

CyberSecurity: Principle and Practice

*BSc Degree in Computer Science
2020-2021*

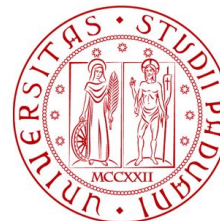
Lesson 11: Intro to Reverse Engineering

Prof. Mauro Conti

Department of Mathematics
University of Padua
conti@math.unipd.it
<http://www.math.unipd.it/~conti/>

Teaching Assistants

Luca Pajola
pajola@math.unipd.it.
Pier Paolo Tricomi
pierpaolo.tricomi@phd.unipd.it



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP



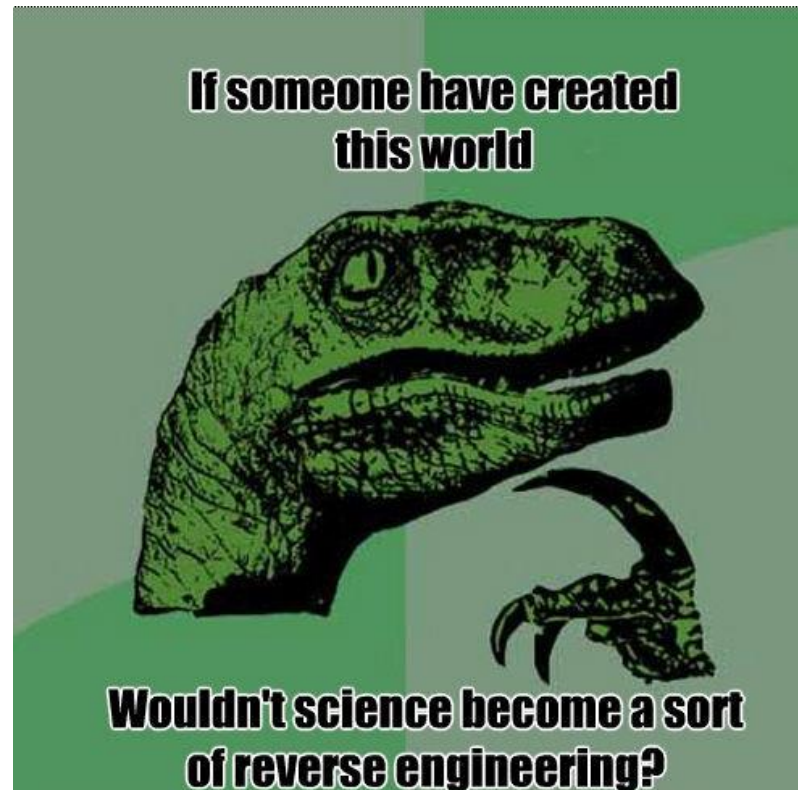
DIPARTIMENTO
MATEMATICA

All information presented here has the only purpose of teaching how reverse engineering works.

Use your mad skillz only in CTFs or other situations in which you are legally allowed to do so.

Do not hack the new Playstation. Or maybe do, but be prepared to get legal troubles 😊

Reverse Engineering?

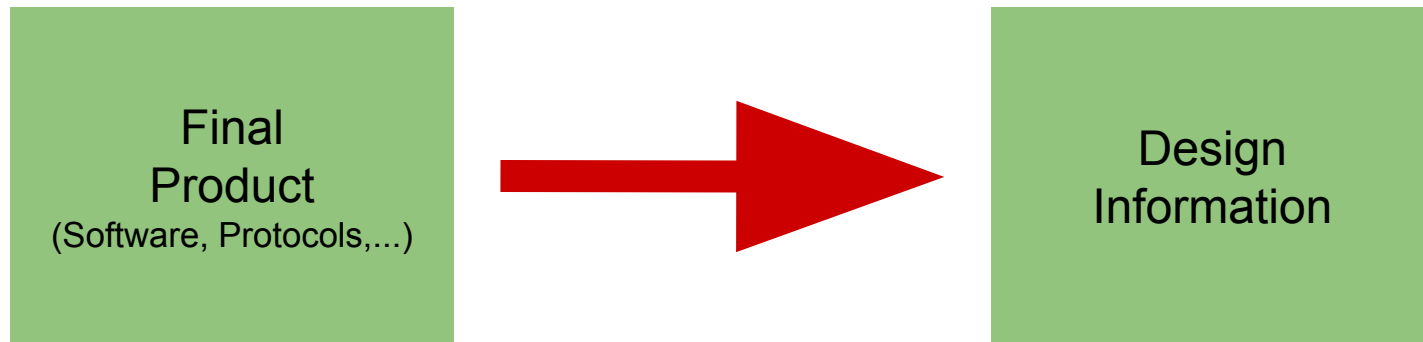


What's Reversing?



