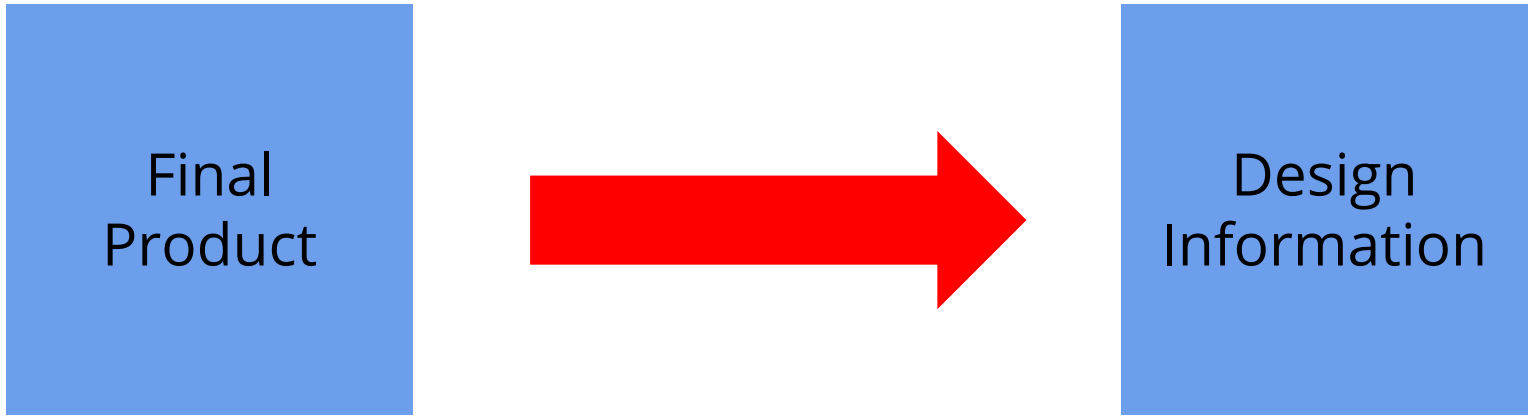
Introduction to Reverse Engineering

Credits:

- Paolo Montesel
- Andrea Biondo

What's reversing?

Not limited to software



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

Reversing in CTFs

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.

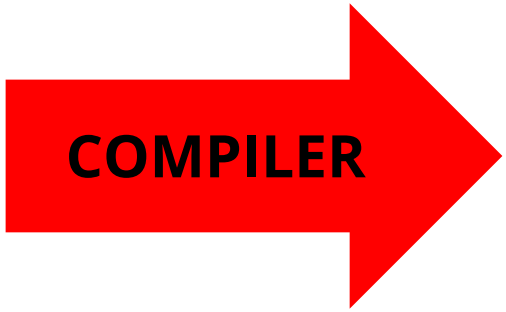
Most of the time, solving a challenge is a bit time consuming but straightforward.

...Unless obfuscation is involved.

(Binary) Software Reverse Engineering

Compiling Software

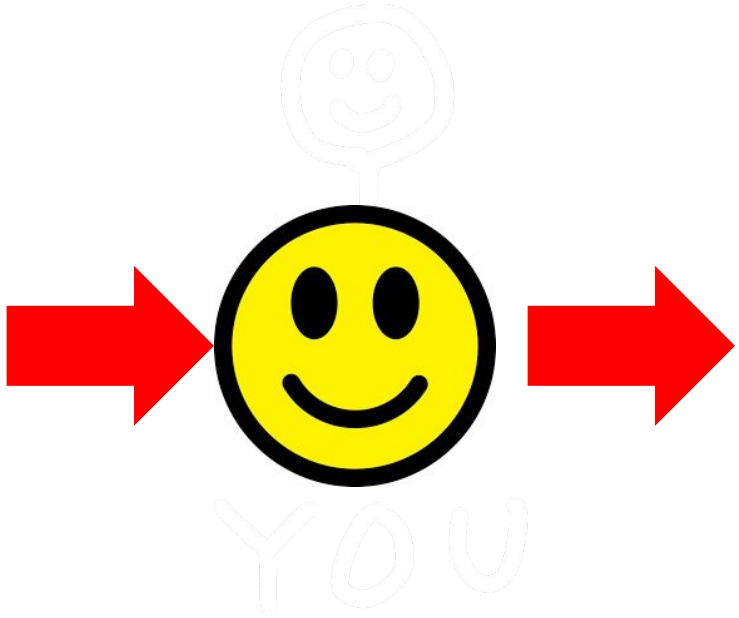
```
int main() {
  puts("YAY");
  return 0;
}
```




```
000100100100
...
```

