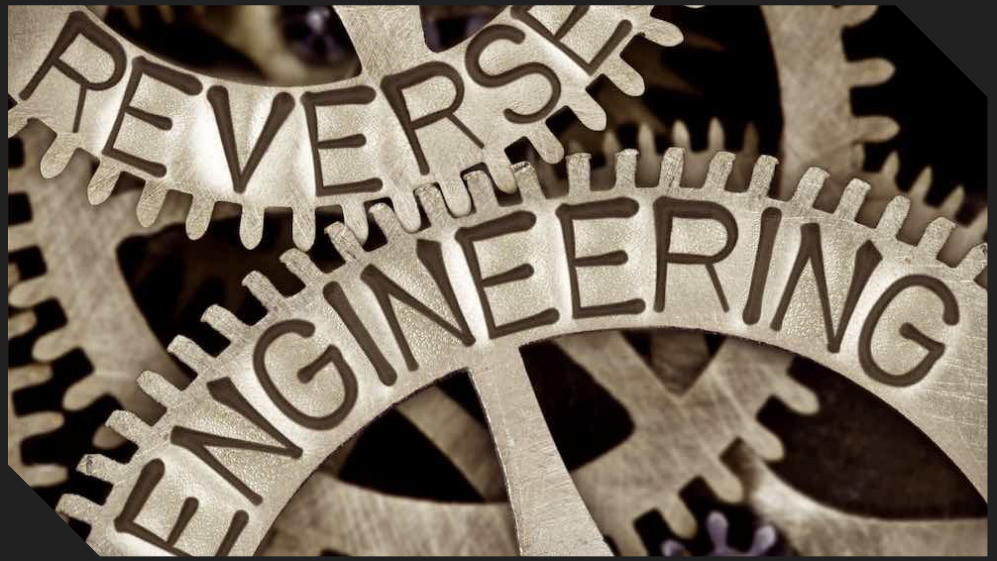
Day 5 : Reverse Engineering



~ [CCSC] CIT Cyber Security Cell ~
OUSSAMA RAHALI
OMAR AOUAJ

CLUB INFORMATIQUE & TÉLÉCOM

cat README.md

Presentation outline

Warning !

We are just beginners in reverse engineering and binary exploitation

So ... !! ;)

`Whatis Reverse Engineering`

★ **Official Def:**
“ … the process by which a man-made object is deconstructed to reveal its designs, architecture, or to extract knowledge from the **object” [1]**

[1] https://en.wikipedia.org/wiki/Reverse_engineering

`Whatis Reverse Engineering`

- ★ Layperson's Terms
  “ Doing whatever you need to do to figure out how something works, to whatever level of understanding you need, with or without documentation.”

> grep “Our Goal” here

Figuring out how a compiled program works

NB :

- Skipping a LOT of things [only basics]