“[...] the process of analyzing a subject system to create representations of the system at a higher level of abstraction.”

Chikofsky, Cross (1990)



Why?

- Missing or poor documentation
- Opening up proprietary platforms
- Security auditing
- Curiosity

In reversing challenges you have to understand how a program works, but you don't have its source code.

You typically have to reverse an algorithm (encryption?) to get the flag.

Most of the time, solving a challenge is a bit time consuming but straightforward.

...Unless obfuscation
is involved 😐

```
var myStr = "document.write('My  
Code')";  
eval(myStr);
```

(a) original code

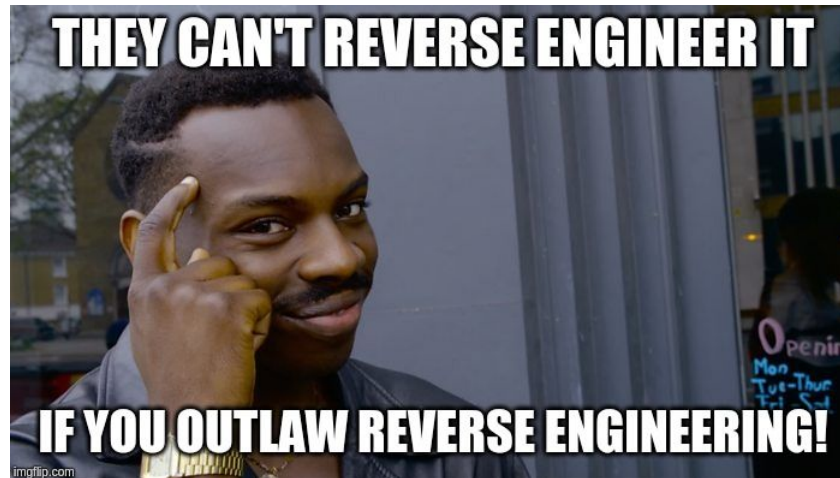
```
var yq = "de");  
var sq = "doc";  
var sm = "ument";  
var kw = "('My Co";  
var myStr = sq + sm + ".write" + kw +  
yq;  
eval(myStr);
```

(b) obfuscated code

Analyze Malwares, remove Ransomwares...

Free Licenses of proprietary software...

A lot of cool stuff, but legally it's a gray area.



Compiling Software

```
Int main() {  
    puts("ILoveCPP");  
    return 0;  
}
```

Source Code



COMPILER

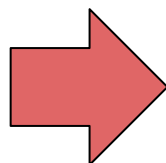
```
00010010010011001  
...
```

Binary

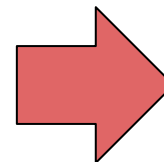
Reversing Software

```
00010010010011001  
...
```

Binary



You

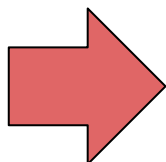
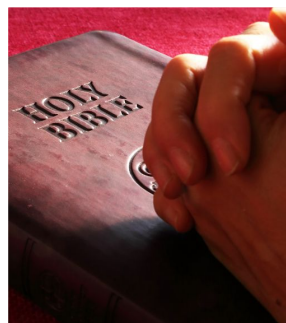


```
Int main() {  
    puts("ILoveCPP");  
    return 0;  
}
```

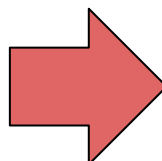
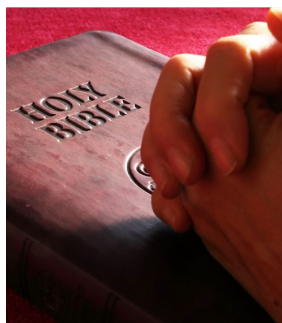
Source Code

