x86 Crash Course

With a focus on Linux and a glance to x86_64

Mario Polino, Armando Bellante, Lorenzo Binosi

March 2, 2022

DEIB, Politecnico di Milano

How to Read the Matrix!

```
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
─────────────────────────────────────────────────────[ REGISTERS ]──────
 EAX  0xffffffe00
 EBX  0x0
 ECX  0x56558570 ← 0x0
 EDX  0x400
 EDI  0xf7fbe000 ← insb   byte ptr es:[edi], dx /* 0x1d4d6c */
 ESI  0xf7fbe5c0 ( _IO_2_1_stdin_ ) ← mov   byte ptr [edx], ah /* 0xfbad2288 */
 EBP  0xffffcde8 → 0xffffd438 → 0xffffd478 → 0xffffd4e8 ← 0x0
 ESP  0xffffcd80 → 0xffffcde8 → 0xffffd438 → 0xffffd478 → 0xffffd4e8 ← ...
 EIP  0xf7fd5079 (__kernel_vsyscall+9) ← pop   ebp
─────────────────────────────────────────────────────[ DISASM ]──────
 ► 0xf7fd5079 <__kernel_vsyscall+9>     pop   ebp
    ↓
   0xf7fd507b <__kernel_vsyscall+11>    pop   ecx
   0xf7fd507c <__kernel_vsyscall+12>    ret

   0xf7ece6e7 <read+39>                 cmp   eax, 0xfffff000
   0xf7ece6ec <read+44>                 ja    read+128 <0xf7ece740>

    ↓
   0xf7ece740 <read+128>                add   esp, 0x14
   0xf7ece743 <read+131>                pop   ebx
   0xf7ece744 <read+132>                pop   esi
   0xf7ece745 <read+133>                jmp   0xf7e01f80

    ↓
   0xf7e01f80                           call  0xf7f1de7d

   0xf7e01f85                           add   edx, 0x1bc07b
─────────────────────────────────────────────────────[ STACK ]──────
00:0000│ esp  0xffffcd80 → 0xffffcde8 → 0xffffd438 → 0xffffd478 → 0xffffd4e8 ← ...
01:0004│      0xffffcd84 ← 0x400
02:0008│      0xffffcd88 → 0x56558570 ← 0x0
03:000c│      0xffffcd8c → 0xf7ece6e7 (read+39) → 0xffff0003d ← 0xfff0003d
04:0010│      0xffffcd90 ← 0x0
... ↓
06:0018│      0xffffcd98 → 0xf7e5bb29 (_IO_file_overflow+9) ← add   edx, 0x1624d7
07:001c│      0xffffcd9c → 0xf7fbc860 (_IO_file_jumps) ← add   byte ptr [eax], al
─────────────────────────────────────────────────────[ BACKTRACE ]──────
 ► f 0 f7fd5079 __kernel_vsyscall+9
   f 1 f7ece6e7 read+39
   f 2 f7e5b9b8 _IO_file_underflow+328
   f 3 f7e5cadb _IO_default_uflow+59
   f 4 f7e405e9 _IO_vfscanf+3945
   f 5 f7e4c4fd __isoc99_scanf+125
   f 6 56555652 main+85
   f 7 f7e01e91 __libc_start_main+241
```

Outline

The x86 Architecture

Instruction Set Architecture (ISA)

- "Logical" specification of a computer architecture
- Concerned with programming concepts
    - instructions, registers, interrupts, memory architecture, . . .
- May differ (widely) from the actual microarchitecture
- Examples:
    - x86 (IA-32 and x86_64)
    - ARM (mobile devices)
    - MIPS (embedded devices, e.g., consumer routers)
   - AVR, SPARC, Power, RISC V, . . .

The x86 ISA

- Born in 1978, 16-bit ISA (Intel 8086)
- Evolved to a 32-bit ISA (1985, Intel 80386)
- Evolved to a 64-bit ISA (2003, AMD Opteron)
- CISC design (e.g., string operations)
- Many legacy features (e.g, segmentation)
- We'll see the basics of the "core" ISA
  - There is also the floating point unit, processor-specific features, and extensions such as SIMD (MMX, SSE, SSE2) with their own instructions and registers[1]