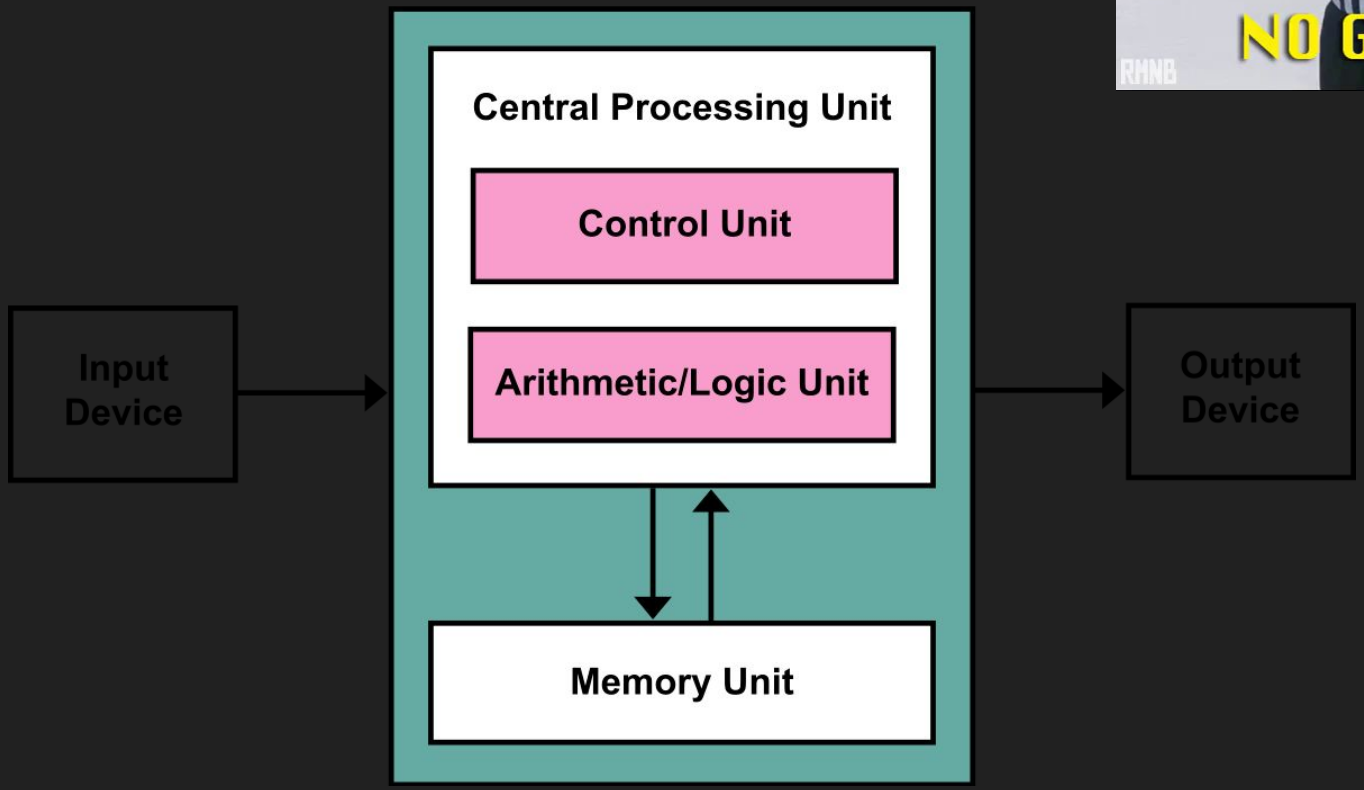
`Whatis in a computer ?`

- ★ Memory [RAM]
- ★ Cache
- ★ Registers :
  - General purposes : [for x86]
    rax, rbx, rcx, rdx, rsi, rdi,
    r8,..,15
  - Segment:
    cs, ss, es, ds, fs, gs
  -

`Whatis in a computer ?`

- ★ […]
  - Flag register RFLAGS/EFLAGS/FLAGS :
    - Carry [CF]
    - Parity [PF]
    - Adjust [AF]
    - Zero [ZF]
    - Sign [SF]
    - Trap [TF]

`whois Von Neumann` ?



`whois Von Neumann` ?

- ★ CPU [Central Process Unit] :
  device in charge of executing the *machine code* of a program

- ★ Machine Code : machine language
  set of *instructions* that the CPU processes

- ★ Each instruction is a primitive command that executes a specific operation such as move data, changes the execution flow of the program, perform arithmetic or logic operations and more ...

Language Types

Compiled (C++, C, Java, Go, …)

Interpreted (Ruby, Python, PHP …)

Compiled Vs Non-compiled

- Non-compiled programs are easy peasy to reverse by anyone with some language basic/skill.
- Compiled require different skills

Executable Formats

- ELF (Executable and linkable format) [Linux / Unix]
- Mach-O [OSX / MacOS]
- PE (Portable Executable) (Windows)
- DOS (ooooooold Windows executable)

`Whatis Assembly`

CPU instructions are represented in hexadecimal format. Due to its complexity, it is impossible for humans to utilize it in its natural format. Therefore, the same machine code gets translated into a more readable language; this is called the assembly language (ASM).

