# Program Exploitation Intro

## x86 Assembly

04/10/2018

Security 1

Univeristà Ca' Foscari, Venezia

# What is Program Exploitation

"Making a program do something unexpected and not planned"

- The “right bugs” can be used to subvert code execution
- It is essential to understand how programs are compiled and executed

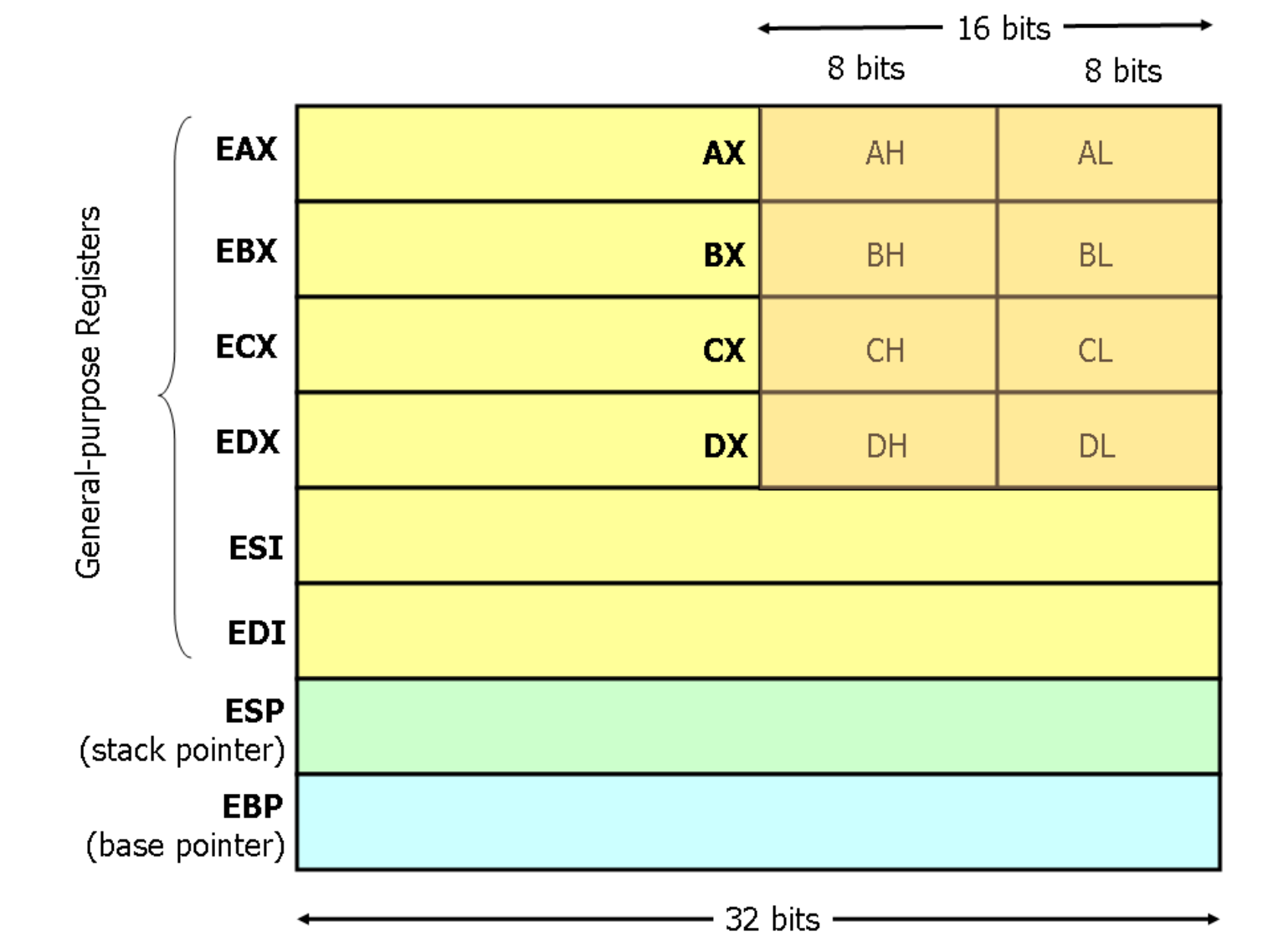
# x86 Assembly

- Executables are written in machine code
- Assembly language makes this code more readable
- We will focus on the 32 bit x86 instruction set

- Intel Syntax: `command <destination>, <source>`
  - `mov eax, 5`
- AT&T Syntax: `command <source>, <destination>`
  - `mov $5, eax`

