

x86 Crash Course

With a focus on Linux and a glance to x86_64

Mario Polino, Lorenzo Binosi, Alessandro Bertani

March 21, 2023

DEIB, Politecnico di Milano

How to Read the Matrix!

LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

```
EAX 0xffffffff00
EDX 0x0
ECX 0x565558570 ← 0x0
EDX 0x400
EDI 0xf7fba000 ← insb byte ptr es:[edi], dx /* 0x1d4d6c */
ESI 0xf7f7be5c0 ( _IO_2_1_stdin_ ) ← mov byte ptr [edx], ah /* 0xfbad2288 */
EBP 0xffffffff00 → 0xffffffff438 → 0xffffffff478 → 0xffffffff4e8 ← 0x0
ESP 0xffffffff00 → 0xffffffffcde8 → 0xffffffff438 → 0xffffffff478 → 0xffffffff4e8 ← ...
EIP 0xf7fd5079 ( _kernel_vsycall+9 ) ← pop ebp
```

[REGISTERS]

```
► 0xf7fd5079 < _kernel_vsycall+9 > pop ebp
    ↓
0xf7fd507b < _kernel_vsycall+11 > pop ecx
0xf7fd507c < _kernel_vsycall+12 > ret
    ↓
0xf7ece6e7 <read+39> cmp eax, 0xffffffff00
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 0xffffffffcde8 → 0xffffffff438 → 0xffffffff478 → 0xffffffff4e8 ← ...
01:0004 0xffffffffc84 ← 0x400
02:0008 0xffffffffc88 → 0x565558570 ← 0x0
03:000c 0xffffffffc8c → 0xf7ece6e7 (read+39) → 0xffff0003d ← 0xffff0003d
04:0010 0xffffffffc90 ← 0x0
    ↓
06:0018 0xffffffffc98 → 0xf7e5bb29 ( _IO_file_overflow+9 ) ← add edx, 0x1624d7
07:001c 0xffffffffc9c → 0xf7fbc860 ( _IO_file_jumps ) ← add byte ptr [eax], al
```

[BACKTRACE]

```
► f 0 f7fd5079 _kernel_vsycall+9
f 1 f7ece6e7 read+39
f 2 f7e5b9b8 _IO_file_underflow+328
f 3 f7e5cadb _IO_default_uflow+59
f 4 f7e405e9 _io_vscanf+3945
f 5 f7e4c4fd __isoc99_scanf+125
f 6 56555652 main+85
f 7 f7e01e91 __libc_start_main+241
```

The x86 Architecture

- Overview on the common 32-bit Intel Architecture (IA)

- Basic Instructions

- x86_64

Program Layout and Functions

- Memory Layout

- The Stack

- Functions

- Calling Conventions

Tooling

The x86 Architecture

Instruction Set Architecture (ISA)

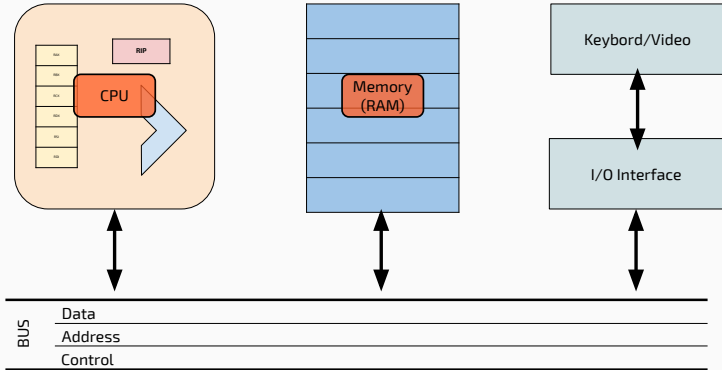
- “Logical” specification of a computer architecture
- Concerned with programming concepts
 - instructions, registers, interrupts, memory architecture, ...
- May differ (widely) from the actual microarchitecture
- Examples:
 - x86 (IA-32 and x86_64)
 - ARM (mobile devices)
 - MIPS (embedded devices,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

