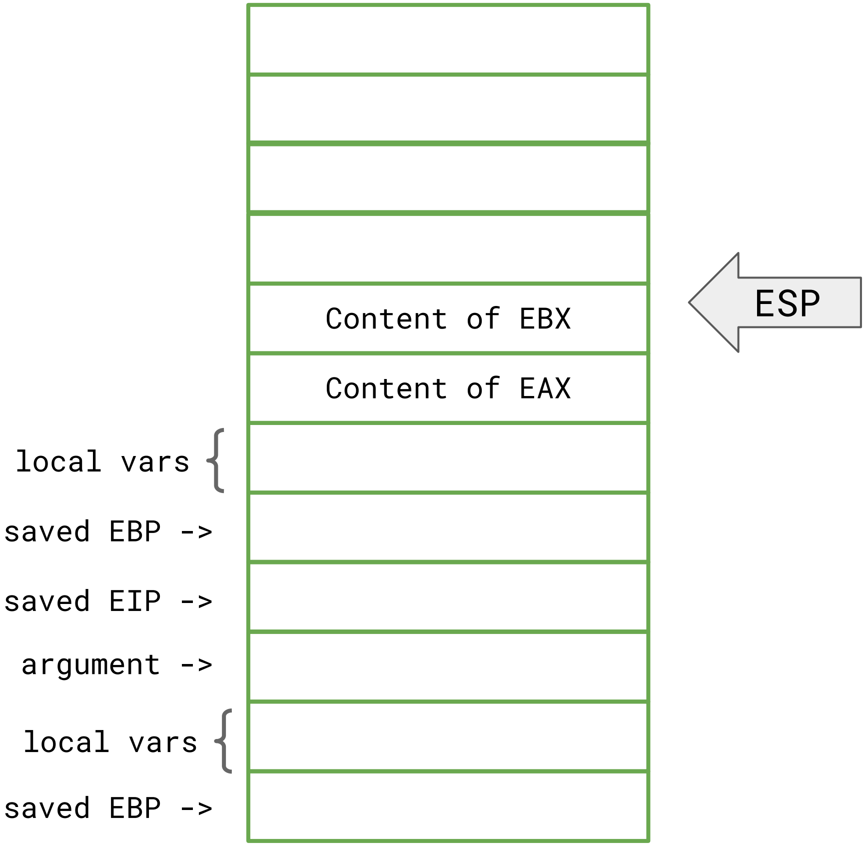
Caller

- PUSH EAX
- **PUSH EBX**
- CALL
- ADD ESP, 0x8



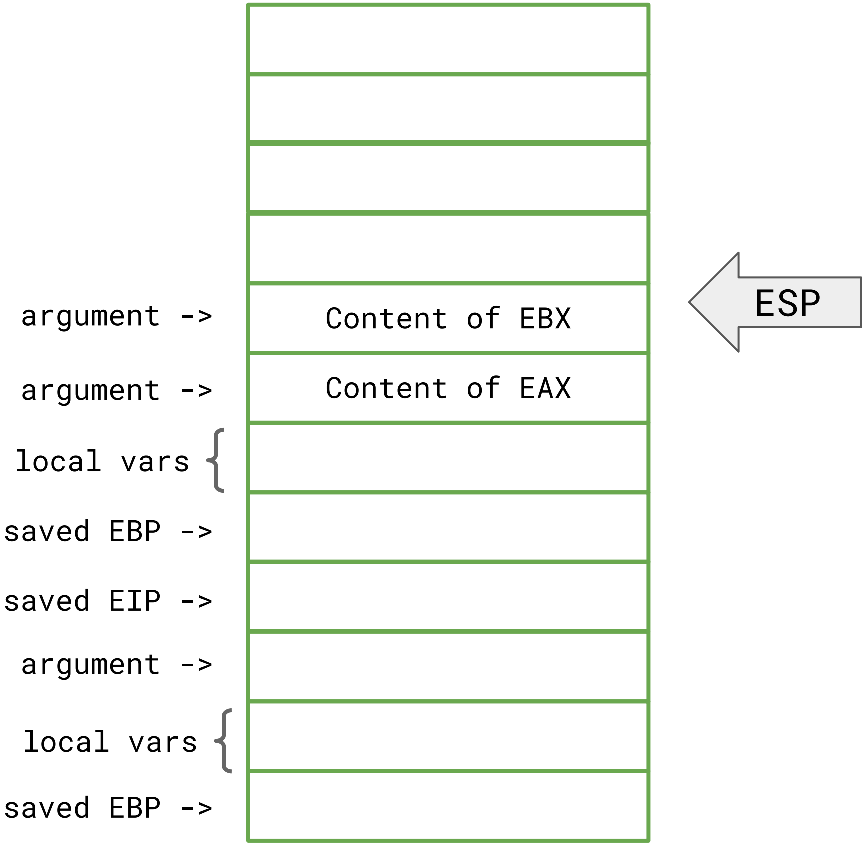
Caller

- PUSH EAX
- **PUSH EBX**
- CALL
- ADD ESP, 0x8



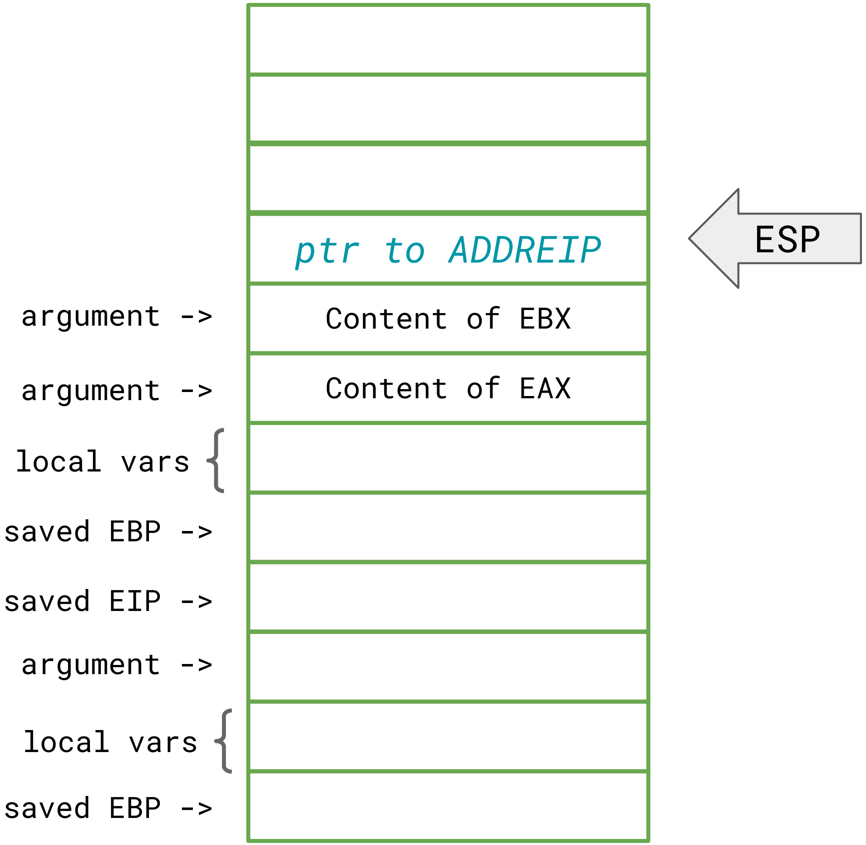
Caller

- PUSH EAX
- **PUSH EBX**