- We will use the Intel syntax

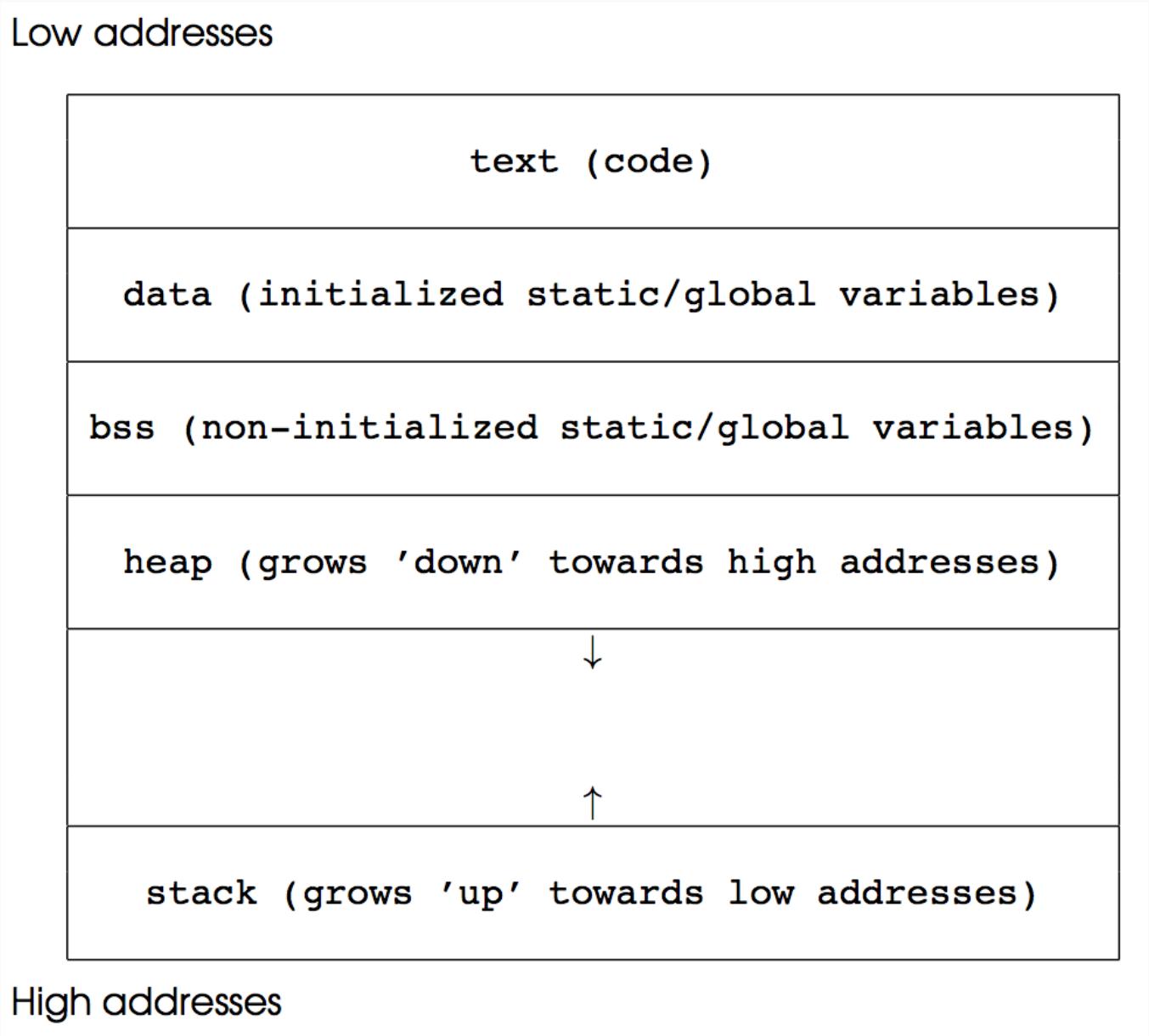
# Registers


16 bits
8 bits
8 bits
General-purpose Registers
EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
EDI
ESP
(stack pointer)
EBP
(base pointer)
32 bits

# Registers

- EAX EBX ECX EDX - General purpose registers
- ESI EDI - Indexes, used in array and string copying
- ESP - Stack pointer, "top" of the current stack frame
- EBP - Base pointer, "bottom" of the current stack frame
- EIP - Instruction pointer, pointer to the next instruction to be executed by the CPU
- EFLAGS - stores flag bits
  - ZF - zero flag, set when result of an operation equals zero
  - CF - carry flag, set when the result of an operation is too large/small
  - SF - sign flag, set when the result of an operation is negative

# Memory Map


Low addresses
text (code)
data (initialized static/global variables)
bss (non-initialized static/global variables)
heap (grows 'down' towards high addresses)
stack (grows 'up' towards low addresses)
High addresses

