Malware Analysis

Advanced Analysis

By Z-Lab team

ISWATLab

The Reverse Engineering chain

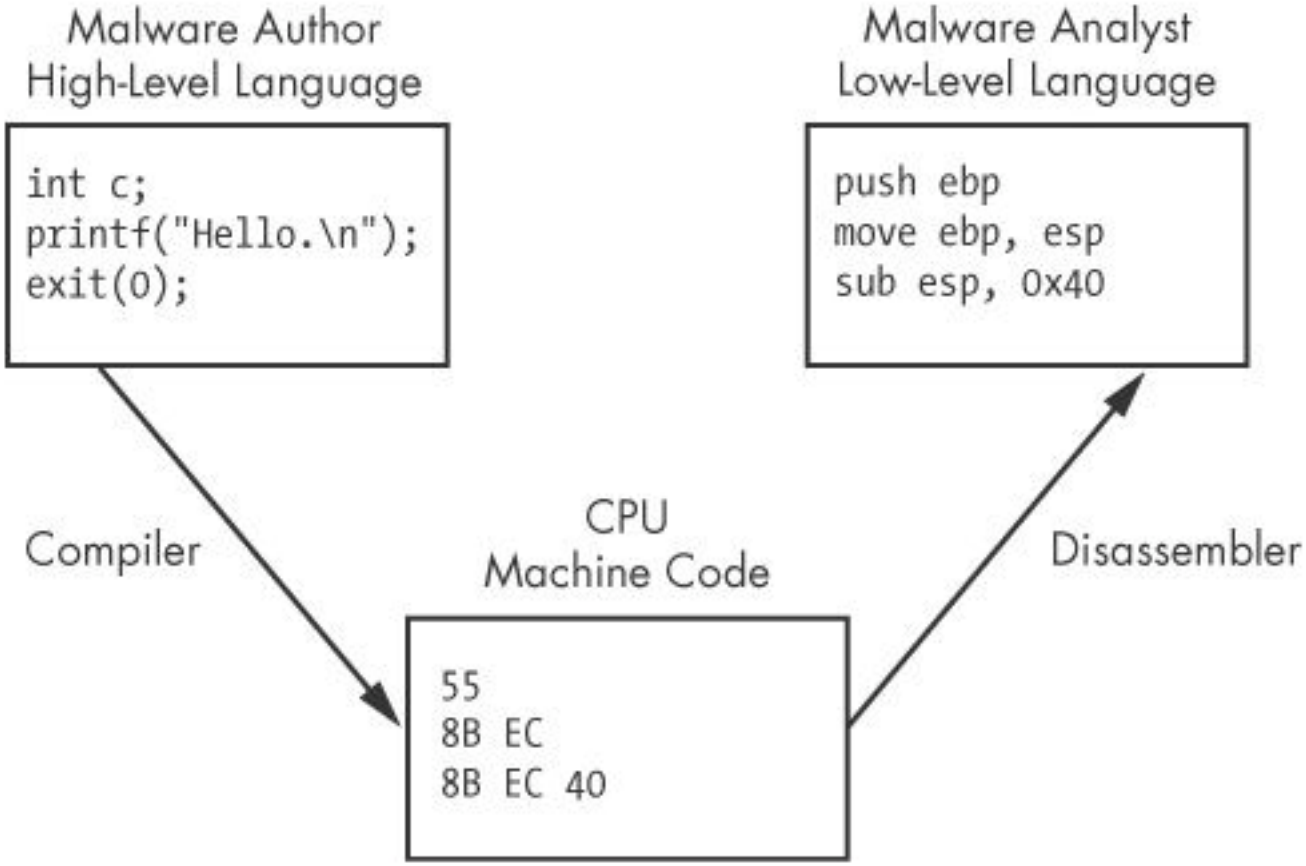


Figure 5-1. Code level examples

Six Levels of Abstraction

1. Hardware
2. Microcode
3. Machine code
4. Low-level languages
5. High-level languages
6. Interpreted languages

Six Levels of Abstraction

1. Hardware
   - Digital circuits
   - XOR, AND, OR, NOT gates
   - Cannot be easily manipulated by software
2. Microcode
   - Also called **firmware**
   - Only operates on specific hardware it was designed for
   - Not usually important for malware analysis

Six Levels of Abstraction

3. Machine code
   - **Opcodes**
     - Tell the processor to do something
     - Created when a program written in a high-level language is compiled

```
792415C0      55            push ebp
792415C1      89E5          mov ebp, esp
792415C3      8B45 08       mov eax, [ebp+0x08]
792415C6      DB28          fld tword [eax]
792415C8      8B4D 0C       mov ecx, [ebp+0x0C]
792415CB      DB29          fld tword [ecx]
792415CD      DEC1          faddp
792415CF      8B55 10       mov edx, [ebp+0x10]
792415D2      DB3A          fstp tword [edx]
792415D4      DB68 0A       fld tword [eax+0x0A]
792415D7      DB69 0A       fld tword [ecx+0x0A]
792415DA      DEC1          faddp
792415DC      DB7A 0A       fstp tword [edx+0x0A]
792415DF      5D            pop ebp
792415E0      C2 0C00       ret 0x000C
```

Six Levels of Abstraction

4. Low-level languages
   - Human-readable version of processor's instruction set
   - Assembly language
     - PUSH, POP, NOP, MOV, JMP ...
   - Disassembler generates assembly language
   - This is the highest level language that can be reliably recovered from malware when source code is unavailable

Six Levels of Abstraction

5. High-level languages
   - Most programmers use these
   - C, C++, etc.
   - Converted to machine code by a **compiler**