- CALL
- ADD ESP, 0x8



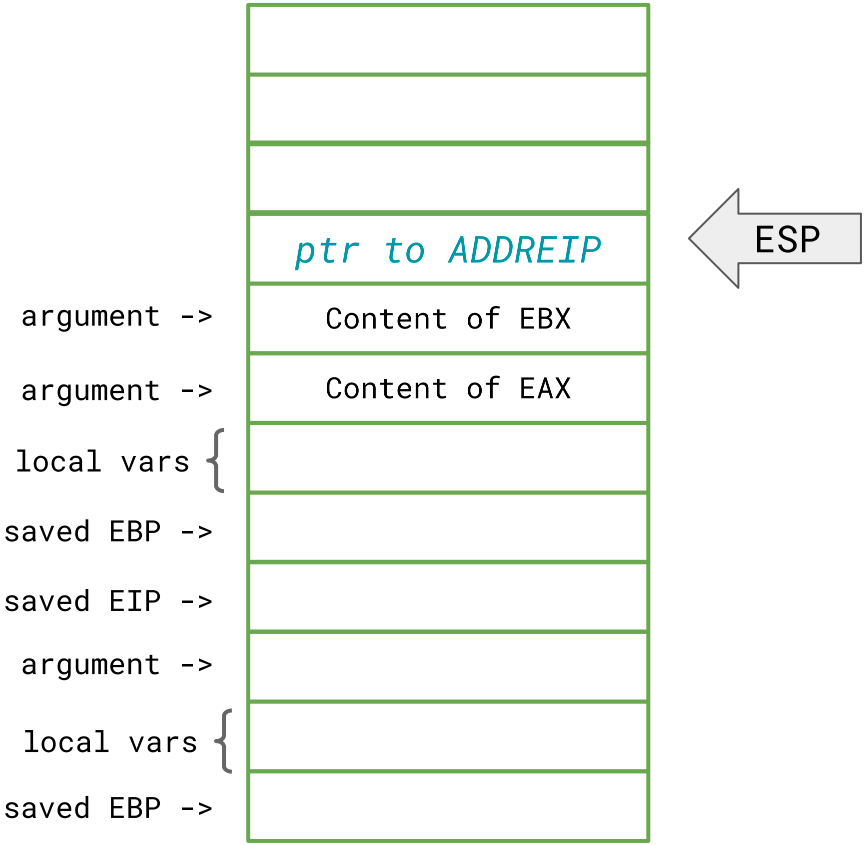
Caller

- PUSH EAX
- PUSH EBX
- **CALL FUN**
    - PUSH EIP
    - JUMP FUN
- ADD ESP, 0x8



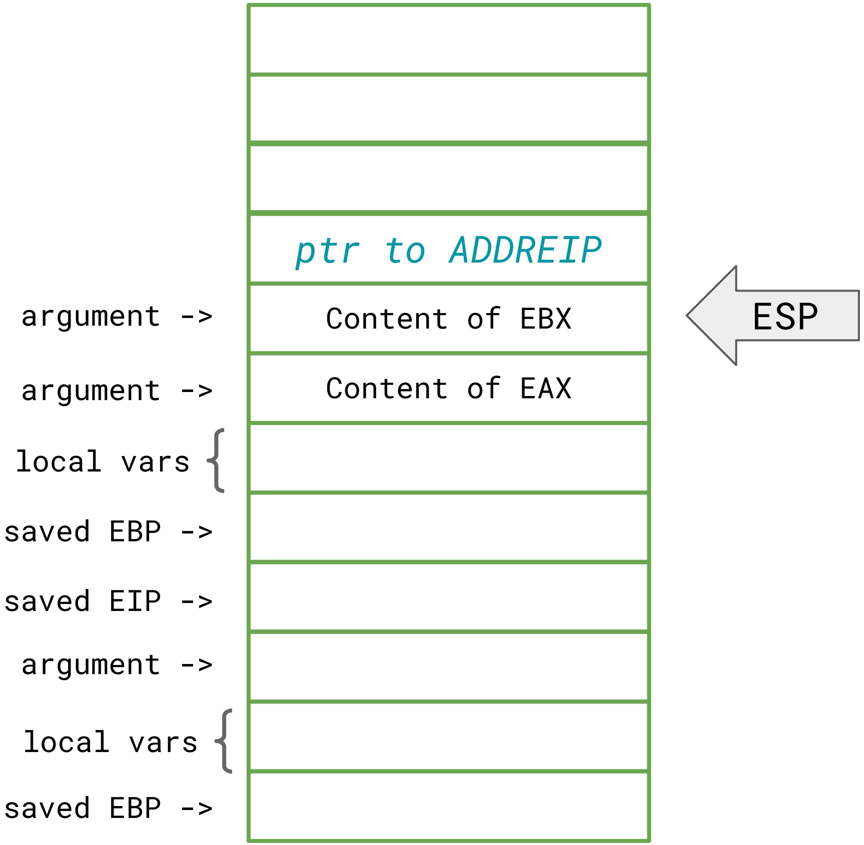
Fun

- RET
  - MOV RIP, [ESP]
  - ADD ESP, 0x4