The x86 ISA

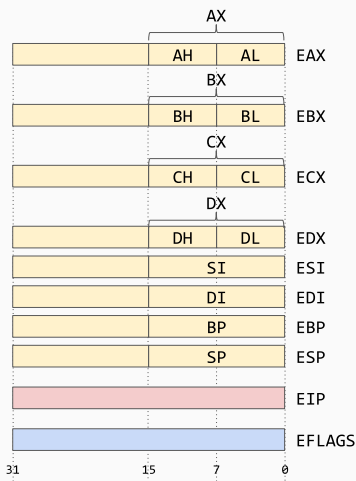
- Born in 1978, 16-bit ISA (Intel 8086)
- Evolved to a 32-bit ISA (1985, Intel 80386)
- Evolved to a 64-bit ISA (2003, AMD Opteron)
- CISC design (e.g., string operations)
- Many legacy features (e.g, segmentation)
- We'll see the basics of the “core” ISA
 - There is also the floating point unit, processor-specific features, and extensions such as SIMD (MMX, SSE, SSE2) with their own instructions and registers¹

¹Complete 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

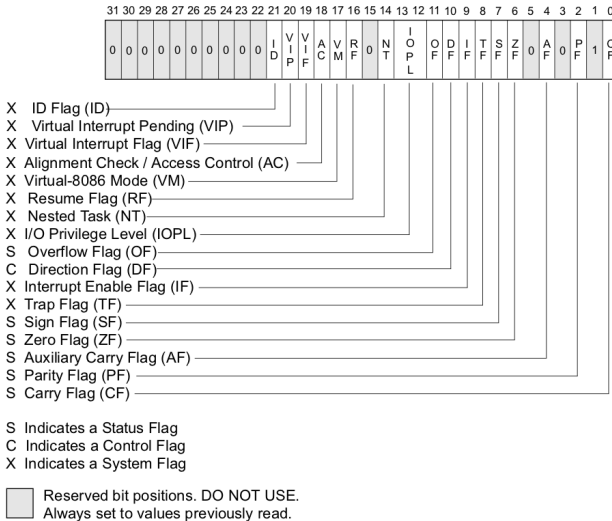


- General-purpose registers
 - EAX, EBX, ECX, EDX
 - ESI, EDI (source and destination index for string operations)
 - EBP (base pointer)
 - ESP (stack pointer)
- Instruction pointer: EIP
 - No explicit access
 - Modified by `jmp`, `call`, `ret`
 - Read through the stack (saved IP)
- Program status and control: EFLAGS
- (segment registers)

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

IA-32: EFLAGS register



x86 Fundamental data types

byte 8 bits

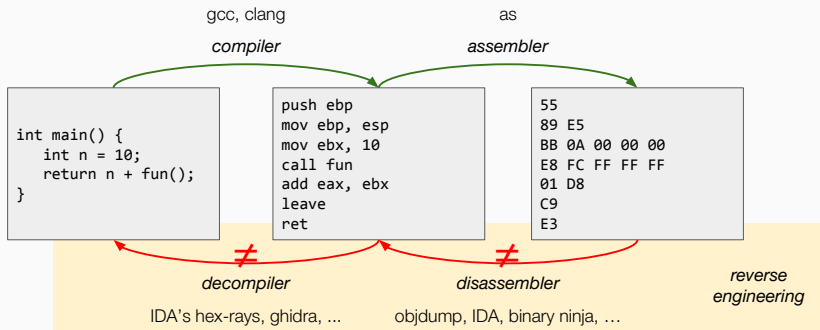
word 2 bytes

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.

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

`mov eax, 0h`

AT&T

`movl $0x0,%eax`

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

Intel

`mov [ebx+4h],0h`

AT&T

`movl $0x0,0x4(%ebx)`

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]` $EAX = *(EBX + 4)$

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

Note: memory to memory is an **invalid** combination²

²Except 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

<code>MOV eax, ebx</code>	<code>MOV eax, FFFFFFFFh</code>	<code>MOV ax, bx</code>
<code>MOV [eax],ecx</code>	<code>MOV [eax],[ecx] NO!!!</code>	<code>MOV al, FFh</code>

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

lea eax, [ebx + 8] → EAX = 0x00403A48

N.B.: with mov eax, [ebx+8] → EAX = 0x0012C140

Integer Arithmetics: add and sub

add <u>destination</u> , <u>source</u>	sub <u>destination</u> , <u>source</u>
$\text{dest} \leftarrow \text{dest} + \text{source}$	$\text{dest} \leftarrow \text{dest} - \text{source}$

- 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
- $\text{dest} \leftarrow \text{implied_op} \times \text{source}$
- **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`