`Whatis Assembly`

Demo0

```c
#include <stdio.h>

void main(void)
{
    puts("Hello CIT!");
}
```

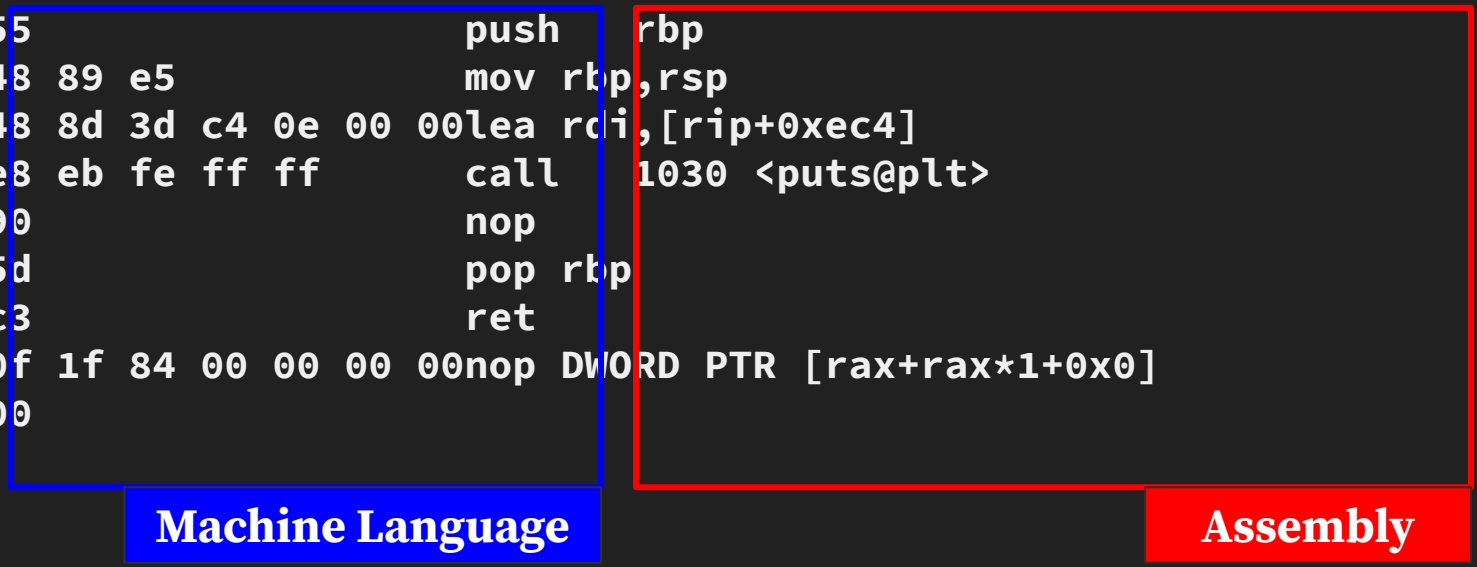
Compiling

```
$ gcc demo0.c -o demo0
$ objdump -M intel -D demo0 | less
```

`Whatis Assembly`

Demo0

```
0000000000001135 <main>:
    1135:   55                      push   rbp
    1136:   48 89 e5                mov rbp,rsp
    1139:   48 8d 3d c4 0e 00 00    lea rdi,[rip+0xec4]
    1140:   e8 eb fe ff ff          call   1030 <puts@plt>
    1145:   90                      nop
    1146:   5d                      pop rbp
    1147:   c3                      ret
    1148:   0f 1f 84 00 00 00 00    nop DWORD PTR [rax+rax*1+0x0]
    114f:   00
```



Registers

The number of bits, 32 or 64, refers to the width of the CPU registers.

- ➢ Each CPU has its fixed set of registers that are accessed when required.
- ➢ Registers ~~ temporary variables used by CPU to store data
- ➢ Some of them have a specific function, while others are used for general data storage.

Registers

- Naming convention of x86 architecture registers:

X86 Naming Convention	Name	Purpose
EAX	Accumulator	Used in arithmetic operation
ECX	Counter	Used in shift/rotate instruction and loops
EDX	Data	Used in arithmetic operation and I/O
EBX	Base	Used as a pointer to data
ESP	Stack Pointer	Pointer to the top of the stack
EBP	Base Pointer	Pointer to the base of the stack (aka Stack Base Pointer, or Frame pointer)
ESI	Source Index	Used as a pointer to a source in stream operation
EDI	Destination	Used as a pointer to a destination in stream operation

Registers

- Old 8-bit CPU had 16-but register divided into two :
  - A low byte : L at the end of the name.
  - A high byte : H at the end of the name.
- 16-bit CPU:
  - Combines the L and H and replaces it with an X
- 32-bit representation :
  - prefixed with an E (Extended)
- 64-bit rep:
  - the E is replaced with the R

Registers

Register	Accumulator	Counter	Data	Base
64-bit	RAX	RCX	RDX	RBX
32-bit	EAX	ECX	EDX	EBX
16-bit	AX	CX	DX	BX
8-bit	AH AL	CH CL	DH DL	BH BL