---

[1]Complete reference: Intel Software Developer's Manual, about 5,000 pages
(https://software.intel.com/en-us/articles/intel-sdm)
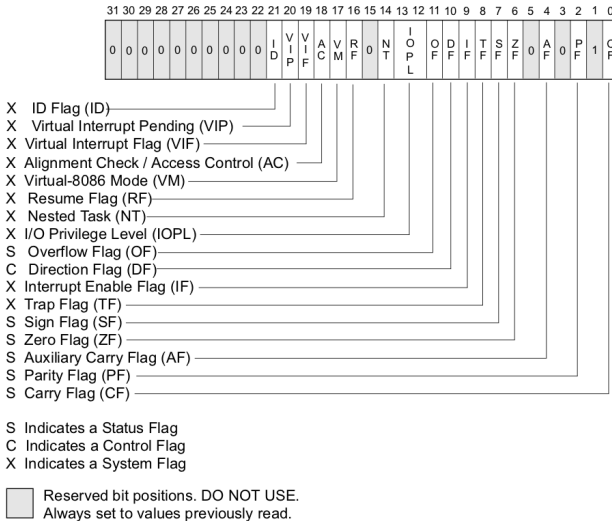
Von Neumann Architecture
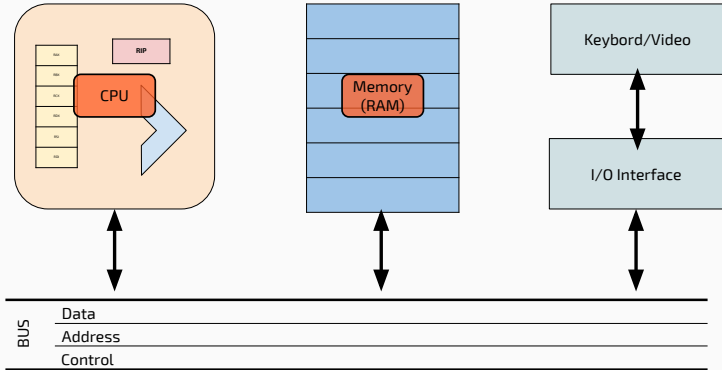
IA-32: Registers
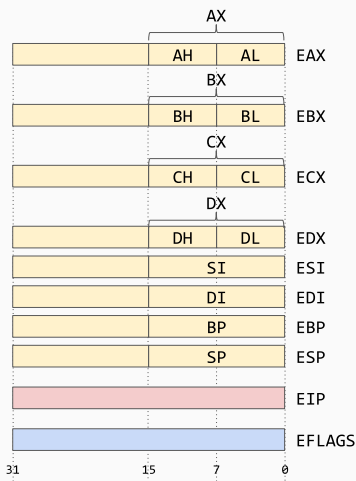
- General-purpose registers
    - EAX, EBX, ECX, EDX
    - ESI, EDI (source and destination index for string operations)
    - EBP (base pointer)
    - ESP (stack pointer)
- Instruction pointer: EIP
    - No explicit access
    - Modified by jmp, call, ret
    - Read through the stack (saved IP)
- Program status and control: EFLAGS
- (segment registers)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.

IA-32: EFLAGS register

- 32-bits register, boolean flags
- **Program status**: overflow, sign, zero, auxiliary carry (BCD), parity, carry
  - Indicate the result of arithmetic instructions
  - Extremely important for control flow
- **Program control**: direction flag
  - controls string instructions (auto-increment or auto-decrement)
- **System**: control operating-system operations
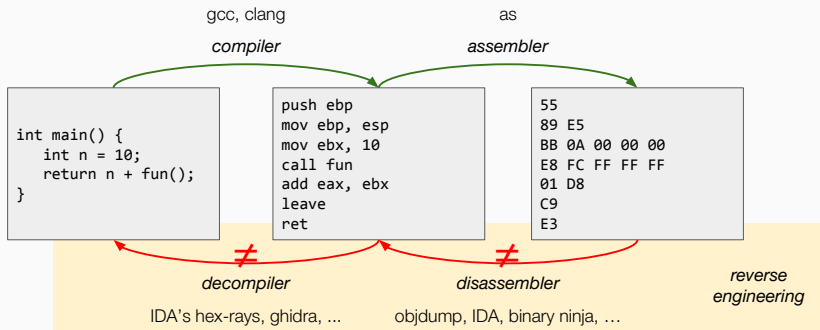
Fundamental data types

byte 8 bits
word 2 bytes
dword Doubleword, 4 bytes (32 bits)
qword Quadword, 8 bytes (64 bits)

Assembly and Machine Code

Assembly language: specific to each ISA, mapped to binary code

For simplicity, we don't deal with the *linking* process.

Assembly: Syntax

Two main syntaxes:

- **Intel**: default in most Windows programs (e.g., IDA)
- **AT&T**: default in most UNIX tools (e.g., gdb, objdump)

Beware: The order of the operands is **different**

We will use the Intel syntax

move the value 0 to EAX

Intel	**AT&T**
mov eax, 0h	movl $0x0,%eax

move the value 0 to the address contained in EBX+4

Intel	**AT&T**
mov [ebx+4h],0h	movl $0x0,0x4(%ebx)

x86: data movement

Examples

Immediate to register:

```
mov eax, 4h              EAX = 4
```

Register to register:

```
mov eax, ebx             EAX = EBX
```

Memory to register (and register to memory):

```
mov eax, [ebx]           EAX = *EBX
mov eax, [ebx + 4h]      EAX = *(EBC + 4)
mov eax, [edx + ebx*4 + 8]   EAX = *(EDX + EBX * 4 + 8)
```

Note: memory to memory is an **invalid** combination[2]

---

[2]Except in some instructions, such as movs (move from string to string).

x86 Assembly and Machine Code

Instruction = opcode + operand

Example

	mov r/m r	ebp, esp
`mov ebp, esp`	89	E5 10
	mov bl	
`mov bl, 10`	B3	0A
	size prefix	
`mov bx, 10`	(66)(BB)	0A 00
	mov ebx	
`mov ebx, 10`	(BB)	0A 00 00 00
	prefix + opcode (1-3 bytes)	*operands*

Beware: in x86, instructions have **variable length**.