Reversing Software

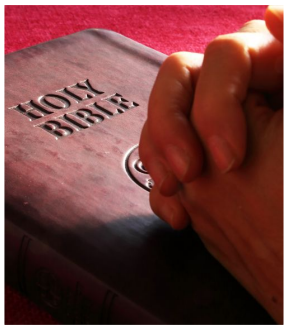
```
000100100100
...
```



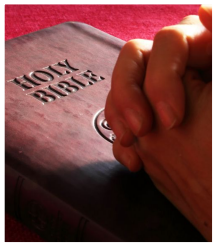

```
int main() {
  puts("YAY");
  return 0;
}
```

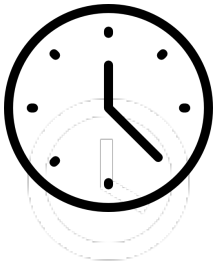
Reversing Software - The Truth

```
00010010
...
```




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



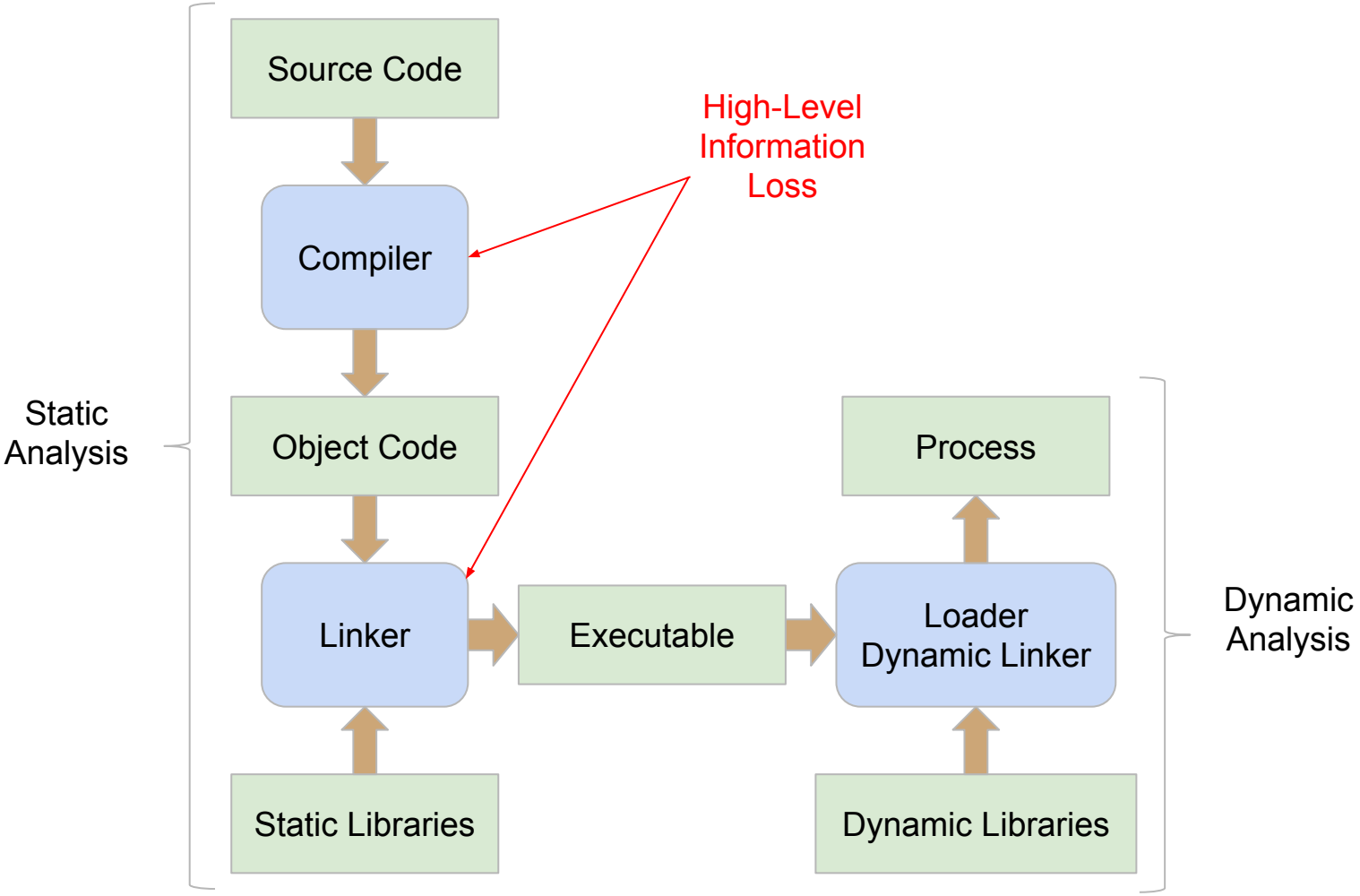


```
int main() {
  puts("YAY");
  return 0;
}
```