- After RET the caller clean the stack



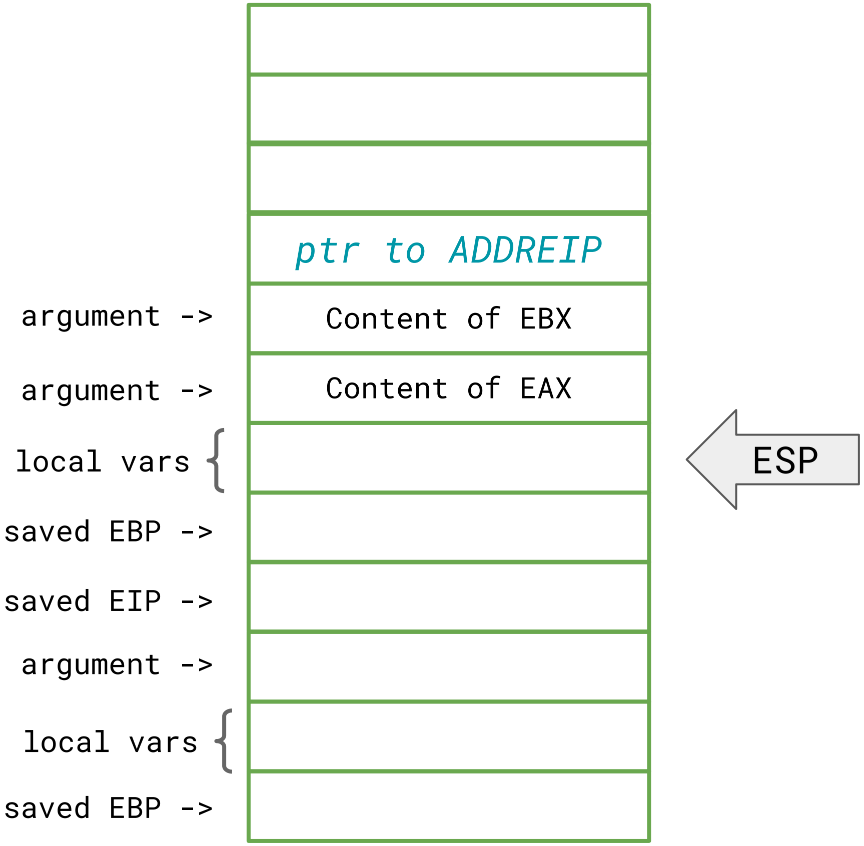
Caller

- PUSH EAX
- PUSH EBX
- CALL FUN
- **ADD ESP, 0x8**



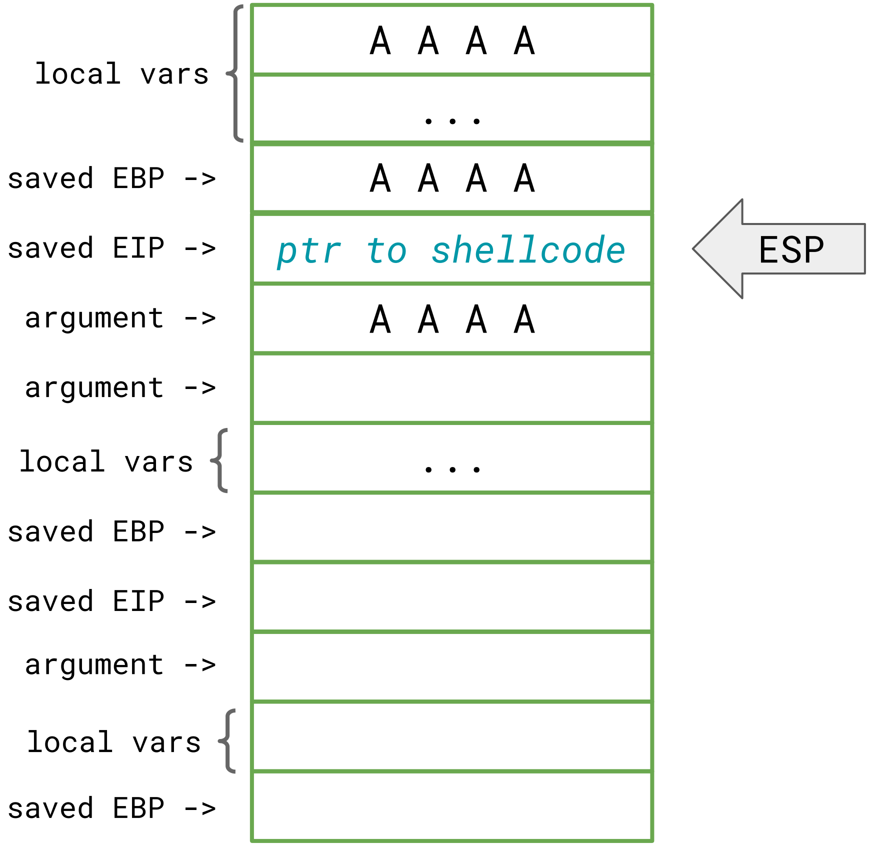
Caller

- PUSH EAX
- PUSH EBX
- CALL FUN
- ADD ESP, 0x8



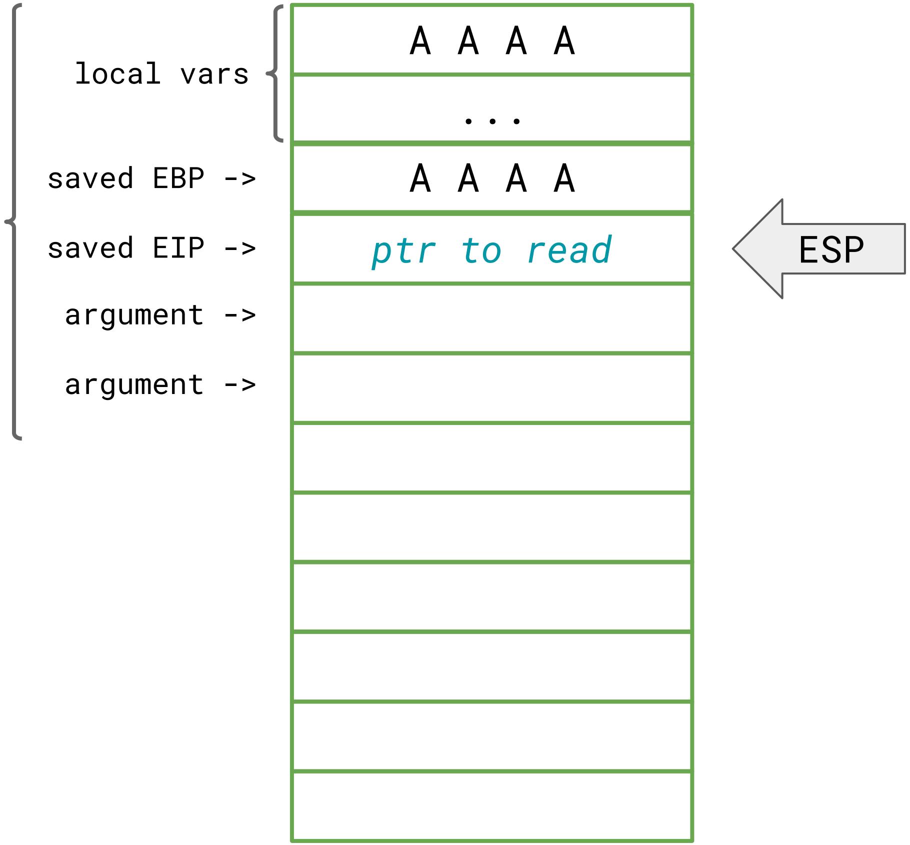