Basic instructions

- **Data Transfer**: `mov, push, pop, xchg, lea`
- **Integer Arithmetic**: `add, sub, mul, imul, div, idiv, inc, dec`
- **Logical**: `and, or, not, xor`
- **Control Transfer**: `jmp, jne, call, ret`
- and lots more...

Data Transfer: `mov`

- `mov` **destination**, **source**
  source: immediate, register, memory location
  destination: register or memory location
- Basic load/store operations
  - Register to register, register to memory, immediate to register, immediate to memory
  - Memory to memory is INVALID (in every instruction)

Examples

MOV eax, ebx	MOV eax, FFFFFFFFh	MOV ax, bx
MOV [eax],ecx	MOV [eax],[ecx] NO!!!	MOV al, FFh

17

Integer Arithmetics: add and sub

add **destination**, **source**	sub **destination**, **source**
dest ← dest + source	dest ← dest − source

- Addressing:
  source: immediate, register, memory location
  destination: register or memory location
  (the destination has to be at least as large as the source)
- Negate a value: neg [op]
- Bitwise operations: and, or, xor, not work similarly

Examples

add esp, 44h	add edx, cx	add al, dh
sub esp, 33h	sub eax, ebx	sub [eax], 1h

Integer Arithmetics: unsigned multiply (`mul`)

- `mul` **source**

  source: register or memory location
- dest $\leftarrow$ implied_op $\times$ source
- **Implied operands** according to the size of **source**
    - First operand: `AL`, `AX`, or `EAX`
    - Destination: `AX`, `DX:AX`, `EDX:EAX` (double the size of **source**)
- Signed multiply: `imul`

Example

- `mul ebx`: `EDX:EAX` $\leftarrow$ `EAX * EBX`
    - most significant bits of the result in `EDX`
    - least significant bits of the result in `EAX`
- `mul cx`: `DX:AX` $\leftarrow$ `AX * CX`
- `mul cl`: `AX` $\leftarrow$ `AL * CL`

Integer Arithmetics: unsigned divide (div)

- div **source**

  source: register or a memory location
- Computes quotient and remainder
- Implied operand: EDX:EAX (according to the size of **source**)
- Signed divide: idiv

Examples

- div ebx (4 bytes)
  - EAX $\leftarrow$ EDX:EAX / EBX
  - EDX $\leftarrow$ EDX:EAX % EBX
- div bx (2 bytes)
  - AX $\leftarrow$ DX:AX / BX   DX = DX:AX % BX
- div bl (1 byte)
  - AL $\leftarrow$ AX / BX   AH = AX % BX

Integer Arithmetics: `cmp` **and** `test`

`cmp` **op1**, **op2**	`test` **op1**, **op2**
Computes op1 − op2	Computes op1 & op2

- Sets the flags (`ZF`,`CF`, `OF`, . . .)
- Discards the result

Examples

`cmp eax, ebx`	`cmp eax, 44BBCCDDh`	`cmp al, dh`
`cmp al, 44h`	`cmp ax,FFFFh`	`cmp [eax],4h`

Control-Flow Instructions: unconditional jump `jmp`
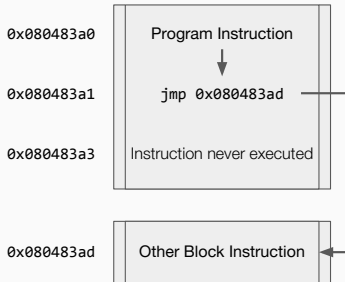
- `jmp` **address** or **offset**
- Unconditional jump: just set the EIP to **address**
- Can be also *relative*: increment or decrement EIP by an offset

Control-Flow Instructions: conditional jumps

j<cc> __address__ or __offset__

Jump to **address** if and only if a certain condition is verified

<cc>: condition

- O,NO,S,NS,E,Z,NE, . . .
- based on one or more status flags of EFLAGS

Examples:

Program Instruction	
0x0804839c	cmp eax, ebx
0x080483a1	jz 0x080483ad
	if ZeroFlag == 0
0x080483a3	Instruction (jump not taken)
	if ZeroFlag == 1
0x080483ad	Instruction (if jump taken)

- jz = jump if zero
- jg = jump if greater than
- jlt = jump if less than

Reference: http://www.unixwiz.
net/techtips/x86-jumps.html
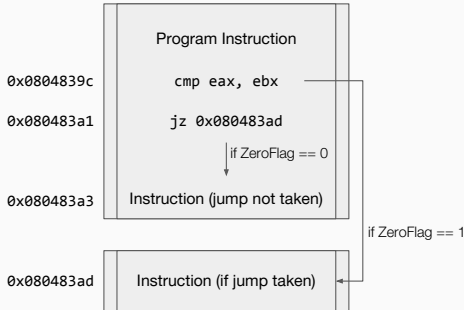
A very simple example (what does it do?)