Why is it relevant?

- You don’t always have access to source code
- Vulnerability assessment
- Malware analysis
- Pwning
- Algorithm reversing (default WPA anyone?)
- Interoperability (SMB/Samba, Windows/Wine)
- Hacking embedded devices

A program's lifecycle



Executables

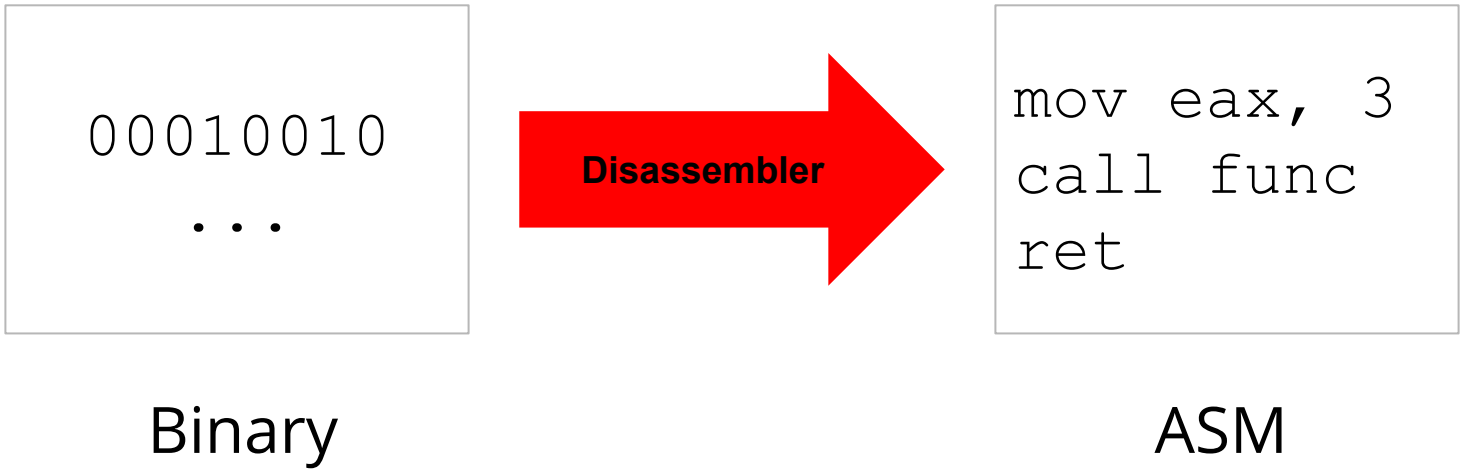
- OS-specific format
  - e.g. ELF (*nix), PE (Windows), Mach-O (MacOS, iOS)
- Generally, same format used for programs and libraries
- Made of *sections* that will be memory-mapped
  - e.g. .text, .(ro)data, .bss
- Specifies imports from dynamic libraries
  - e.g. GOT/PLT (ELF), IAT (PE)
- Loading methods: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

The Tools

Techniques

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

Disassembler



Disassemblers

- **IDA Pro** (https://www.hex-rays.com/products/ida/)
  - GUI
  - Industry standard
  - $$$$$
- **Binary Ninja** (https://binary.ninja/)
  - GUI
  - Very nice scripting features + has “undo” functionality
  - $$
- **Radare2** (https://github.com/radare/radare2)
  - CLI (experimental GUI @ https://github.com/radareorg/cutter/releases)
  - Opensource
- **Ghidra**
  - NSA reversing tool
  - Will be released in the next few days as open-source software!
- **Objdump**

Can’t I just use a decompiler?