Register	Stack Pointer	Base Pointer	Source	Destination
64-bit	RSP	RBP	RSI	RDI
32-bit	ESP	EBP	ESI	EDI
16-bit	SP	BP	SI	DI
8-bit	SPL	BPL	SIL	DIL

Registers

- EIP (x86 naming convention) : Instruction Pointer controls the program execution by storing the pointer to the @ of the next instruction that will be executed.
  => In other words : **it tells the CPU where the next instruction is.**

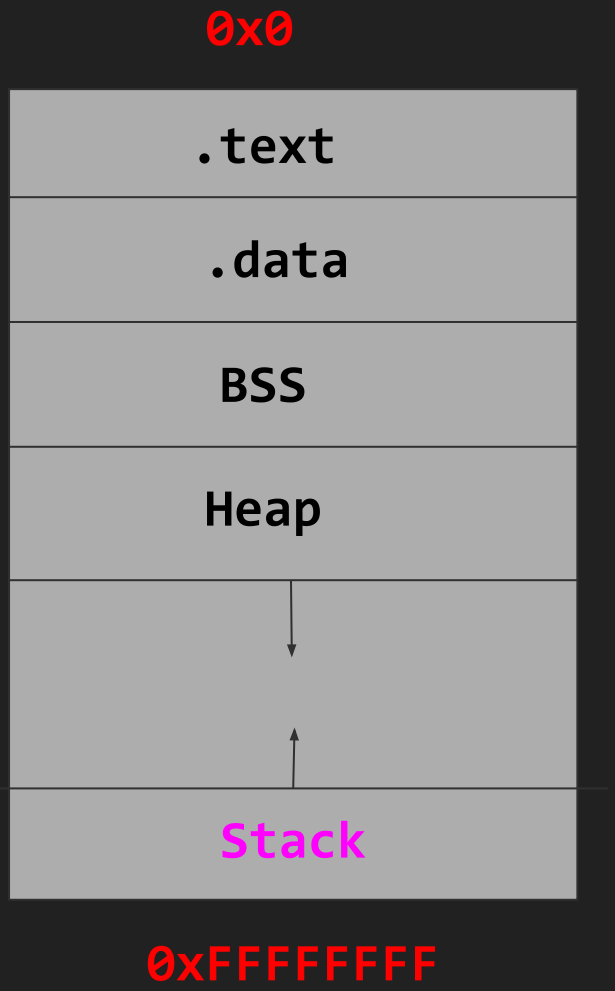
Process Memory

Instructions (Read only)

Initialized variable (static & global declared vars)

Uninitialized variable (eg. static int O)

$ ./demo0



ESP (Stack Pointer)

EBP (Base Pointer)

Stack

PUSH

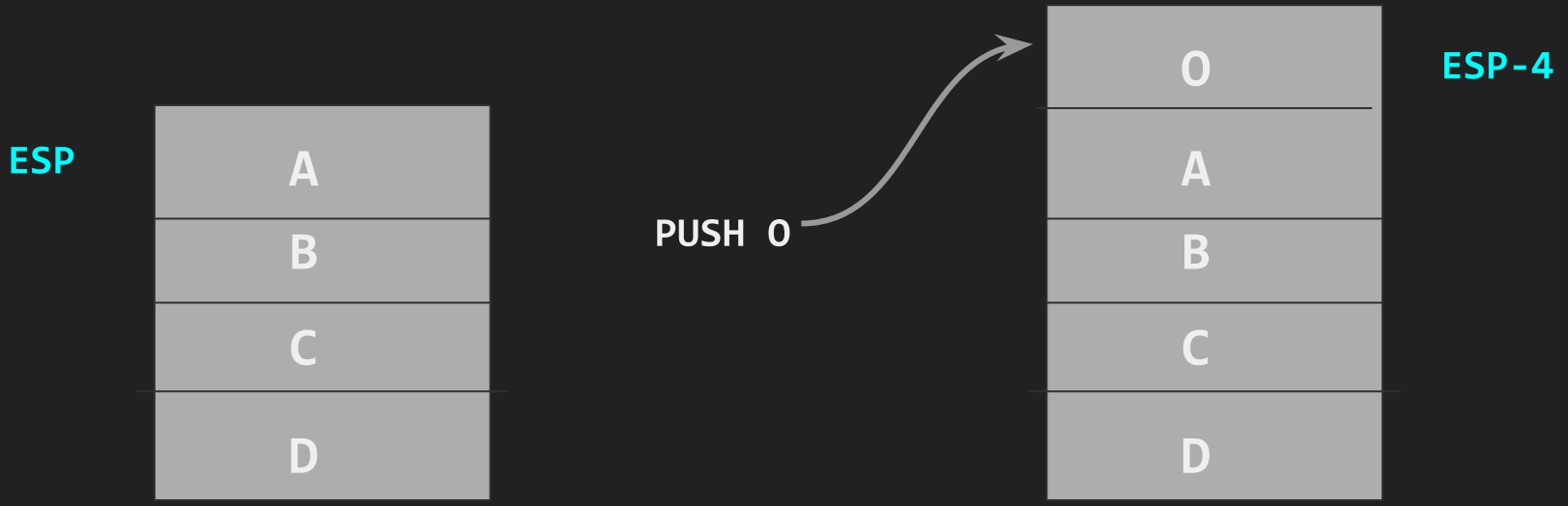
1. *PUSH Instruction*
   -> PUSH 0
2. *PUSH Process*
   -> PUSH executed, ESP modified
3. *Starting Value*
   -> ESP point to the top of the stack