<code>cmp <u>op1</u>, <u>op2</u></code>	<code>test <u>op1</u>, <u>op2</u></code>
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: conditional jumps

j<cc> address or offset

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

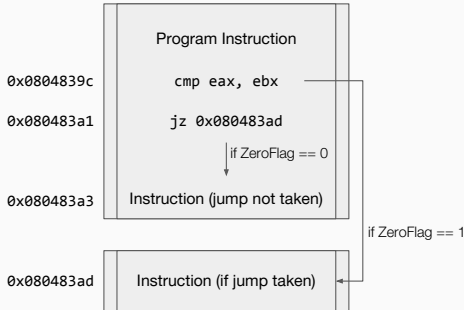
<cc>: condition

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

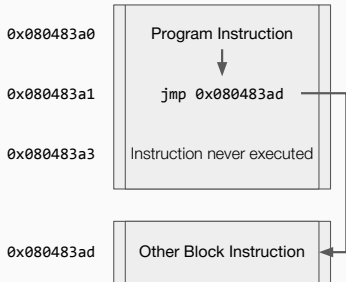
Examples:

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

Reference: <http://www.unixwiz.net/techtips/x86-jumps.html>



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

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:
    ...
```

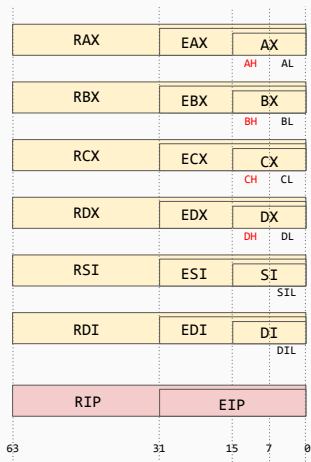
Basic Instructions: `nop`

- `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

The x86_64 ISA

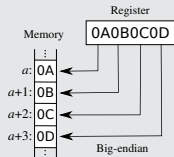


Endianness

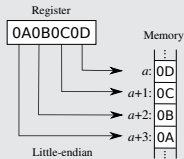
Endianness: convention that specifies in which order the bytes of a data word are lined up sequentially in memory.

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

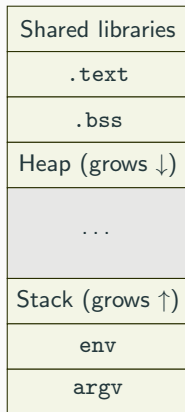
- **PE (Portable Executable):** used by Microsoft binary executables
- **ELF:** common binary format for Unix, Linux, FreeBSD and others
- In both cases, we are interested in how each executable is mapped into memory, rather than how it is organized on disk.

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

Executable	Description
.plt	This section holds stubs which are responsible of external functions linking.
.text	This section holds the "text," or executable instructions, of a program.
.rodata	This section holds read-only data that contribute to the program's memory image
.data	This section holds initialized data that contribute to the program's memory image
.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.
.debug	This section holds information symbolic debugging.
.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.

Simplified program memory layout

Low addresses (0x80000000)



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`

`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

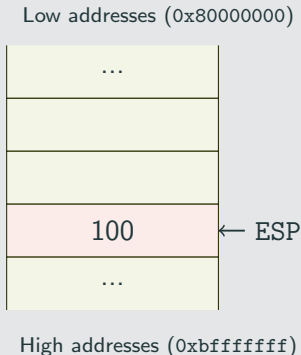
Initial condition: `EAX = 30`

`push eax`

is equivalent to:

`sub esp, 4`

`mov DWORD PTR [esp], eax`



Stack Management Instructions: `push`

`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

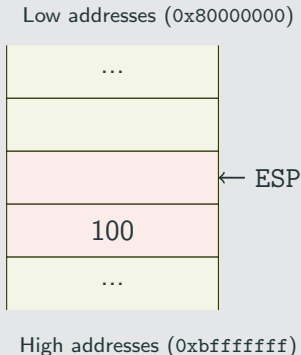
Initial condition: `EAX = 30`

`push eax`

is equivalent to:

`sub esp, 4`

`mov DWORD PTR [esp], eax`



Stack Management Instructions: `push`

`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

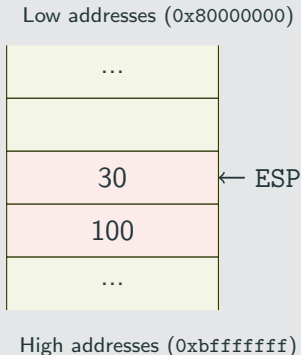
Final condition: `EAX = 30`

`push eax`

is equivalent to:

`sub esp, 4`

`mov DWORD PTR [esp], eax`



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



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 = 30

pop eax

is equivalent to:

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