# The Stack

- Region of memory where local variables are stored
- Supports push and pop operations
- Grows towards lower memory addresses $\rightarrow$ higher values have lower address
- When a function is called a stack frame is set up
  - EBP contains the address of the base of the current stack frame
  - ESP contains the address of the top element of the current stack frame
- Every function call pushes the arguments and the return address to the stack

# Intructions

# Moving data and Arithmetic operations

- mov <dst>, <src>: copies the <src> value to <dst>
- lea <dst>, <src> stands for "load effective address": loads the address of <src> into <dst>

- add <dst>, <src>: adds the value in <src> to <dst>
- sub <dst>, <src>: subtracts the value in <src> from <dst>
- and <dst>, <src>: performs a logical and between <src> and <dst>, placing the result in <dst>
- cmp <dst>, <src>: compares <src> with <dst>. This is done by subtracting <src> from <dst> and updating flags

NOTE: There are various addressing modes!

- `mov DWORD PTR [ebp - 4], eax`
- `mov eax, 3`

# Stack Manipulation

- push <target>: pushes the value in <target> to the stack
- pop <target>: pops a value from the stack into <target>

# Control Flow (Jumps and Calls)

- jle <target>: jumps to the address in <target> if the previously compared <src> was less than or equal to <dst>
- jge <target>: jumps to the address in <target> if the previously compared <src> was greater than or equal to <dst>
- jmp <target>: jumps to the address in <target>. This is achieved by copying the target address into the Instruction Pointer (EIP) register
- call <address>: calls the function at <address>. Before jumping to the function, the address of the next instruction is pushed to the stack in order to be able to return
- ret: pops the return address off the stack and returns control to that location

- nop: **n**o-**op**eration - does nothing

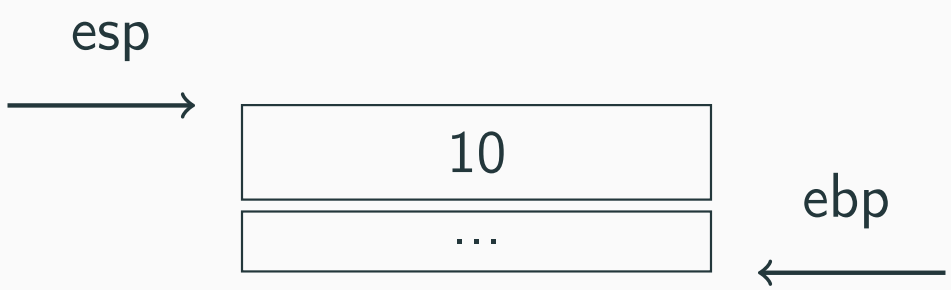
# Function Calls

# Function Calls and Stack Frames

```
func(10);          push 10
                   call func    /* push next inst. addr */
                                /* jmp func */
```

# Function Calls and Stack Frames

```
func(10);          => push 10
                      call func    /* push next inst. addr */
                                   /* jmp func */
```



# Function Calls and Stack Frames

```
func(10);           push 10
                 => call func    /* push next inst. addr */
                                 /* jmp func */
```

# Function Calls and Stack Frames

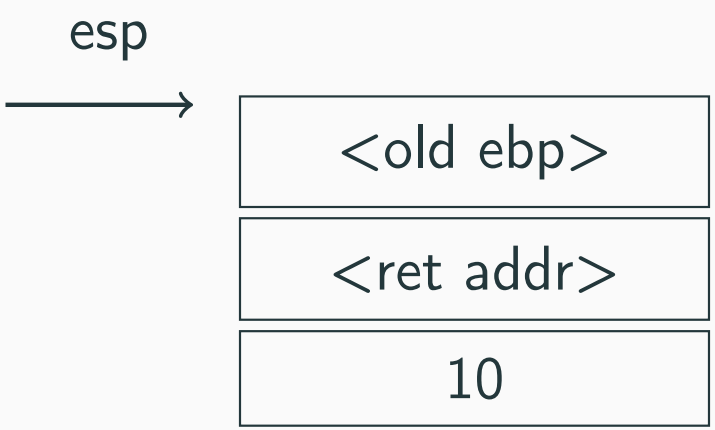
```
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

```
push ebp
mov ebp, esp
sub esp, 8
mov DWORD PTR [ebp - 4], 0
mov eax, DWORD PTR [ebp + 8]
mov DWORD PTR [ebp - 8], eax
```

# Function Calls and Stack Frames

```
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

```
=> push ebp
   mov ebp, esp
   sub esp, 8
   mov DWORD PTR [ebp - 4], 0
   mov eax, DWORD PTR [ebp + 8]
   mov DWORD PTR [ebp - 8], eax
```