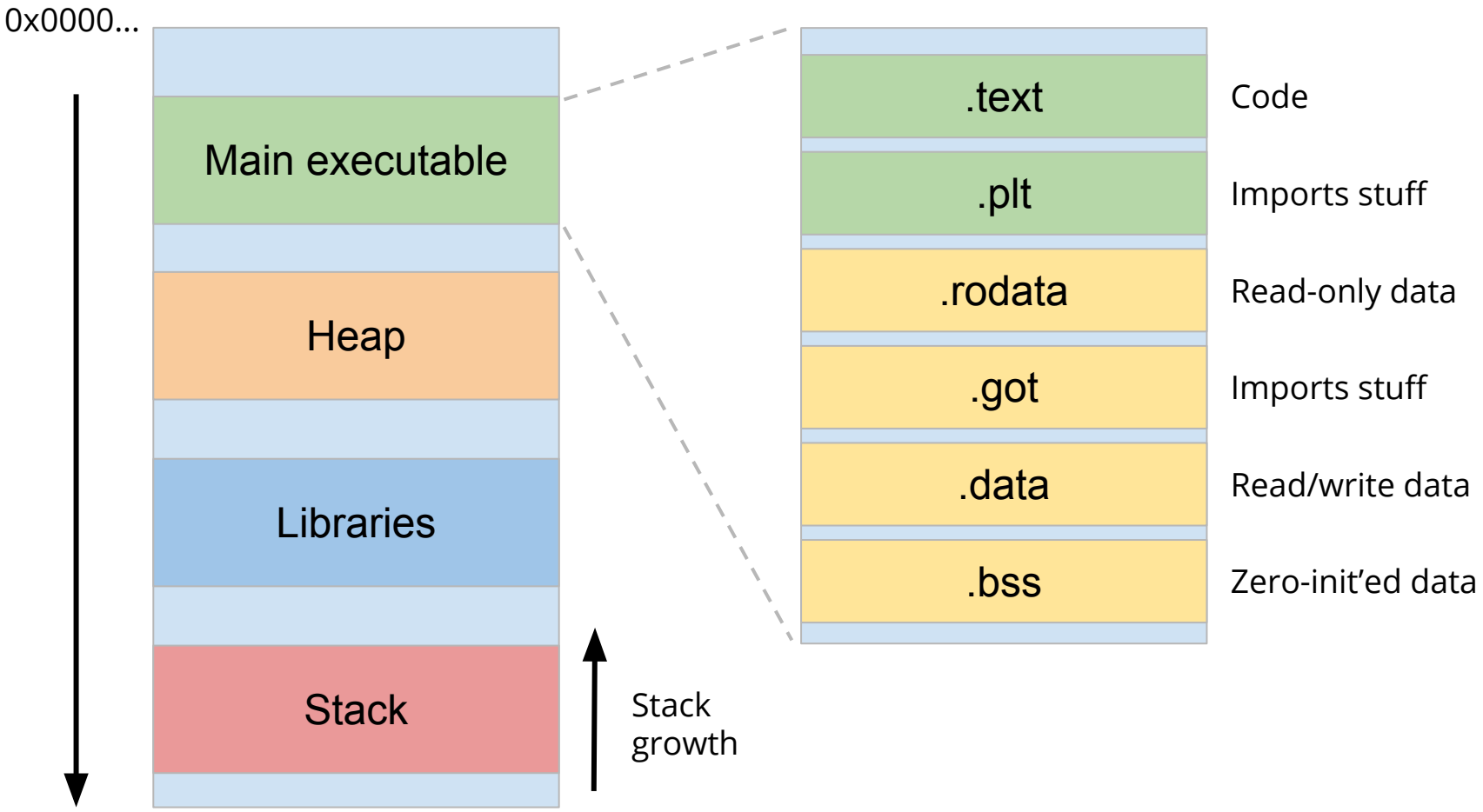
- Can speed up the reversing, but...
- Decompiling is (generally) undecidable
- Fails in many cases
- Sometimes you want to work at the ASM level (pwning)