Assume that the input is in registers: ECX and EDX; output: EAX

```
        mov eax, ecx
        mov ebx, edx
        cmp ebx, 0
        jz label
loop:
        cmp ebx, 1
        jle ret
        mul ecx
        sub ebx, 1
        jmp loop
label:
        mov eax, 1
ret:
        ...
```

Load effective address (`lea`)

- `lea` **destination**, **source**
  source: memory location
  destination: register
- Like a mov, but it is storing the pointer, not the value
- It does NOT access memory

Example

	Registers		Memory	
EAX	0x00000000		0x7C81776F	0x00403A40
			0x7C911000	0x00403A44
EBX	0x00403A40		0x0012C140	0x00403A48
	...		0x7FFDB000	0x00403A4C

$$\text{lea eax, [ebx + 8]} \rightarrow \text{EAX = 0x00403A48}$$
N.B.: with mov eax, [ebx+8] → EAX = 0x0012C140

Basic Instructions: `nop`

- `nop` = **No Operation**. Just move to next instruction.
- The opcode is pretty famous and is 0x90
- Really useful in exploitation (we will see!)

Interrupts and Syscalls

- int **value**
    - **value**: software interrupt number to generate (0-255)
    - Every OS has its set of interrupt numbers (e.g., 80h for Linux system calls)
- syscall used for Linux 64-bit
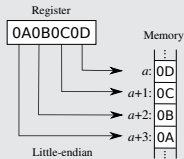- sysenter used by Microsoft Windows
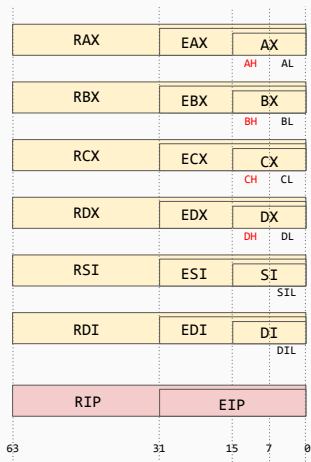
Endianness

Endianness: convention that specifies in which order the bytes of a data word are lined up sequentially in memory.

Big-endian (left)

Systems in which the *most significant byte* of the word is stored in the *smallest address* given.
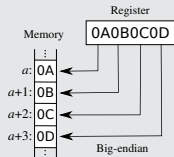
Little-endian

Systems in which the *least significant* byte is stored in the *smallest address*. IA-32 is "little endian".

Program Layout and Functions

Binary File Formats

- **PE (Portable Executable):** used by Microsoft binary executables
- **ELF:** common binary format for Unix, Linux, FreeBSD and others
- In both cases, we are interested in how each executable is mapped into memory, rather than how it is organized on disk.

How an executable is mapped to memory in Linux (ELF)

Executable	Description
.bss	This section holds uninitialized data that contributes to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run.
.comment	This section holds version control information.
.data/.data1	These sections hold initialized data that contribute to the program's memory image
.debug	This section holds information symbolic debugging.
.text	This section holds the "text," or executable instructions, of a program.
.init	This section holds executable instructions that contribute to the process initialization code. That is, when a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called main for C programs).
.got	This section holds the global offset table.

How an executable is mapped to memory in Windows (PE)

Executable	Description
.text	Contains the executable code
.rdata	Holds read-only data that is globally accessible within the program
.data	Stores global data accessed throughout the program
.idata	stores the import function information;
.edata	stores the export function information;
.pdata	Present only in 64-bit executables and stores execption-handling information
.rsrc	Stores resources needed by the executable
.reloc	Contains information for relocation of library files

Simplified program memory layout

Low addresses (0x80000000)

Shared libraries
`.text`
`.bss`
Heap (grows ↓)
...
Stack (grows ↑)
`env`
`argv`
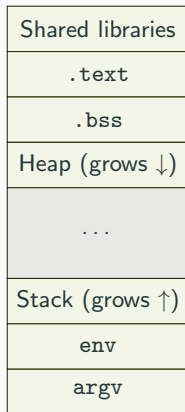
High addresses (0xbfffffff)

The Stack

- LIFO (last in first out) data structure
- Used to manage functions
  - local variables
  - return addresses
  - ...
- Handled through the register ESP (stack pointer)
- Remember: the stack grows **toward lower addresses** (downward the address space)

Stack Management Instructions: push

<div align="center">

push **immediate** or **register**

</div>

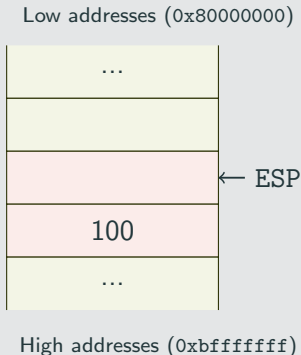Stores the immediate or register value at the top of the stack and decrements the ESP of the operand size
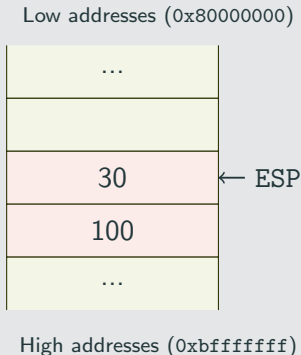
Example

Initial condition: EAX = 30

push eax

is equivalent to:

sub esp, 4
mov DWORD PTR [esp], eax

Low addresses (0x80000000)

...
100	← ESP
...