Reversing Software (The Truth)

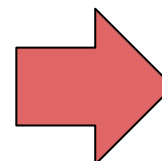
010100
...



```
mov eax, 3  
call func  
ret
```



You



```
Int main() {  
    puts("ILoveCPP")  
;  
    return 0;  
}
```



You

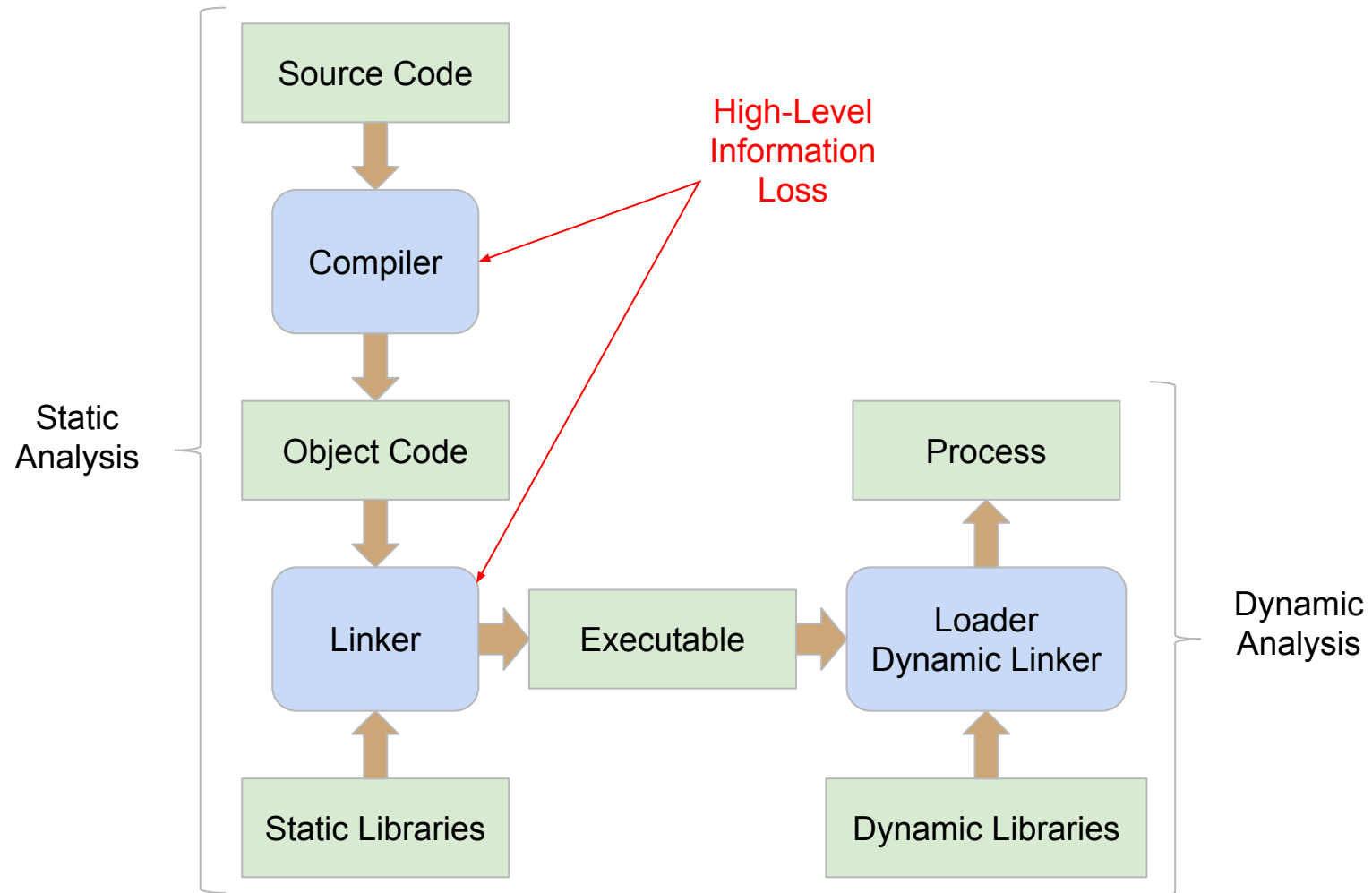


Why is it relevant?