Introduction to x86 ASM

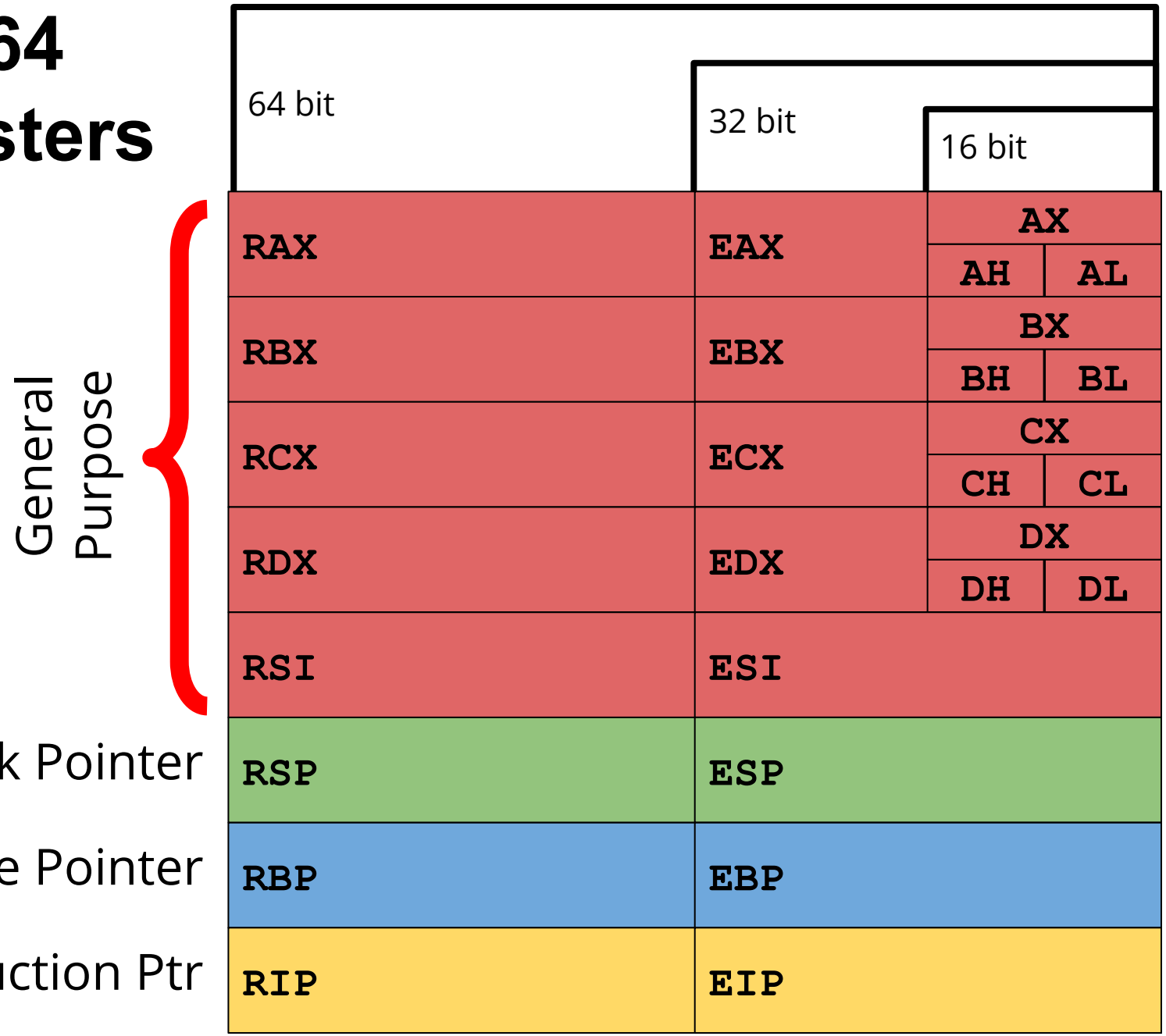
Introduction to x86(_64) ASM

- 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 (booooring)

Quick recap on Linux process' memory



x86_64 Registers



Instructions - MOV <dst>, <src>

- Copy <src> into <dst>
- MOV EAX, 16
  - EAX = 16
- MOV EAX, [ESP+4]
  - 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