Stack

PUSH

Process:

PUSH subtracts 4 (in 32-bits)or 8 (in 64-bits) form the ESP and writes the data to the memory address in the ESP.



Stack

POP

1. *POP Process*
   -> POP executed, ESP modified

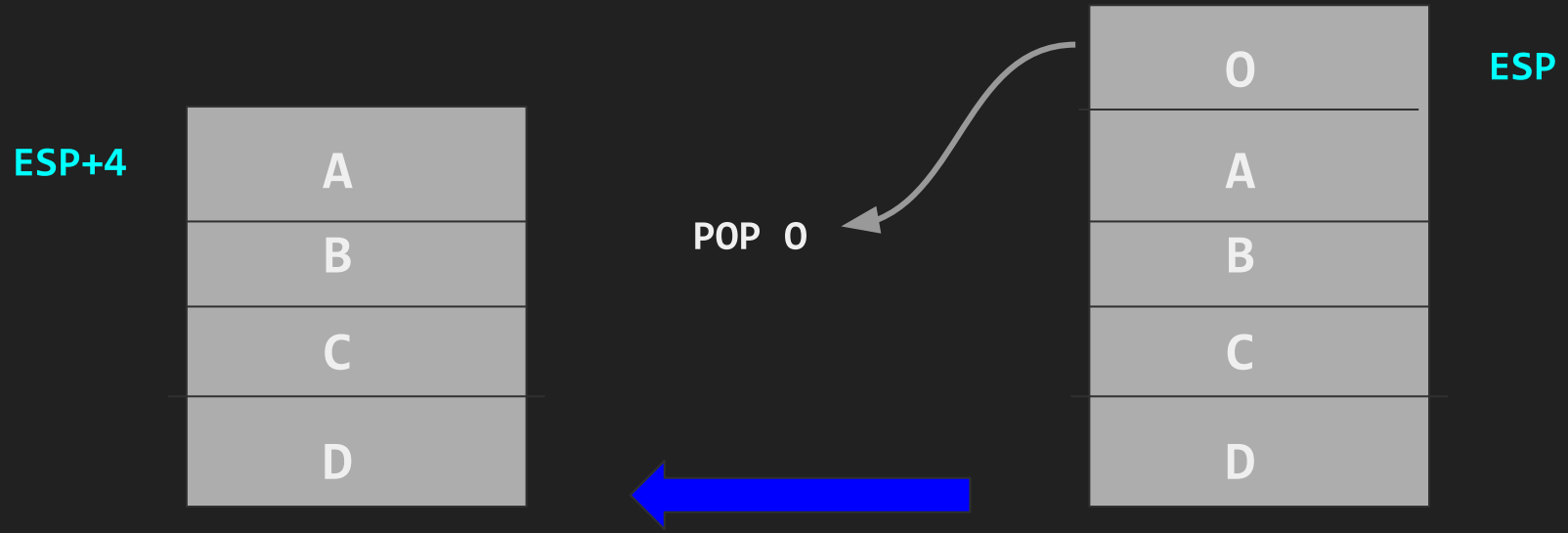
2. *Starting Value*
   -> ESP point to the top of the stack

Stack

POP

Process:

POP is the opposite of PUSH, and it retrieves data from the top of the Stack after that the ESP value is incremented by 4 in x86 or by 8 in x64.



Assembly in depth !