```
add esp, 4
```



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

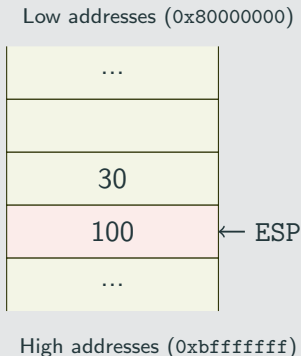
Final condition: EAX = 30

pop eax

is equivalent to:

mov eax, DWORD PTR [esp]

add esp, 4



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

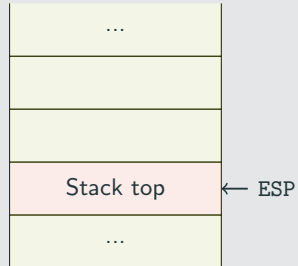
Example: Let's call `func`, located at `0x800bff00`

Equivalent to:

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

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

Low addresses (`0x80000000`)



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

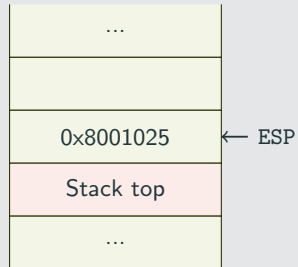
Example: Let's call `func`, located at `0x800bff00`

Equivalent to:

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

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

Low addresses (`0x80000000`)



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

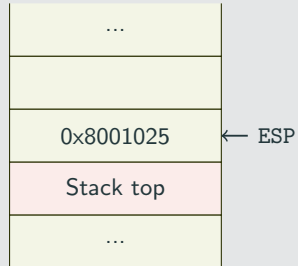
Example: Let's call `func`, located at `0x800bff00`

Equivalent to:

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

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

Low addresses (`0x80000000`)



High addresses (`0xbfffffff`)

EIP = `0x800bff00`

Returning from a function

Instruction `ret`:

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

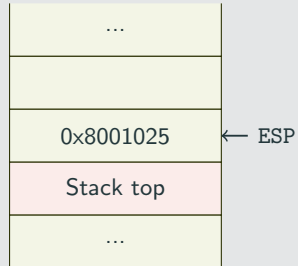
Example: let's return from `func`

Equivalent to:

- `pop eip`

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

Low addresses (0x80000000)



High addresses (0xbfffffff)

EIP = 0x800bff00

Returning from a function

Instruction `ret`:

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

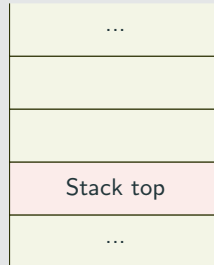
Example: let's return from `func`

Equivalent to:

- `pop eip`

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

Low addresses (0x80000000)



← ESP

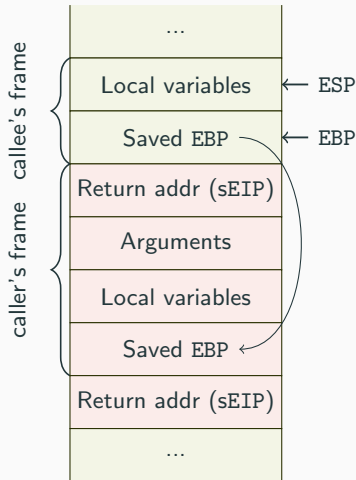
High addresses (0xbfffffff)

EIP = 0x8001025

Functions and Stack Frames

- 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

Low addresses (0x80000000)



High addresses (0xbfffffff)

aasdasd

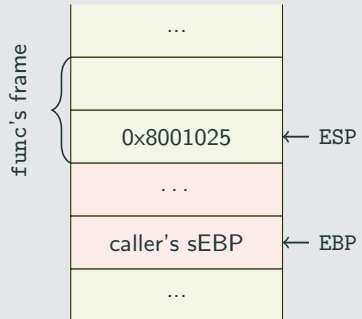
Entering a function

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

Setup the stack frame

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

Low addresses (`0x80000000`)



High addresses (`0xbfffffff`)

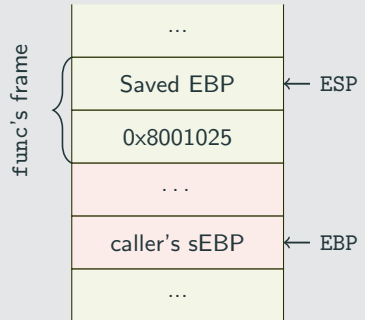
Entering a function

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

Setup the stack frame

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

Low addresses (`0x80000000`)



High addresses (`0xbfffffff`)

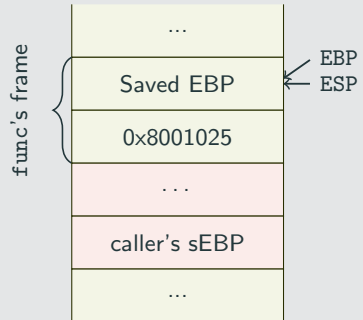
Entering a function

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

Setup the stack frame

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

Low addresses (`0x80000000`)



High addresses (`0xbfffffff`)

Leaving a function

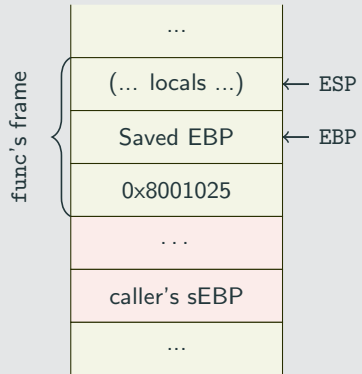
Instruction `leave`: restores the caller's base pointer

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

Equivalent to:

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

Low addresses (0x80000000)



High addresses (0xbfffffff)

Leaving a function

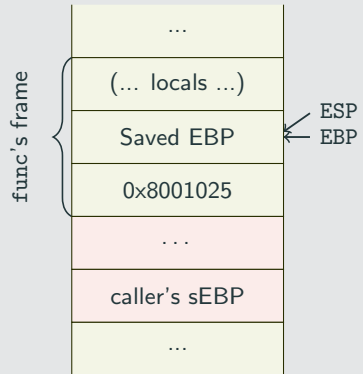
Instruction `leave`: restores the caller's base pointer

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

Equivalent to:

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

Low addresses (0x80000000)



High addresses (0xbfffffff)

Leaving a function

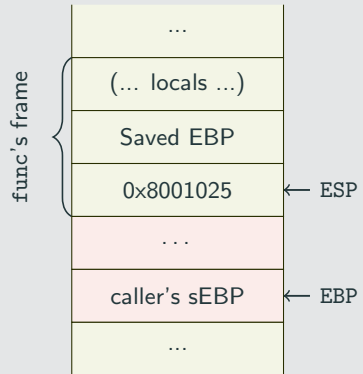
Instruction `leave`: restores the caller's base pointer

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

Equivalent to:

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

Low addresses (0x80000000)



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 EAX
- Caller-saved registers: EAX, ECX, EDX (other are callee-saved)

cdecl: Example

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