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

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

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 ‘yay’...
```

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 ^= EBX
- Truth table:

	0	1
0	0	1
1	1	0

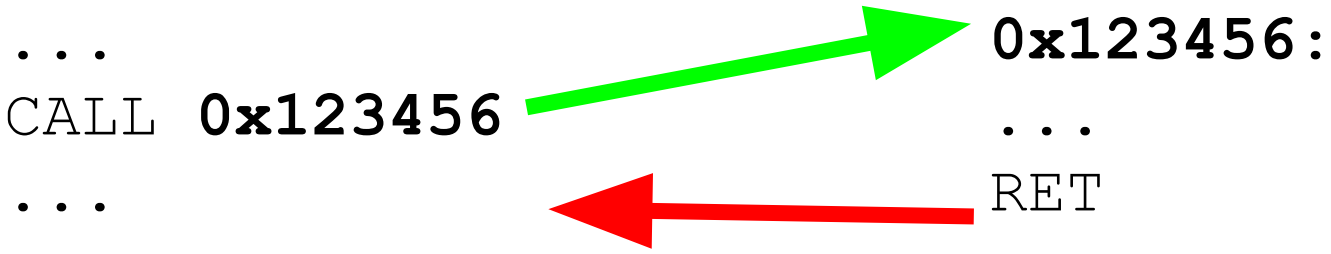
Instructions - CALL <dst>

- 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

CALL / RET



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

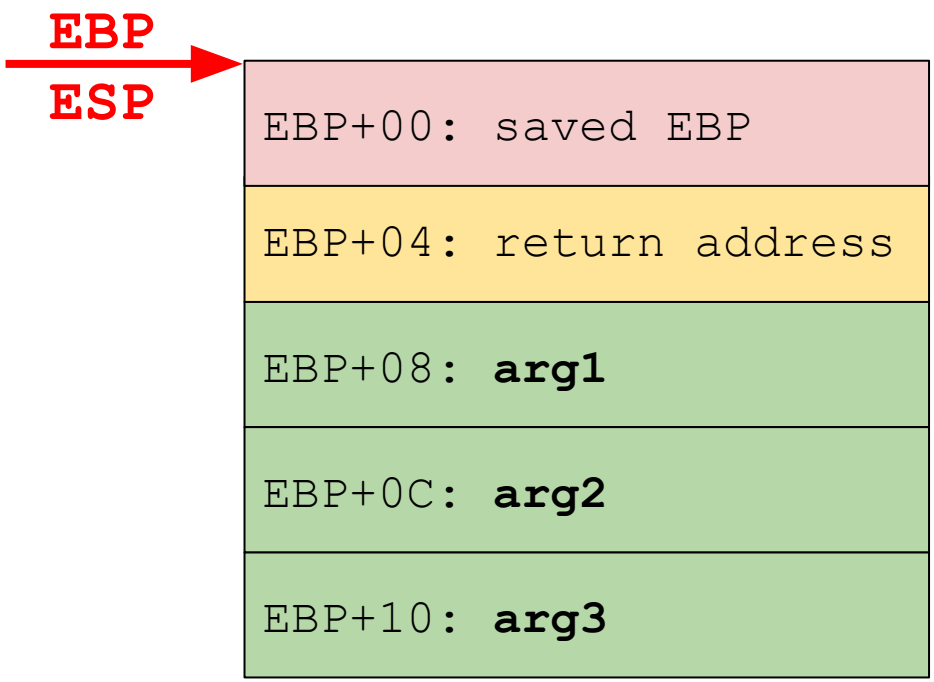
Calling Convention - cdecl

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

int caller(void)
{
    int ret;

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

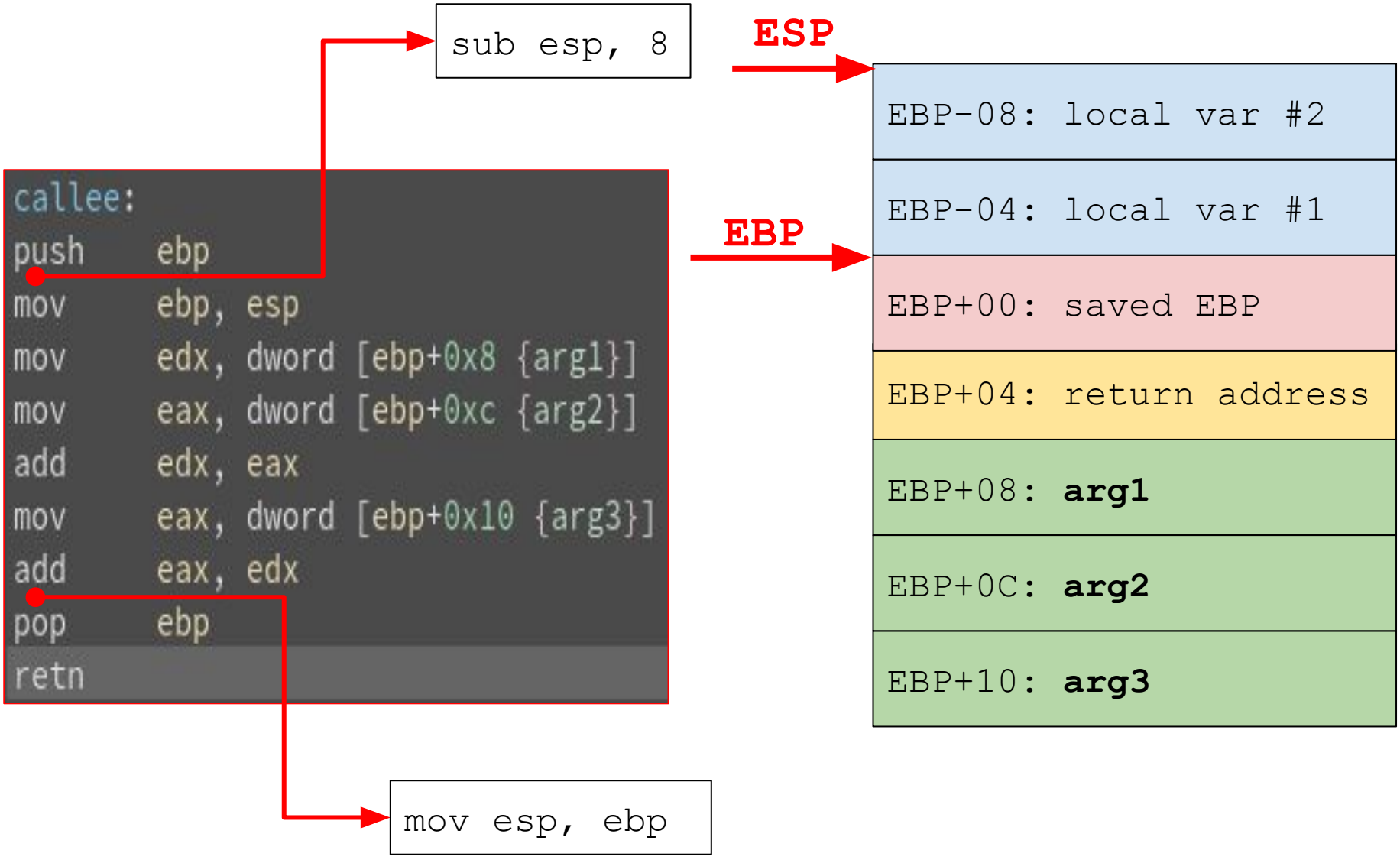
```asm
caller:
    ; make new call frame
    push    ebp
    mov     ebp, esp
    ; push call arguments
    push    3
    push    2
    push    1
    ; call subroutine 'callee'
    call    callee
    ; remove arguments from frame
    add     esp, 12
    ; use subroutine result
    add     eax, 5
    ; restore old call frame
    pop     ebp
    ; return
    ret
```

Calling Convention - cdecl

```
callee:
push    ebp
mov     ebp, esp
mov     edx, dword [ebp+0x8 {arg1}]
mov     eax, dword [ebp+0xc {arg2}]
add     edx, eax
mov     eax, dword [ebp+0x10 {arg3}]
add     eax, edx
pop     ebp
retn
```



Calling Convention - cdecl



Calling Convention - SystemV AMD64

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

```c
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
```

Other useful instructions

NOP - Single-byte instruction that does nothing

RET - Return from a function

MOVZX - Move and zero extend

MOVSX - Move and sign extend