Instruction Format :

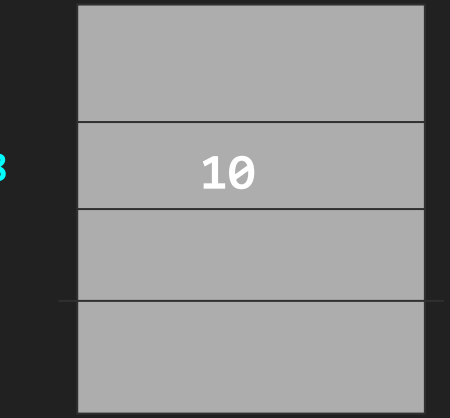
```
operation arg

operation arg1,arg2
```

Assembly in depth !

mov

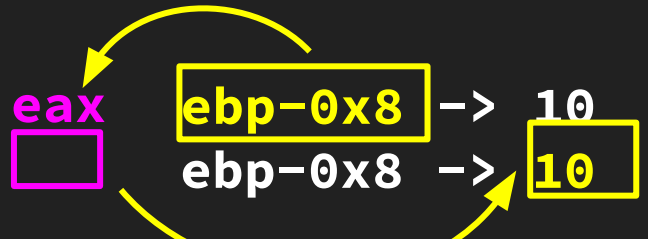
The move instruction just moves data from one register to another.

```
mov arg1,arg2;
```



Example :

```
mov eax,ebp-0x8;
mov eax,[ebp-0x8];
```



Assembly in depth !

add

This just adds the two values together, and stores the sum in the first argument

```
add arg1,arg2;
```

Example :

```
add rax,rdx;
```

=> rax equal to rax+rdx

Assembly in depth !

sub

will subtract the second operand from the first one, and store the difference in the first argument.

```
sub arg1,arg2;
```

Example :

```
sub rsp,0x5;
```

=> rsp equal to rsp-0x5

Assembly in depth !

lea

The lea instruction calculates the address of the second operand, and moves that address in the first

```
lea arg1,[arg2];
```

Example :

```
lea rdi,[rbx+0x5];
```

=> move the @ rbx+0x5 into rdi

Assembly in depth !

xor

This will perform the binary operation xor on the two arguments it is given, and stores the result in the first operation.

```
xor arg1,arg2;
```

Example :

```
xor rdx,rax;
```

=> rdx equl to rdx ^ rax

Assembly in depth !

push

The push instruction will grow the stack by either 8 bytes for x64 (4 for x86), then push the contents of a register onto the new stack space.

```
push arg1;
```

Example :

```
push rax;
```

=> grow the stack by 8-bytes and content of rax => top of the stack

Assembly in depth !

POP

The pop instruction will pop the top 8 bytes for x64 (4 for x86) off of the stack and into the argument. Then it will shrink the stack

```
pop arg1;
```

Example :

```
pop rax;
=> top 8-bytes of the stack => rax
```

Assembly in depth !

Jmp

The jmp instruction will jump to an instruction address.
It is used to redirect code execution.

```
jmp arg1;
```

Example :

```
jmp 0x64262;
```

=> jump to 0x64262 and continue the execution .

Assembly in depth !

Call & ret

Similar to the jmp BUT :

Example :

```
call 0x64262;
    <=>
push rbp;
push rip;
jmp 0x64262;
… ret
pop rip;
pop rbp;
```

Assembly in depth !

Cmp

The cmp instruction is similar to that of the sub instruction. Except it doesn't store the result in the first argument. It checks if the result is less than zero, greater than zero, or equal to zero. Depending on the value it will set the flags accordingly.

Assembly in depth !

jnz / jz

This jump if not zero and jump if zero (jnz/jz) instructions are pretty similar to the jump instruction. The difference is they will only execute the jump depending on the status of the zero flag. For jz it will only jump if the zero flag is set. The opposite is true for jnz

Reversing Assembly

Demo0 :

```c
#include <stdio.h>

void main(void)
{
    puts("Hello CIT!");
}
```

Debugging

```
$ gcc demo0.c -o demo0
$ gdb demo0
```

Reversing Assembly

```
0000000000001135 <main>:
    1135:   55                      push   rbp
    1136:   48 89 e5                mov rbp,rsp
    1139:   48 8d 3d c4 0e 00 00    lea rdi,[rip+0xec4]
    1140:   e8 eb fe ff ff          call   1030 <puts@plt>
    1145:   90                      nop
    1146:   5d                      pop rbp
    1147:   c3                      ret
    1148:   0f 1f 84 00 00 00 00    nop DWORD PTR [rax+rax*1+0x0]
    114f:   00
```