High addresses (0xbfffffff)

Stack Management Instructions: push

<div align="center">

push **immediate** or **register**

</div>

Stores the immediate or register value at the top of the stack and decrements the ESP of the operand size

Example

Initial condition: EAX = 30

push eax

is equivalent to:

sub esp, 4
mov DWORD PTR [esp], eax

Low addresses (0x80000000)

...
	← ESP
100
...

High addresses (0xbfffffff)

push **immediate** or **register**

Stores the immediate or register value at the top of the stack and decrements the `ESP` of the operand size

Example

Low addresses (0x80000000)

Final condition: `EAX = 30`

`push eax`

is equivalent to:

`sub esp, 4`
`mov DWORD PTR [esp], eax`

...
30	← ESP
100
...

High addresses (0xbfffffff)

Stack Management Instructions: `pop`

pop **destination**

Loads to the destination a word off the top of the stack, then it increases ESP of the operand's size.
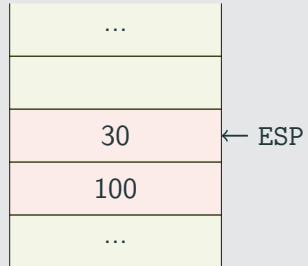
Example

Initial condition: EAX = ???

```
pop eax
```

is equivalent to:
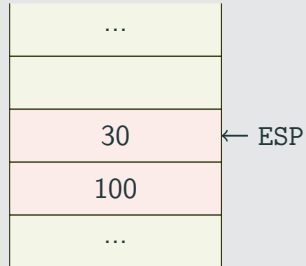
```
mov eax, DWORD PTR [esp]
add esp, 4
```

Low addresses (0x80000000)

| ... |
| |
| 30 | ← ESP |
| 100 |
| ... |

High addresses (0xbfffffff)

Stack Management Instructions: pop

<div align="center">

pop **<u>destination</u>**

</div>

Loads to the destination a word off the top of the stack, then it increases ESP of the operand's size.
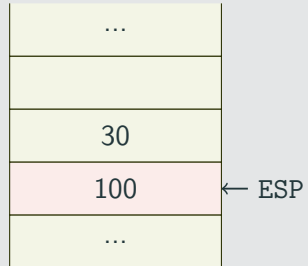
Example

Initial condition: EAX $= 30$

pop eax

is equivalent to:

```
mov eax, DWORD PTR [esp]
add esp, 4
```

Low addresses (0x80000000)

...
30	← ESP
100
...

High addresses (0xbfffffff)

Stack Management Instructions: pop

pop **destination**

Loads to the destination a word off the top of the stack, then it increases ESP of the operand's size.

Example

Final condition: EAX $= 30$

pop eax

is equivalent to:

```
mov eax, DWORD PTR [esp]
add esp, 4
```

Low addresses (0x80000000)

...
30
100	← ESP
...

High addresses (0xbfffffff)

Calling a function

Instruction `call`:

- Push to the stack the address of the next instruction
- Move the address of the first instruction of the callee into EIP
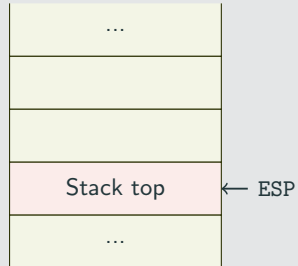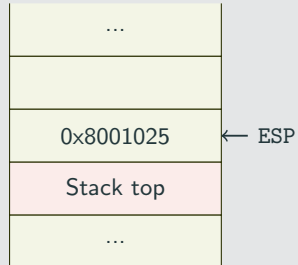
Example: Let's call `func`, located at 0x800bff00

Low addresses (0x80000000)

Equivalent to:

- push address(of the instruction after the call!)

- jmp func

(reminder: we can't read or set EIP directly!)

...
Stack top	← ESP
...

High addresses (0xbfffffff)

EIP = 0x8001020

Instruction `call`:

- Push to the stack the address of the next instruction
- Move the address of the first instruction of the callee into EIP

Example: Let's call `func`, located at 0x800bff00

Low addresses (0x80000000)

Equivalent to:

- push address(of the instruction after the call!)
- jmp func

(reminder: we can't read or set EIP directly!)

...
0x8001025	← ESP
Stack top
...