Buffer overflow

- Overwrite EIP and jump to a **shellcode**



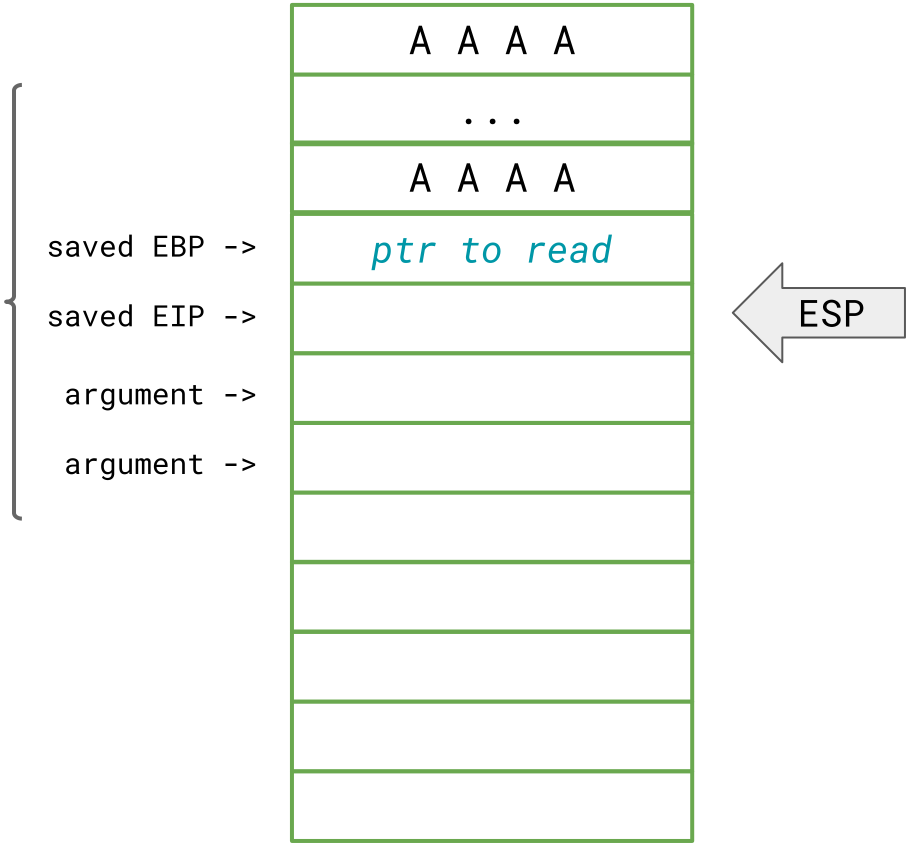
Return to Function (LibC)

- Overwrite EIP and jump to a **function**
- Setup the **arguments**



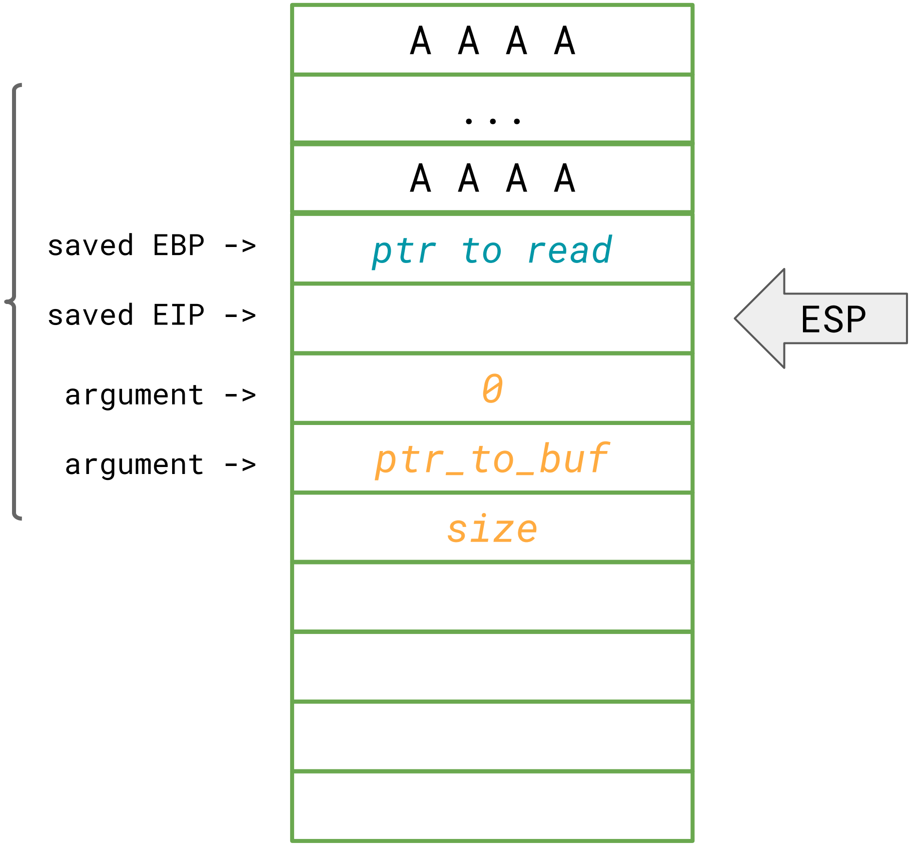
Return to Function (LibC)