```
//...
```

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

```
; ...
```

```
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: `fastcall`

- 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)

Tooling

Shell for Dummies³

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

³cmd --help or cmd -h to get the available options

Shell for Dummies ⁴

file visualization	cat <i>file</i> less <i>file</i> hexdump -Cv <i>file</i>	#print <i>file</i> to stdout #visualize <i>file</i> better, q to quit #visualize raw bytes
redirections	command > <i>file</i> command >> <i>file</i> command < <i>file</i> cmd1 cmd2	#write stdout of command to <i>file</i> #append stdout of command to <i>file</i> #use <i>file</i> 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 <i>file</i> grep <i>expression</i> python -c 'cmd1;cmd2;...'	#give exec permission to <i>file</i> #search for <i>expr</i> in stdin #executes python commands

⁴cmd --help or cmd -h to get the available options

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

- **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 "AAAAAAAAAAAAAA"'`** (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 proceed 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 latest 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`
`disass *address (or f-name)`

#sets intel syntax
#disassemble from given address

run program

`run (r)`
`start`
`run arg1`
`run <<< arg1`

#runs the program
#runs the program and imm. stops
#runs program with *arg1* in argv
#runs program with *arg1* in stdin

memory layout

`vmmap`

#show memory layout

⁶CTRL + C to Break and Debug

GDB - How to Survive ⁷

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
breakpoints	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 Fs are bx, wx, gx, c, s, i) #search for string in memory #print address of symbol

⁷CTRL + D to Exit

- 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?