High addresses (0xbfffffff)

EIP = 0x8001020

Calling a function

Instruction `call`:

- Push to the stack the address of the next instruction
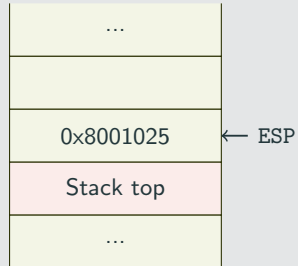- Move the address of the first instruction of the callee into EIP

Example: Let's call `func`, located at **0x800bff00**

Low addresses (0x80000000)

Equivalent to:

- push address(of the instruction after the call!)

- `jmp func`

(reminder: we can't read or set EIP directly!)

...
0x8001025   ← ESP
Stack top
...

High addresses (0xbfffffff)

EIP = 0x800bff00

Returning from a function

Instruction `ret`:

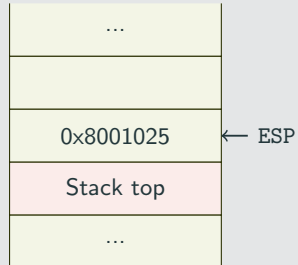- Restores the return address saved by `call` from the top of the stack

Example: let's return from `func`

Low addresses (0x80000000)

Equivalent to:

- `pop eip`

(reminde: we can't read or set EIP directly!)

...
0x8001025	← ESP
Stack top
...

High addresses (0xbfffffff)

EIP = 0x800bff00

Returning from a function

Instruction `ret`:

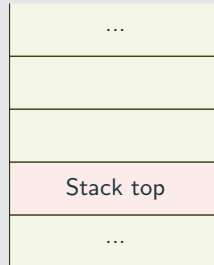- Restores the return address saved by `call` from the top of the stack

Example: let's return from `func`

Low addresses (0x80000000)

Equivalent to:

- `pop eip`

(reminde: we can't read or set EIP directly!)

...
Stack top	← ESP
...

High addresses (0xbfffffff)

EIP = 0x8001025

Functions and Stack Frames

Low addresses (0x80000000)

- Stack frame = stack area allocated to a function
- EBP register: pointer to the beginning (base) of a function's frame
- At the beginning of a function:
  - Save EBP to stack
  - Set EBP to the address of the function's frame

High addresses (0xbfffffff)

Entering a function
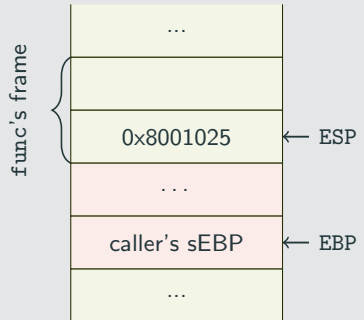
Example: We've just called func, located at 0x800bff00

Low addresses (0x80000000)

Setup the stack frame

- push ebp
- mov ebp, esp

func's frame	...
	0x8001025 ← ESP
	...
	caller's sEBP ← EBP
	...

High addresses (0xbfffffff)

Entering a function

Example: We've just called `func`, **located at 0x800bff00**

Low addresses (0x80000000)

func's frame

...
Saved EBP	← ESP
0x8001025
...
caller's sEBP	← EBP
...

High addresses (0xbfffffff)

Setup the stack frame

- `push ebp`

- `mov ebp, esp`

Example: We've just called `func`, **located at 0x800bff00**

Low addresses (0x80000000)

Setup the stack frame

- `push ebp`
- `mov ebp, esp`

func's frame
- ...
- Saved EBP ← EBP / ← ESP
- 0x8001025
- ...
- caller's sEBP
- ...

High addresses (0xbfffffff)

Leaving a function

Instruction `leave`: restores the caller's base pointer

Example: We're about to return from `func`

Low addresses (0x80000000)

Equivalent to:

- `mov esp, ebp`
- `pop ebp`

func's frame	...	
	(... locals ...)	← ESP
	Saved EBP	← EBP
	0x8001025	
	...	
	caller's sEBP	
	...	

High addresses (0xbfffffff)

Leaving a function

Instruction `leave`: restores the caller's base pointer

Example: We're about to return from `func`

Low addresses (0x80000000)

Equivalent to:

- `mov esp, ebp`
- `pop ebp`

High addresses (0xbfffffff)

Leaving a function

Instruction `leave`: restores the caller's base pointer

Example: We're about to return from `func`

Low addresses (0x80000000)

Equivalent to:

- `mov esp, ebp`
- `pop ebp`

High addresses (0xbfffffff)

Calling Conventions

- Defines
    - how to pass parameters (stack, registers or both, and who is responsible to clean them up)
    - how to return values
    - caller-saved or callee-saved registers
- The high-level language, the compiler, the OS, and the target architecture all together "implement" and "agree upon" a certain calling convention
    - it's part of the ABI, the Application Binary Interface

Calling Conventions: cdecl (C declaration)