- You don't always have access to source code
- Vulnerability assessment
- Malware analysis
- Pwning
- Algorithm reversing
- Hacking embedded devices
- ...

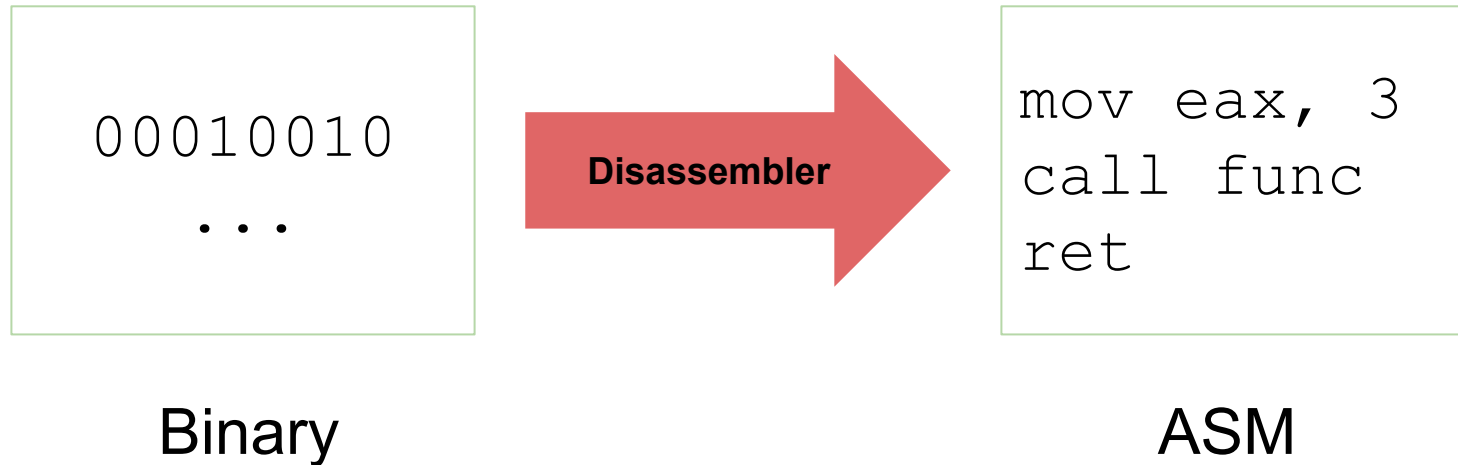
A program's lifecycle



- OS-specific format
 - e.g. ELF (*nix), PE (Windows), Mach-O (MacOS, iOS)
- Generally, same format used for programs and libraries
- Made of sections that will be memory-mapped
 - e.g. .text, .(ro)data, .bss
- Specifies imports from dynamic libraries
 - e.g. GOT/PLT (ELF), IAT (PE)
- Loading methods:
 - Fixed address
 - Relocation
 - Position-independent

- Introduced in System V Release 4, used by most Unix-like OSes
 - Executables, object code, shared libraries, core dumps
- Designed to be flexible, extensible and cross-platform
- Program headers describe segments (i.e. virtual mappings)
- Section headers describe sections and how to load them into segments
- Supports relocation