- Overwrite EIP and jump to a **function**
- Setup the **arguments**



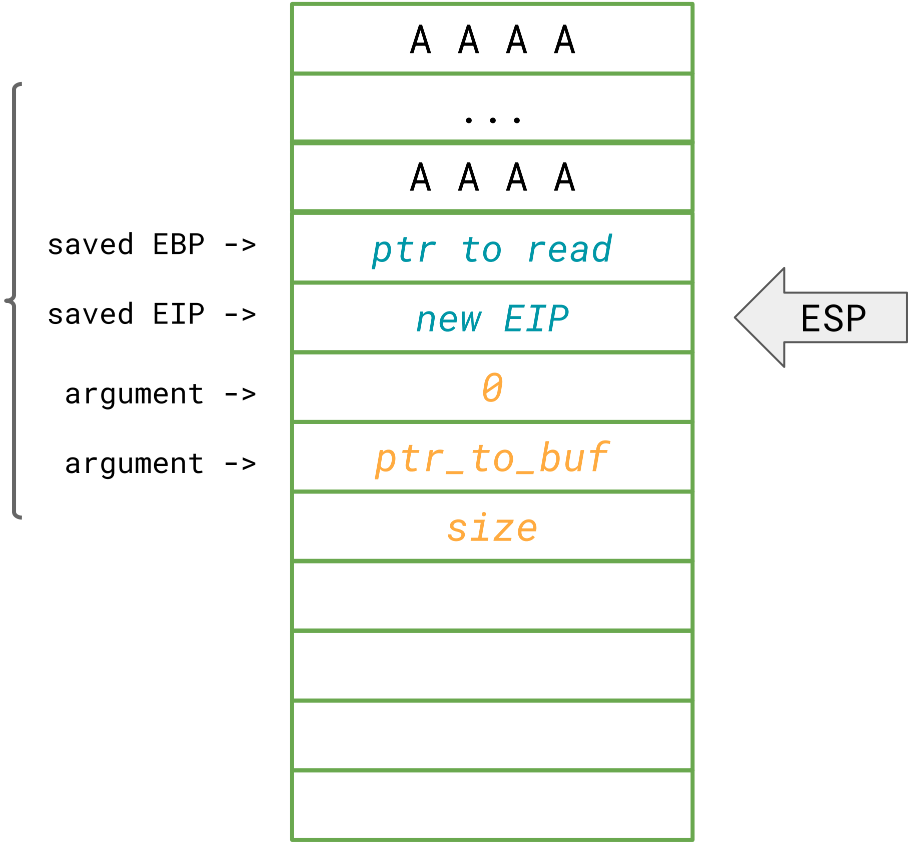
Return to Function (LibC)

- Overwrite EIP and jump to a **function**
- Setup the **arguments**



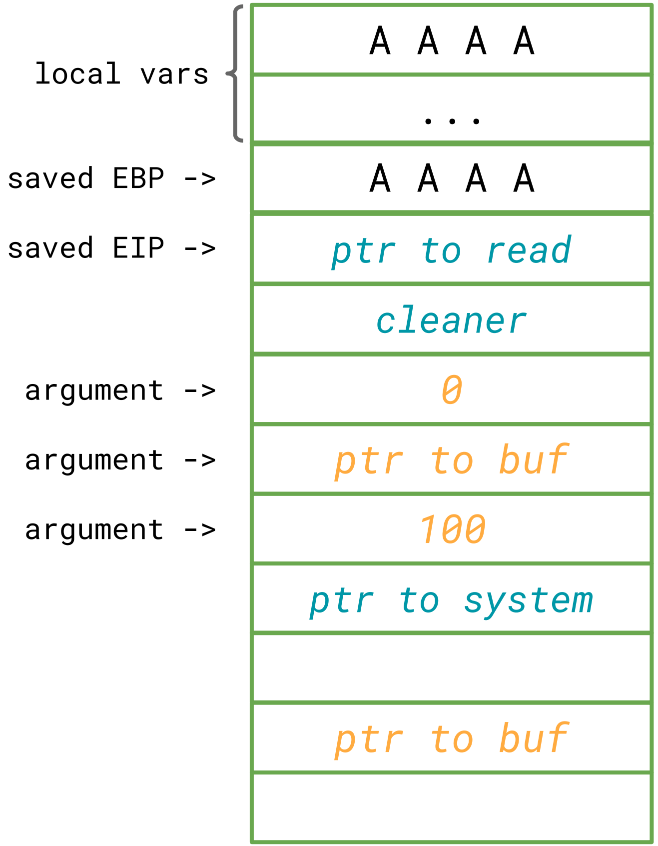
Chain

- Overwrite EIP and jump to a **function**
- Setup the **arguments**
- **Multiple times**



Chain Functions (aka ROP)

- Overwrite EIP and jump to a **function**
- Setup the **arguments**
- **Multiple times**



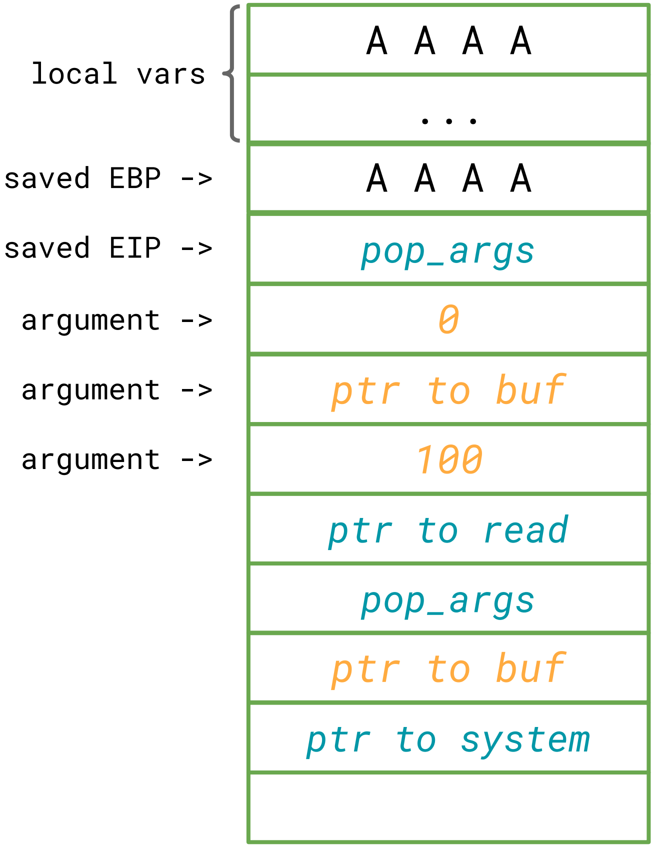
Gadget

- Instructions followed by a **ret**
- or a **int 0x80**
- or a **syscall**
- or a **jmp <reg>**

```
pop eax ; ret
pop ebp ; ret
pop edx ; pop eax ; ret
pop edx ; pop eax ; jmp rax
```

ROP 64-bit

- Arguments on regs
- Setup the **arguments**
- Cleaner Gadget become Setter



ROP gadgets

A gadget is a sequence of useful instructions followed by an instruction that gives control back to us (usually a **ret**).

If we want to call more than one function, we need something to clean the stack to continue the chain, the easiest way is a gadget like: pop regX; pop regY; ret

Gadgets can be found manually analyzing a binary or with automated programs.