- Default calling convention used by most x86 C compilers
  - Can be forced with the modifier `_cdecl`
- Arguments: passed **through the stack**, right to left order
- Cleanup: the **caller removes** the parameters from the stack *after* the called function completes
- Return: register `RAX`
- Caller-saved registers: `EAX`, `ECX`, `EDX` (other are callee-saved)

cdecl: **Example**

```c
void demo_cdecl(int a, int b, int c, int z);

//...

demo_cdecl(1, 2, 3, 4); //calling
```

```asm
; ...
push 4 ; push last parameter value
push 3 ; push third parameter value
push 2 ; ...
push 1
call demo_cdecl ; call the subroutine
add esp, 16 ; clean up the stack
```

Calling Conventions: `stdcall`

- Microsoft's Win32 API standard calling convention (modifier: `_stdcall`
- Parameters: passed using the stack (as in `_cdecl`)
- Main difference: the callee is responsible for clearing the function parameters from the stack before returning
  - To do this, the function needs to know the right number of parameter passed $\rightarrow$ can be used only with functions having a fixed number of parameters (e.g., no `printf`).

```
void _stdcall demo_stdcall(int a, int b, int c);
demo_stdcall(1, 2, 3);
```

```
ret 12 ; return and clear 12 bytes from the stack
```

Low addresses (0x80000000)

...
array[0]
...
array[4]
Saved EBP
Return address
1
2
...

High addresses (0xbfffffff)

```
void function (int a, int b) {
  int array [5];
}

int main (int argc, char** argv) {
  function(1, 2);
  printf("This is where the ret
      address points");
}
```

Assembler-level View of the Same Example

```
public main
main proc near
push ebp
mov ebp, esp ; create stack
and esp,0FFFFFFF0h
sub esp,10h ; make room for vars
mov dword ptr[esp+4],2 ; push 2
mov dword ptr[esp],1 ; push 1
call function
mov dword ptr [esp], offset format; "This is..."
call _printf
leave
ret
main endp

public function
function proc near
push ebp
mov ebp,esp ; create stack
sub esp,14h ; make room for array
leave
retn 8 ; remove 2 dwords
function endp
```

Low addresses (0x80000000)

...
array[0]
...
array[4]
Saved EBP
Return address
1
2
...

High addresses (0xbfffffff)

47

- Modifier: _fastcall

- Up to 2 parameters passed via registers: `ECX` and `EDX`

- Other parameters pushed to the stack (right to left order, stack cleanup by callee as in `stdcall`)

```
; demo_fastcall(1, 2, 3, 4);
push 4 ; push 4th parameter
push 3 ; push 3rd parameter
mov edx, 2 ; move 2nd parameter in EDX
mov ecx, 1 ; move 1st parameter in ECX
call demo_fastcall
; do not clean up the stack (done by callee)
```

Calling Conventions: Linux x86-64 (System V ABI)

- Parameters passed **in registers**: rdi, rsi, rdx, rcx, r8, r9, subsequent ones on the stack (reverse order, caller cleanup)
- Callee-saved registers: rbx, rsp, rbp, r12, r13, r14, and r15
- Caller-saved registers (scratch): rax, rdi, rsi, rdx, rcx, r8, r9, r10, r11
- Return value: rax (if 128-bit: rax and rdx)

Linux x86-64 calling convention: Example

```
main:
  push rbp
  mov rbp, rsp
  sub rsp, 16
  mov DWORD PTR [rbp-4], edi
  mov QWORD PTR [rbp-16], rsi
  mov esi, 2 ; Second parameter
  mov edi, 1 ; First parameter
  call function
  mov esi, eax ; Return value -> first param
  mov edi, OFFSET FLAT:.LC0 ; "The return ...
  mov eax, 0
  call printf
  leave
  ret

function:
  push rbp
  mov rbp, rsp
  mov DWORD PTR [rbp-4], edi
  mov DWORD PTR [rbp-8], esi
  mov edx, DWORD PTR [rbp-4]
  mov eax, DWORD PTR [rbp-8]
  add eax, edx
  pop rbp
  ret
```

```
int function (int a, int b) {
    return a + b;
}

int main (int argc, char** argv) {
    return printf("The return value is %d\n",
        function(1,2));
}
```

Low addresses

...
2
1
Saved RBP
Return address
...

High addresses

Tooling

Shell for Dummies [3]

important paths	/	#root (first) directory
	~/	#your home directory
	.	#current directory
	..	#parent directory
filesystem utils	pwd	#show current directory
	cd *path*	#change directory to *path*
	ls *path*	#list files in the directory at *path*
	cp *path_src path_dst*	#copy *path_src* to *path_dst, -r if copying directories*
	mv *path_src path_dst*	*#move path_src to path_dst*
basic text editor	nano *path/file*	#opens/create *file* in *path* (ctrl+x to exit)
	vim *path/file*	#opens/create *file* in *path* (i to edit; esc, :wq to save and quit)
remote actions	ssh *user@server_addr*	#ssh to *server_addr* as *user*
	scp [*u@s:*]*p_src* [*u@s:*]*p_dst*	#cp to/from remote server