Machine Language

Assembly

Get your hands dirty

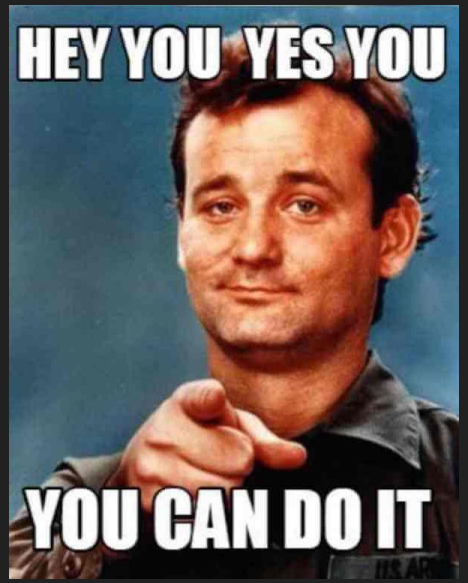
Easy but Tricky !

Demo2 :

Reversing Assembly

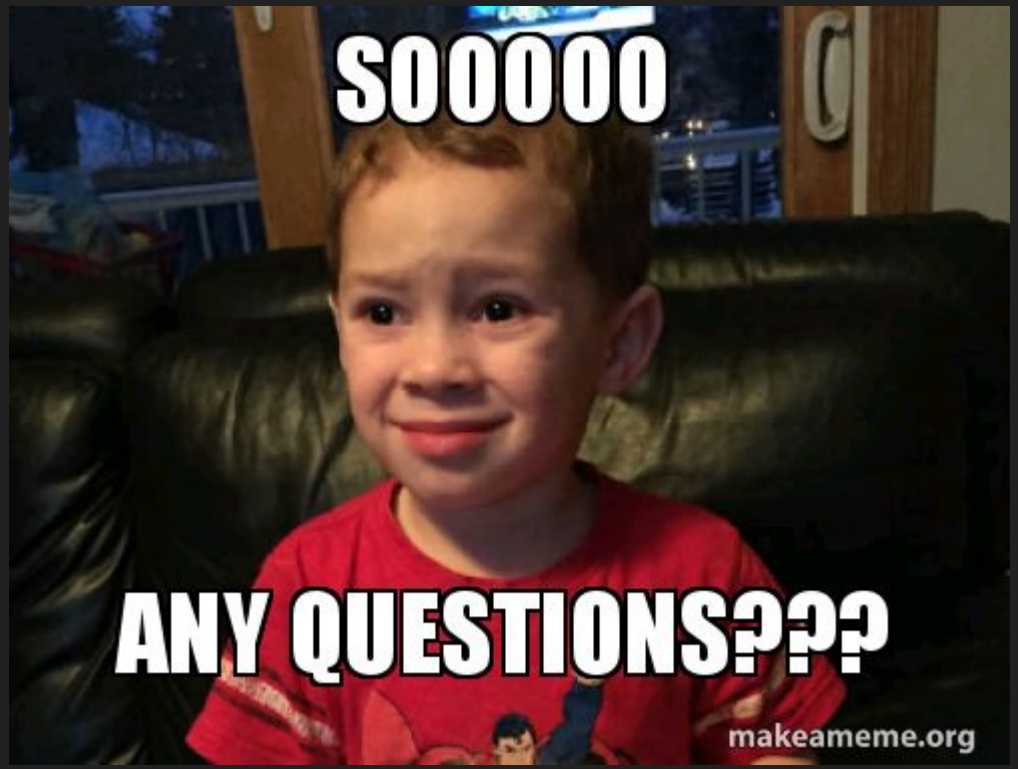
Why EIP [Instruction Pointer] is really important ?

Demo2 :

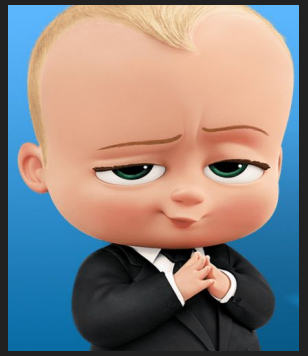


shutdown

tfi dak lmch9of



ls -al .Contact_us



OUSSAMA RAHALI

Facebook : /oussama.rahali.925

OMAR AOUAJ

Facebook : /omar.aouaj.77