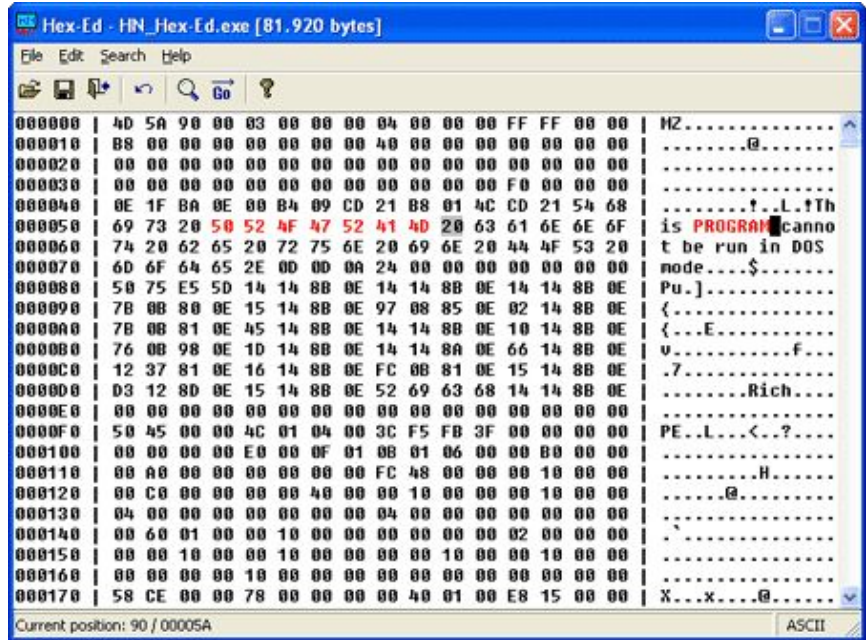
- Static analysis doesn't run the executable
 - Disassembly, decompilation
 - Abstract interpretation
 - Symbolic execution
- Dynamic analysis runs the executable
 - Debugging
 - Dynamic binary instrumentation



- [IDA](#)
 - GUI, Industry standard
 - Pro Version: \$\$\$\$\$. We will use the [free version](#)
- [Radare2](#)
 - CLI (experimental GUI [here](#))
 - Opensource
- [Ghidra](#)
 - NSA reversing tool (open source!)
- Binary Ninja, Objdump
 - If you wish...

- Patch programs
- Inspect file formats
- Change content of files

Many different options here
(bless, hexedit, biew, ...)



- Executable information
 - file, readelf, PEview, ...
- Useful commands
 - strings, ptrace, ltrace...
- Debuggers
 - gdb, WinDbg, OllyDbg, Immunity Debugger, qira, ...
- Decompilers

Can't I just use a decompiler?

Can speed up the reversing, but...

Decompiling is (generally) undecidable

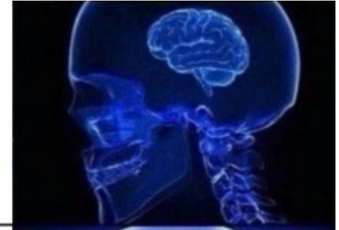
Fails in many cases

Sometimes you want to work at the ASM level (pwning)

Introduction to x86(_64) ASM

- Your computer probably runs on x86_64
 - x86 still supported
 - 32 bit vs 64 bit
- This is NOT supposed to be a complete ASM lesson

X++;