# Function Calls and Stack Frames

```
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

```
    push ebp
=> mov ebp, esp
    sub esp, 8
    mov DWORD PTR [ebp - 4], 0
    mov eax, DWORD PTR [ebp + 8]
    mov DWORD PTR [ebp - 8], eax
```

# Function Calls and Stack Frames

```
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

```
    push ebp
    mov ebp, esp
=> sub esp, 8
    mov DWORD PTR [ebp - 4], 0
    mov eax, DWORD PTR [ebp + 8]
    mov DWORD PTR [ebp - 8], eax
```

# Function Calls and Stack Frames

```
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

```
    push ebp
    mov ebp, esp
    sub esp, 8
    mov DWORD PTR [ebp - 4], 0
    mov eax, DWORD PTR [ebp + 8]
=>  mov DWORD PTR [ebp - 8], eax
```

# Function Calls and Stack Frames

```
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

```
...
mov esp, ebp
pop ebp
ret
```

# Function Calls and Stack Frames

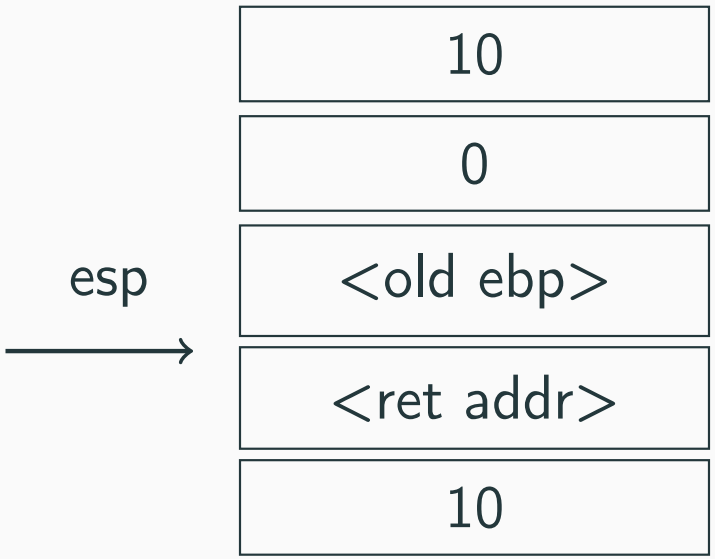
```
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

```
    ...
=> mov esp, ebp
   pop ebp
   ret
```

# Function Calls and Stack Frames

```
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

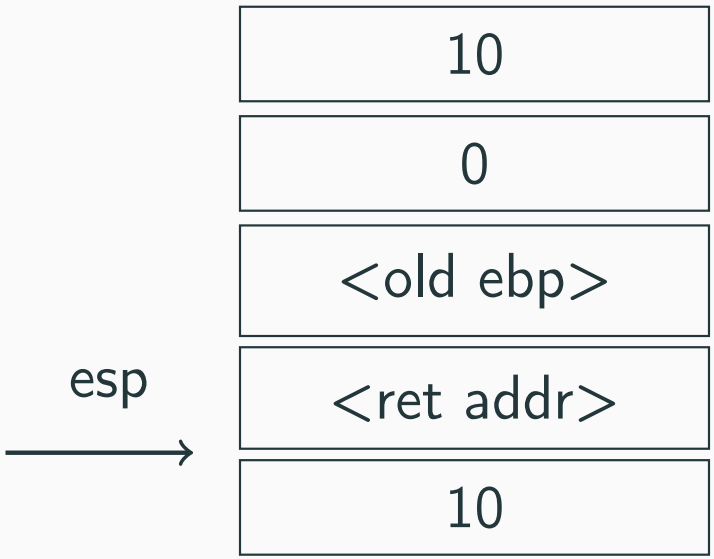
```
    ...
    mov esp, ebp
=>  pop ebp
    ret
```



# Function Calls and Stack Frames

```
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

```
    ...
    mov esp, ebp
    pop ebp
=>  ret
```



# Reading Assembly code

# Tools

- The assembly code of a program can be printed unsing the `objdump` command

```
objdump -M intel -D <program>
```

- Or we can use the `gdb` debugger and print the *disassembly* of a function

```
$ gdb <program>
(gdb) set disassembly-flavor intel
(gdb) disassemble main
```

# Additional Material

# Links

- [Intel Assembly] `https://secgroup.dais.unive.it/teaching/security-course/assembly-e-gdb/`
  **Intel Assembly Materials @ Secgroup**
- [GDB Intro] `https://secgroup.dais.unive.it/teaching/security-course/gdb/`
  **GDB Materials @ Secgroup**
- [x86 ASM Reference] `http://www.mathemainzel.info/files/x86asmref.html`
  **Quick reference to x86 opcodes**