ROP gadgets: useful tools

Useful tools to find ROP gadgets are:

- Ropper: github.com/sashs/Ropper
- ROPgadget: github.com/JonathanSalwan/ROPgadget
- rp++: github.com/0vercl0k/rp
- ropshell (cool online gadget library): ropshell.com
- one_gadget: github.com/david942j/one_gadget
- xrop: github.com/acama/xrop

ROP gadgets: useful tools

Example using ropper:

```
$ ropper -f myprogram
0x080487bd  adc al, 0x41; ret;
0x0804842e  adc al, 0x50; call edx;
0x08048466  adc byte ptr [eax - 0x3603a275], dl; ret;
0x080484f7  adc byte ptr [eax], bh; mov ebx, dword ptr [ebp - 4]; leave; ret;
0x08048531  add al, 0x24; ret;
0x08048529  add al, 0x59; pop ebp; lea esp, dword ptr [ecx - 4]; ret;
...
```

Save to a text file:

```
$ ropper --nocolor -f myprogram > gadgets.txt
```

Weird Machines

- Vulnerabilities and **abstractions** create weird machines
  - ROP, etc.
- Writing an Exploit is **Programming** a **Weird Machine**. ~Sergey Bratus

Magic gadgets

Every single libc binary must contain some code to execute /bin/sh somehow, since libc provides the `system(cmd)` function, which basically does `execl("/bin/sh", ...)`.

A "magic gadget", also called "one gadget", is a gadget that **can spawn a shell alone if the program jumps to it!**

There usually are several magic gadgets laying around in the libc binary, each requiring different constraints to work.

Magic gadgets

The one_gadget tool is a very cool Ruby program which can automatically find magic gadgets and their constraints:

```
$ one_gadget /lib/x86_64-linux-gnu/libc.so.6

0x3f306 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x3f35a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xd6b9f execve("/bin/sh", rsp+0x60, environ)
constraints:
  [rsp+0x60] == NULL
```

/lib/x86_64-linux-gnu/libc.so.6:

```
3f35a:  mov   rax,QWORD PTR [rip+0x359b57]
3f361:  lea   rdi,[rip+0x1228b1]
3f368:  lea   rsi,[rsp+0x30]
3f36d:  mov   DWORD PTR [rip+0x35c109],0x0
3f377:  mov   DWORD PTR [rip+0x35c103],0x0
3f381:  mov   rdx,QWORD PTR [rax]
3f384:  call  b8640 <execve@@GLIBC_2.2.5>
```

ROP chain: 32bit vs 64bit

In x86 32bit arguments are almost always passed on the stack (as per the **cdecl** calling convention), but in x86 64bit arguments are usually passed in registers (as per the **System V** calling convention).

If we want to build a 64bit ROP chain we need to use gadgets to *pop the arguments from our chain to the needed registers*. Even if we're only calling one function!

ROP chain: 32bit vs 64bit

32bit **cdecl** convention: arguments on the stack

64bit **System V** convention: arguments in `RDI, RSI, RDX, RCX, R8, R9, XMM0…7`

```
<...padding...>
<func_1>
<gadget_1>
<func_1's arg1>
<func_1's arg2>
<func_2>
<gadget_2>
<func_2's arg1>
<func_3>
<gadget_3>
<func_3's arg1>
<...>
```

```
pop <some_reg>
pop <some_reg>
ret
```

```
add esp, 0x4
ret
```

```
pop <some_reg>
ret
```

```
<...padding...>
<gadget_1>
<func_1's arg1>
<func_1's arg2>
<func_1>
<gadget_2>
<func_2's arg1>
<func_2>
<gadget_3>
<func_3's arg1>
<func_3>
<...>
```

```
pop rdi
pop rsi
ret
```

```
pop rdi
sub eax, 0x8
ret
```

```
pop rdi
ret
```

ROP chain: not only calling functions

Sometimes you cannot call functions, but who needs to call library functions when you've got the right gadgets?

```
pop rbp
ret
```

```
mov DWORD PTR [rsi], ebx
sub rsp, 0x20
ret
```

```
xchg rsi, rdi
ret
```

```
add al, 0x48
add edx, 1
syscall
```

```
pop r15
pop r10
pop r13
ret
```

```
pop rbp
mov edi, 0x61e600
jmp rax
```

```
mov dword ptr [rdi + 0x10], ecx
xor ch, ch
mov byte ptr [rdi + 0x12], ch
ret
```

```
int 0x80
```

More than ROP...

If you're interested, you might want to also take a look at SROP: Sigreturn Oriented Programming.

This technique takes advantage of the sigreturn syscall to take control of the registers (and thus the execution) by using gadgets which are usually always in memory at runtime.

SROP is generally "simpler" than ROP and often only needs one gadget (to execute the sigreturn syscall).

16-byte stack alignment

The x86-64 System V ABI guarantees **16-byte stack alignment** before a call.

(Ergo the address inside RSP should end with 0)

system@libc take advantage of that for 16-byte aligned loads/stores

Load another Library (libc-2.xx.so)

- **env LD_PRELOAD**
  - `LD_PRELOAD=./libc-2.23.so ./binary`
- **ld.so**
  - `./ld-2.23.so --library-path ./lib ./binary`
  - lib contains libc.so.6
- **patchelf** (https://github.com/NixOS/patchelf)
  - `patchelf --set-interpreter ./ld-2.23.so --replace-needed libc.so.6 ./libc-2.23.so ./binary`
- **YOLO** (Do not use this!)
  - Replace system library
- **Docker/Virtual Machine**

Weird Machines

- Vulnerabilities and **abstractions** create weird machines
  - ROP, etc.
- Writing an Exploit is **Programming** a **Weird Machine**. ~Sergey Bratus

ROP chain: 32bit vs 64bit

32bit **cdecl** convention: arguments on the stack

64bit **System V** convention: arguments in `RDI, RSI, RDX, RCX, R8, R9, XMM0…7`

```
<...padding...>
<func_1>
<gadget_1>
<func_1's arg1>
<func_1's arg2>
<func_2>
<gadget_2>
<func_2's arg1>
<func_3>
<gadget_3>
<func_3's arg1>
<...>
```

```
pop <some_reg>
pop <some_reg>
ret
```

```
add esp, 0x4
ret
```

```
pop <some_reg>
ret
```