X+=1



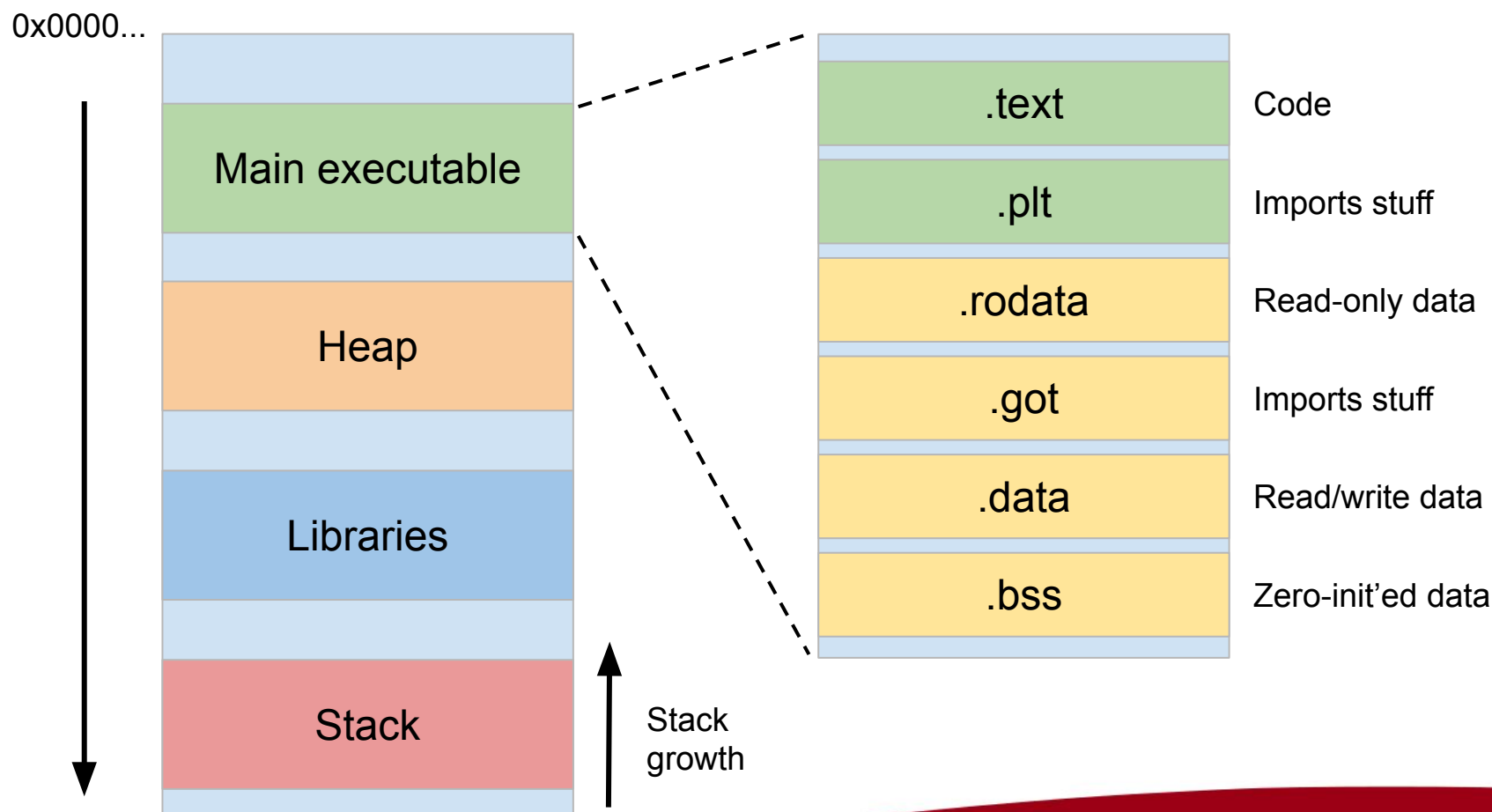
X=X+1



**LD R0,R6
ADD R0,R0,#1
STR R0,R6,#0**



a (Linux) process' memory



Quick Recap



(some) x86_64 Registers

General
Purpose

64 bit	32 bit	16 bit	
RAX	EAX	AX	
		AH	AL
		BX	
		BH	BL
RBX	EBX	CX	
		CH	CL
RCX	ECX	DX	
		DH	DL
RDX	EDX		
RSI	ESI		
RSP	ESP		
RBP	EBP		
RIP	EIP		

Stack Pointer

Base Pointer

Instruction Ptr

Instructions - MOV <dst>, <src>

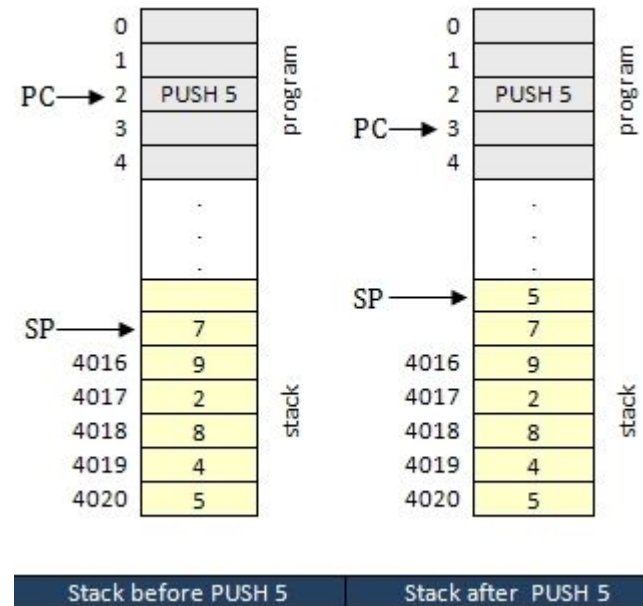
- Copy <src> into <dst>
- MOV EAX, EBX
 - EAX = EBX
- MOV EAX, 16
 - EAX = 16
- MOV EAX, [ESP+4] [X] = “value at address X”
 - EAX = *(ESP+4)
- MOV AL, 'a'
 - AL = 0x61