[3] `cmd --help` or `cmd -h` to get the aviable options

Shell for Dummies [4]

file visualization	cat *file* less *file* hexdump -Cv *file*	#print *file* to stdout #visualize *file* better, q to quit #visualize raw bytes
redirections	command > file command >> file command < file cmd1 | cmd2	#write stdout of command to *file* #append stdout of command to *file* #use *file* as stdin of command #stdout of cmd1 as stdin cmd2
argv from command	cmd \`cmd2\` cmd $(cmd2)	#executes cmd2 first and uses the output to eval the next command
other useful commands	chmod +x *file* grep *expression* python -c 'cmd1;cmd2;...'	#give exec permission to *file* #search for *expr* in stdin #executes python commands

---

[4] `cmd --help` or `cmd -h` to get the aviable options

objdump

- `man objdump`
  objdump **displays information** about one or more **object files**.
- `-x` all-headers
- `-d` disassemble
- `-M intel` intel syntax (default is AT&T)

Debugging: GDB

- **What is GDB?**
  GDB is GNU Project's Debugger: allows to follow, step by step, at assembler-level granularity, a running program, or what a program was doing right before it crashed.[5]

---

[5]http://www.gnu.org/software/gdb/

Start, break and navigate the execution with gdb

- Suppose you have an executable binary and want run it
  - **gdb /path/to/executable** loads the binary in gdb
- Now you decide to start the program with two parameters
  - **run 1 "abc"** passes 1 via argv[1] and "abc" as argv[2]
  - **run 'printf "AAAAAAAAAAAA"'** (with the back ticks)
    we're passing the output of the print (very useful when you
    need to pass non printable characters such as raw bytes)
- Suppose you want to stop the execution at the address of a
  certain instruction
  - **break *0xDEADBEAF** places a break point at that address
  - **break *main+1** with debugging symbols this can be less
    painful
  - **catch syscall** block the execution when a syscall happens

Start, break and navigate the execution with gdb

- Now the execution stops at our break point. Here we can do several things
- Examples:
  - **ni** allows to procede instruction per instruction
  - **next 4** moves 4 lines ahead (if you have the line-numbers information in the binary)
  - **si** step into function
  - **finish** run until the end of current function
  - **continue** runs until the next break point (if any)
- To see info about the execution state:
  - **info registers** to inspect the content of the registers
  - **info frame** to see the values of the stack frame related to the function where we are in
  - **info file** print the information about the sections of the binary

Navigate the stack

- Suppose we're stopped somewhere in the code and want to inspect the stack
- Some useful view of the stack is achievable with:
    - **x/100wx $esp** prints 100 words of memory from the address found in the ESP to ESP+100 (x = hexadecimal formatting)
    - **x/10wo $ebp-100** prints 10 words of memory from EBP-100 to EBP-100+10 (o = octal formatting)
    - **x/s $eax** prints the elements pointed by EAX (s = string formatting)
- Do you have debug symbols? (i.e., gcc -ggdb)
    - **print args** prints info about the main parameters
    - **print a** prints the content of variable 'a'
    - **print *b** prints the value pointed by 'b'

Our friend gdb

- **The '~/.gdbinit' file**
  Gdb is a command line tool and it supports the configuration
  script as almost all the *nix software.
  Some options that you may want to tune are:
  - **set history save on**
    To have the lastest commands always available also when we
    re-open gdb
  - **set follow-fork-mode child**
    Allows you, if the process spawns children, to follow them and
    not only wait their end.
  - **set disassembly-flavor [intel | att]**
    This option sets in which predefined syntax your disassembled
    will be showed up. The default one is at&t
- Highly recommended to install pwndbg
  https://github.com/pwndbg/pwndbg

GDB – How to Survive [6]

start	gdb -q *program*	#starts gdb silently for program
disassemble	set disassembly-flavor intel	#sets intel syntax
	disass **address* (or *f-name*)	#disassemble from given *address*
run program	run (r)	#runs the program
	start	#runs the program and imm. stops
	run *arg1*	#runs program with *arg1* in argv
	run <<< *arg1*	#runs program with *arg1* in stdin
memory layout	vmmap	#show memory layout

---

[6]CTRL + C to Break and Debug

execution	stepi (si) nexti (ni) finish (f) continue (c)	#exec next inst - enters a function #exec next inst - skips the function #exec till next return statement #continue exec till next break/ watch
breakpoinits	b *address* b *address* if $reg==val del *br_num*	#set software breakp at *address* #set conditional breakp #remove breakpoint *br_num*
watchpoints	w *address* rw *address*	#set watch for write at *address* #set watch for read at *address*
examine	x/*numF* *address* search *string* p *symbol*	#show *num* data of type *F* (useful *F*s are bx, wx, gx, c, s, i) #search for string in memory #print address of symbol

---

[7]CTRL + D to Exit

strace

- Intercepts and records system calls and signals
- Dumps to standard error name, argument and return value of each system call

Useful options

- -p <pid> attach to existing process
- -f trace child process
- -o <filename> output to file
- -e <expr> modifies which events to trace (see manpage)

- Intercepts and records dynamic library calls
- Similar to strace, but at a different layer

Questions?