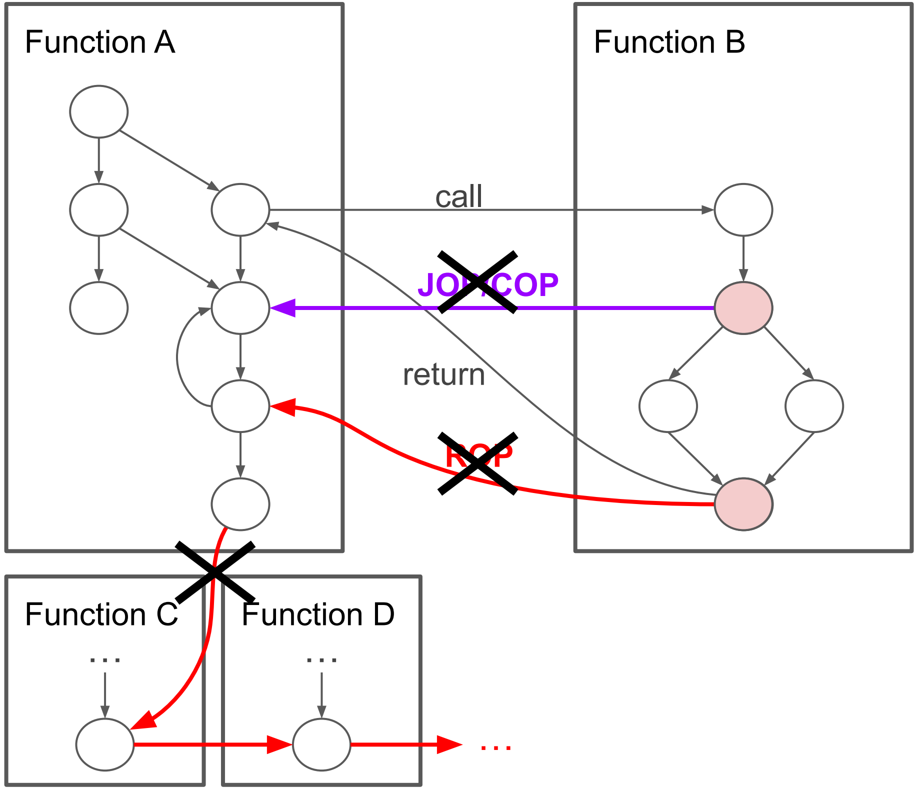
```
<...padding...>
<gadget_1>
<func_1's arg1>
<func_1's arg2>
<func_1>
<gadget_2>
<func_2's arg1>
<func_2>
<gadget_3>
<func_3's arg1>
<func_3>
<...>
```

```
pop rdi
pop rsi
ret
```

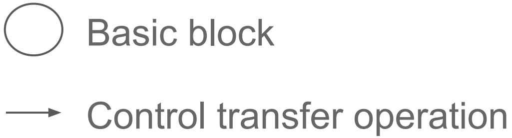
```
pop rdi
sub eax, 0x8
ret
```

```
pop rdi
ret
```

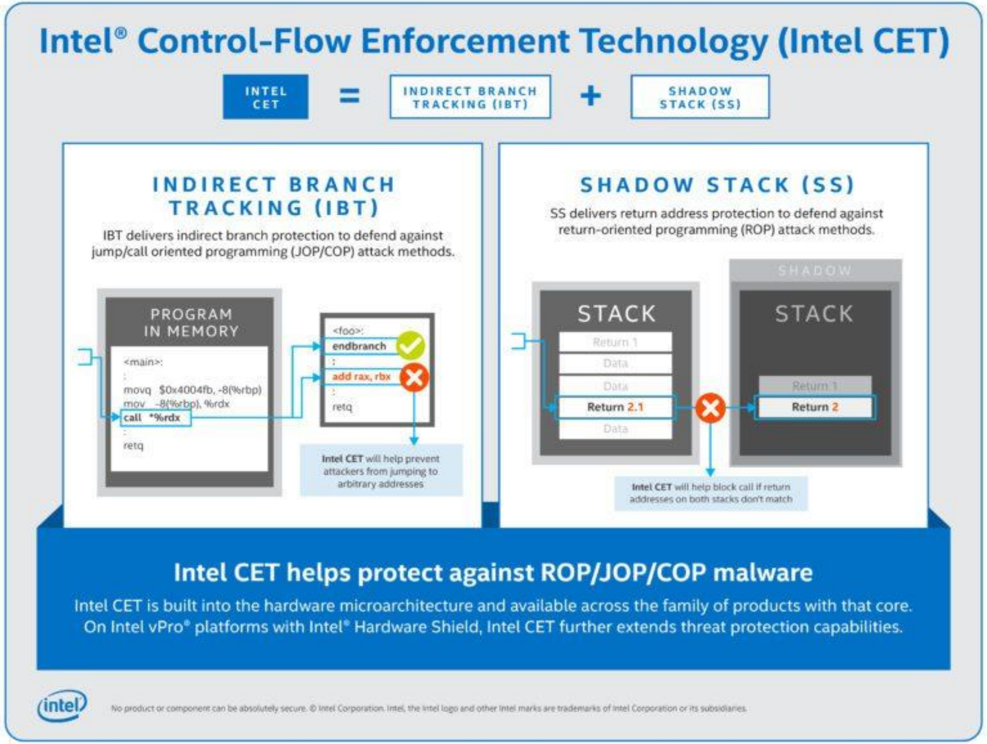
Control-Flow Integrity (CFI)



Strong CFI defenses such as Intel's Control-flow Enforcement Technology (CET) defeat ROP and JOP/COP style exploits.