Instructions - LEA <dst>, <src>

- Load Effective Address of <src> into <dst>
- Used to access elements from a buffer/array
- Used to perform simple math operations
- LEA ECX, [EAX+3]
 - $ECX = EAX + 3$
- LEA EAX, [EBX+2*ESI]
 - $EAX = EBX + 2 * ESI$

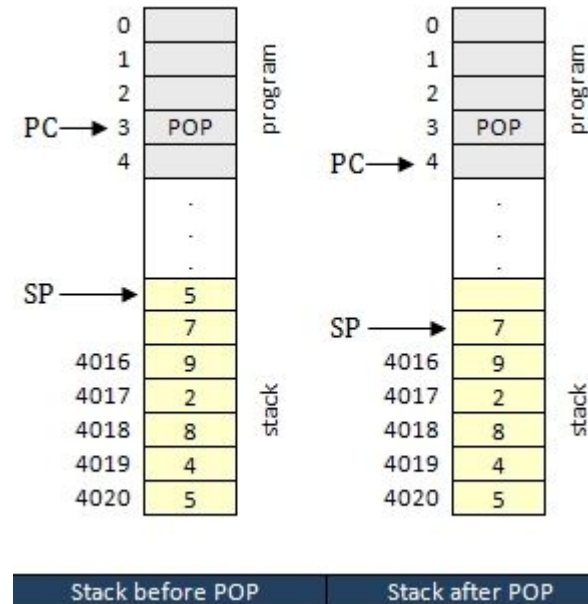
Instructions - PUSH <src>

- Decrement RSP and put <src> onto the stack (push)
- PUSH EAX
 - ESP -= 4
 - *ESP = (dword) EAX
- PUSH CX
 - ESP -= 2
 - *ESP = (word) CX
- PUSH RAX
 - RSP -= 8
 - *RPS = (qword) RAX



Instructions - POP <dst>

- <dst> takes the value on top of the stack, RSP gets incremented
- POP EAX
 - $EAX = *ESP$
 - $ESP += 4$
- POP CX
 - $CX = *ESP$
 - $ESP += 2$



PUSH/POP example

```
PUSH EAX  
POP EBX  
=  
MOV EBX, EAX
```

Instructions - ADD <dst>, <src>

- <dst> += <src>
- ADD EAX, 16
 - EAX += 16
- ADD AH, AL
 - AH += AL
- ADD ESP, 0x10
 - Remove 16 bytes from the stack

Instructions - SUB <dst>, <src>

- <dst> -= <src>
- SUB EAX, 16
 - EAX -= 16
- SUB AH, AL
 - AH -= AL
- SUB ESP, 0x10
 - Allocate 16 bytes of space on the stack

Flags

- x86 instructions can modify a special register called FLAGS
- FLAGS contains 1-bit flags:
 - Ex: OF, SF, ZF, AF, PF, and CF
- ZF = Zero Flag
 - (set if the result of last operation was zero)
- SF = Sign Flag
 - (set if the result of last operation was negative ($\text{dst} - \text{src} < 0$))

Flags - Examples

MOV RAX, 555

SUB RAX, 555

=>

ZF = 1

SF = 0

MOV RAX, 123

SUB RAX, 555

=>

ZF = 0

SF = 1

Instructions - CMP <dst>, <src>

- CoMPare
- Perform a SUB but throw away the result
- Used to set flags
- CMP EAX, 13
 - EAX value doesn't change
 - $TMP = EAX - 13$
 - Update the FLAGS according to TMP

Instructions - JMP <dst>

- JuMP to <dst>
- JMP RAX
 - Jump to the address saved in RAX
- JMP 0x1234
 - Jump to address 0x1234

Instructions - Jxx <dst>

- Conditional jump
- Used to control the flow of a program (ex.: IF expressions)
- JZ/JE => jump if ZF = 1
- JNZ/JNE => jump if ZF = 0
- JB, JA => Jump if <dst> Below/Above <src> (unsigned)
- JL, JG => Jump if <dst> Less/Greater than <src> (signed)
- Many others
- See <http://unixwiz.net/techtips/x86-jumps.html>