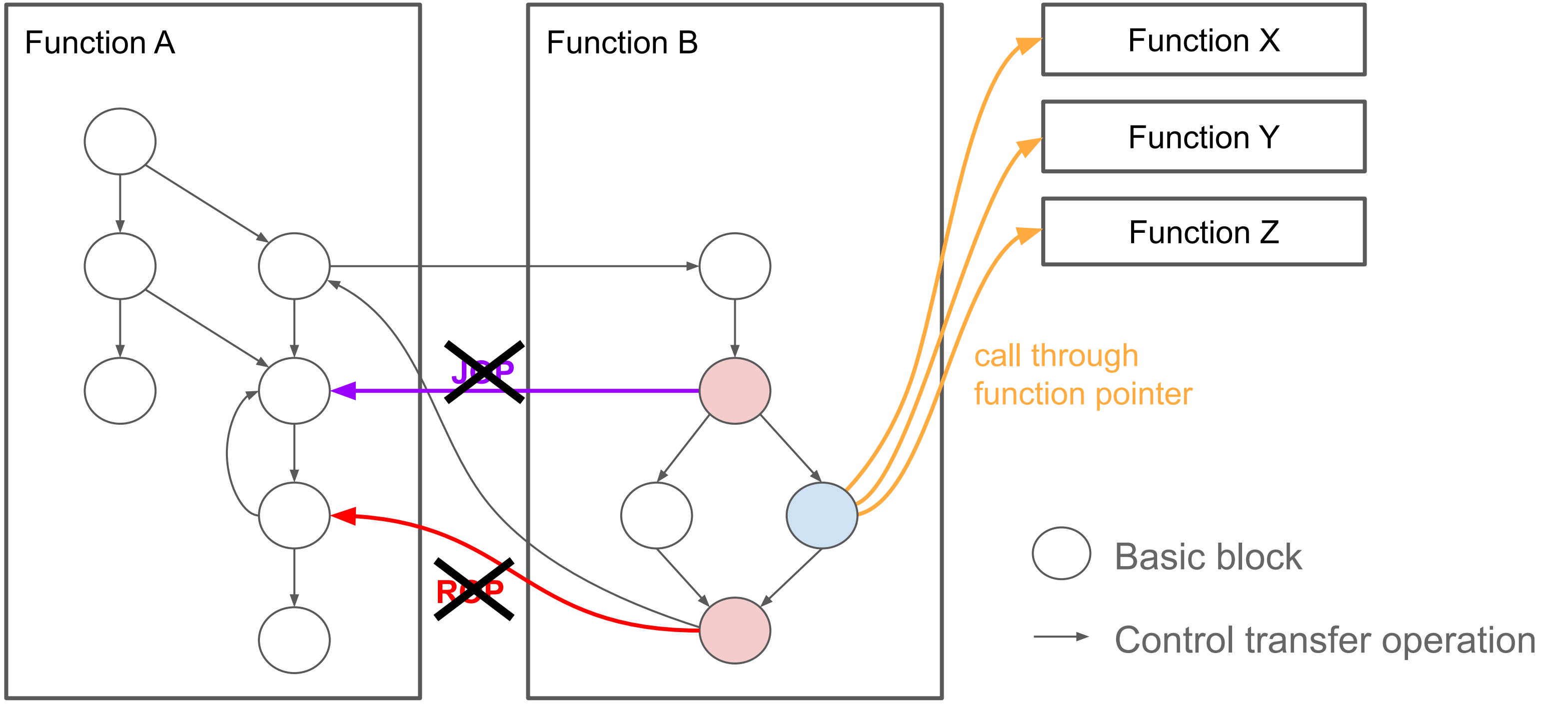


Intel CET



https://www.intel.com/content/dam/develop/external/us/en/documents/catc17-introduction-intel-cet-844137.pdf

Function Pointer Hijacking



Problem Statement

Vulnerable User Program

```
/* Read a and b ... */
library_func(a, b);
```

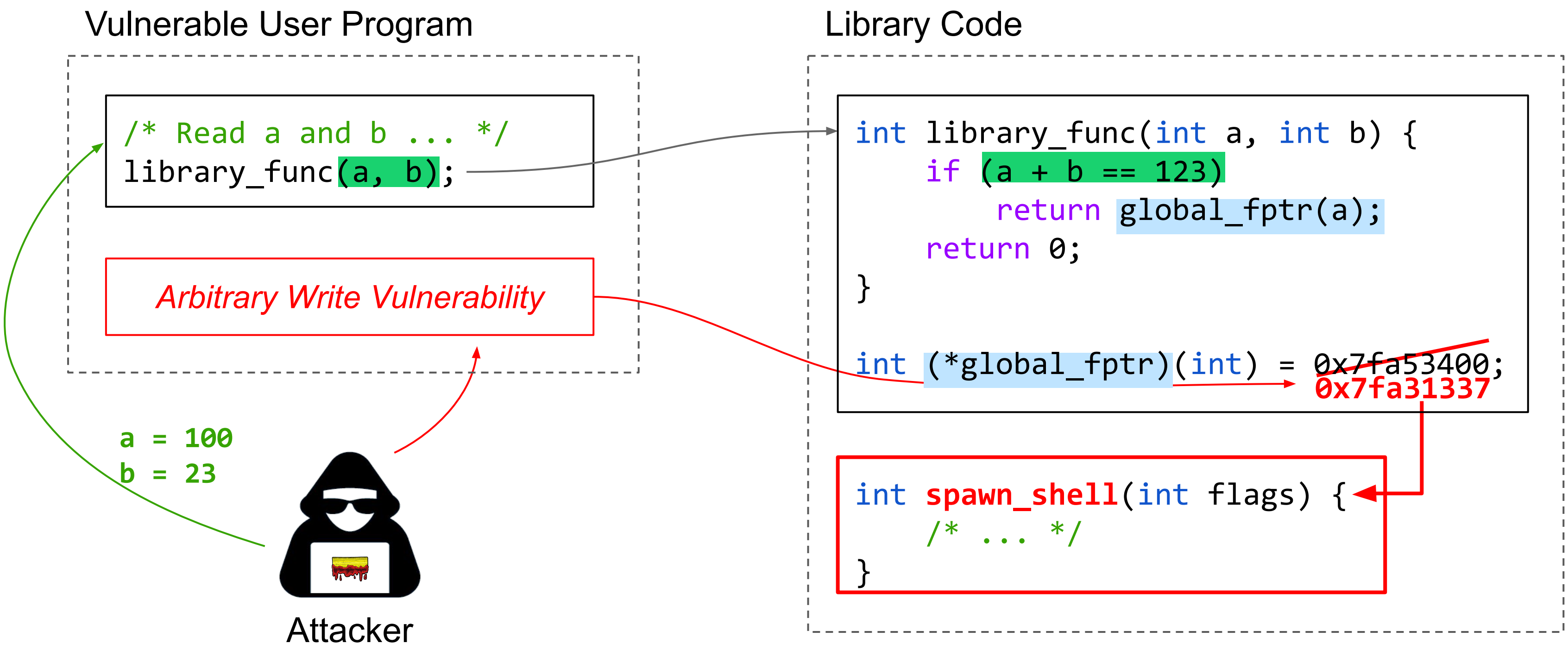
Library Code

```
int library_func(int a, int b) {
    if (a + b == 123)
        return global_fptr(a);
    return 0;
}

int (*global_fptr)(int) = 0x7fa53400;
```

```
int harmless_func(int x) {
    /* ... */
}
```

Problem Statement



Experimental Validation: Static Analysis

Library	Estimated lines of source code	Unique global function pointers	Reachable function pointers	Unique call sites	**Exported functions able to reach any call site**	Unique paths to call sites
libgnutls v3.6.16	422 804	15	14	1 338	827	29 817
libasound v1.2.4	94 288	3	2	383	243	7 739
libxml2 v2.9.10	353 481	8	6	2 125	225	254 096
libfuse v3.11	21 568	1	1	110	110	110
libcurl v7.84	152 921	5	5	271	48	11 238
libnss v2.31	10 568	21	18	34	15	74
libpcre v8.39	107 530	3	3	13	12	36
libbsd v0.11.3	11 316	8	8	8	8	8
Total	1 174 476	64	57	4 282	1 488	303 118