Jxx - Example: Password length == 16?

```
MOV RAX, password_length
```

```
CMP RAX, 0x10
```

```
JZ ok
```

```
JMP exit
```

```
ok:
```

```
...print 'length is correct'...
```

Jxx - Example: Given number ≥ 11 ?

```
MOV RAX, integer_user_input
```

```
CMP RAX, 11
```

```
JB fail
```

```
JMP ok
```

```
fail: ...print 'too short'...
```

```
ok: ...print 'OK'...
```

Instructions - XOR <dst>, <src>

- Perform a bitwise XOR between <dst> and <src>
- XOR EAX, EBX
 - $EAX \wedge EBX$
- Truth table:

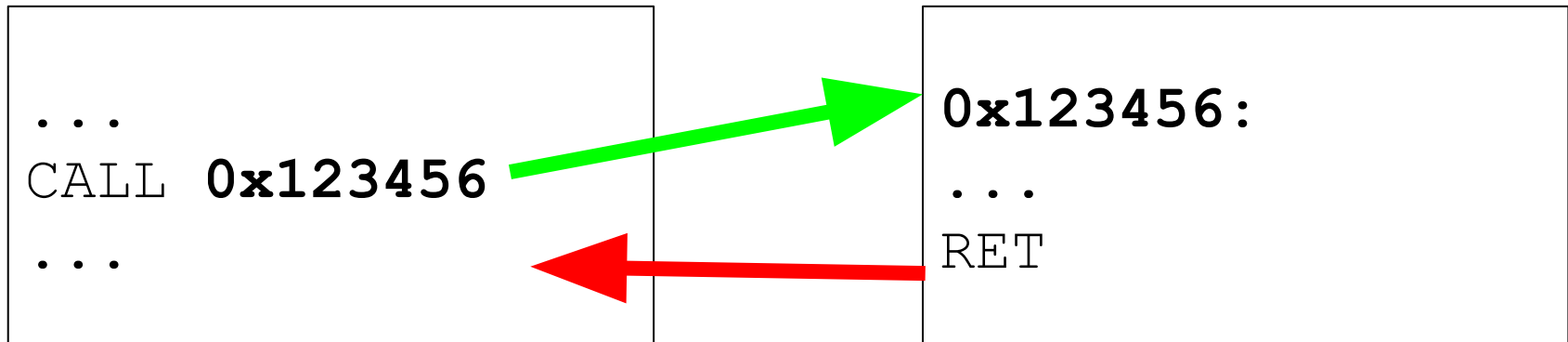
	0	1
0	0	1
1	1	0

Instructions - CALL <dst>

- CALL a subroutine
- CALL 0x123456
 - Push return address on the stack
 - RIP = 0x123456
- Function parameters passed in many different ways

Instructions - RET

- RETurn from a subroutine
- RET
 - Pop return address from stack
 - Jump to it



How are function parameters passed around?

- On x86, there are many calling conventions
- Sometimes parameters are passed in registers
- Sometimes on the stack
- Return value usually in RAX/EAX
- You should take some time to look at them

https://en.wikipedia.org/wiki/X86_calling_conventions

Calling Convention - SystemV AMD64

- Arguments in registers: rdi, rsi, rcx, r8, r9
- Further args on stack
- Red-zoning: leaf function with frames ≤ 128 bytes do not need to reserve stack space

```
int callee(int, int, int);

int caller(void)
{
    int ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

```
caller:
    ; set up stack frame
    push rbp
    mov  rbp, rsp
    ; set up arguments
    mov  edi, 1
    mov  esi, 2
    mov  edx, 3
    ; call subroutine 'callee'
    call callee
    ; use subroutine result
    add  eax, 5
    ; restore old stack frame
    pop  rbp
    ; return
    ret
```


A very useful instruction

NOP - Single-byte instruction that does nothing

Very useful in patching (to remove CALL, CMP,...)

Now it's Demo Time!

- Ex 1 : Can you guess the pin?
- Ex 2: "One of our employees has locked himself out of his account. can you help 'john galt' recover his password? And no snooping around his emails you hear."
- Ex 3: A bomb is going to explode! Defuse the first 4 levels, or go further if you can!

Questions? Feedback? Suggestions?



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