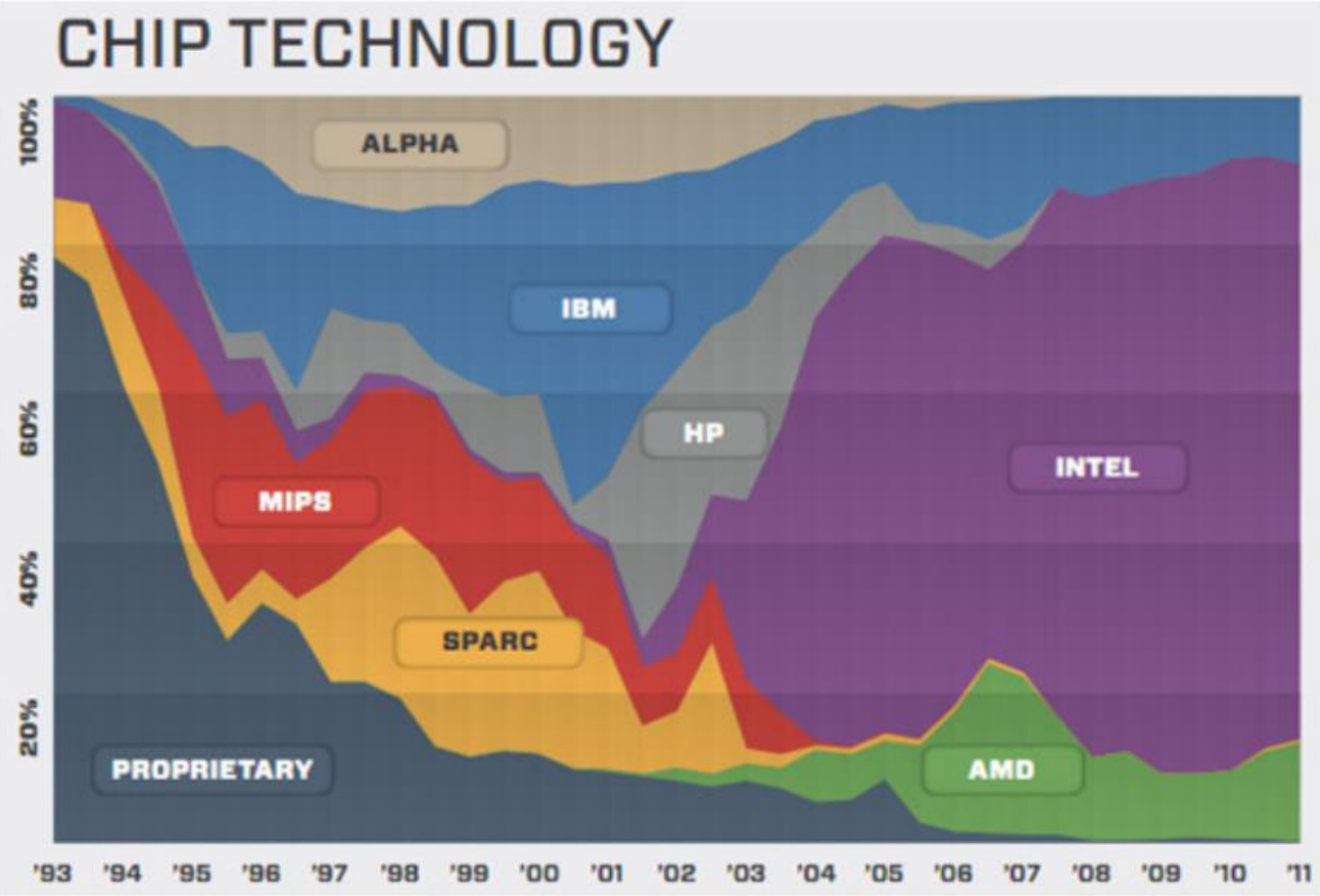
```
#include <stdio.h>

int main() {
        printf("Hello, World\n");
        return 0;
}
```

Six Levels of Abstraction

6. Interpreted languages
   - Highest level
   - Java, C#, Perl, .NET, Python
   - Code is not compiled into machine code
   - It is translated into **bytecode**
     - An intermediate representation
     - Independent of hardware and OS
     - Bytecode executes in an **interpreter**, which translates bytecode into machine language on the fly at runtime
     - Ex: Java Virtual Machine

X86 Architecture

- X86 (x64) is the dominant architecture today

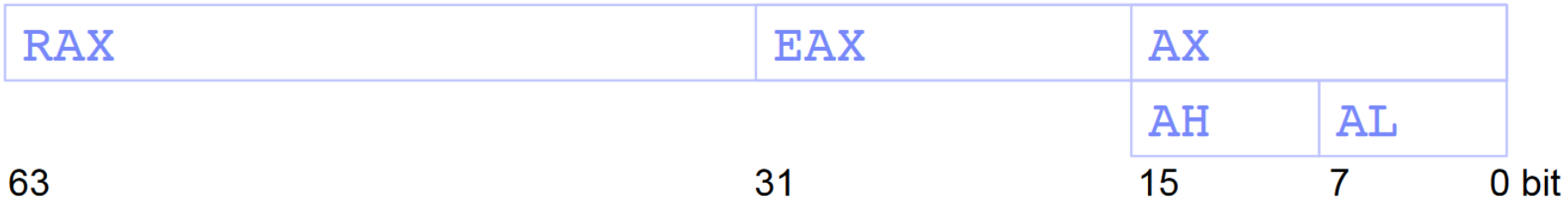


X86 Architecture

- CISC vs RISC
  - CISC :
    - Eased programming effort in the early days
    - Many, possibly complex (larger) instructions
    - Larger instructions take more time to decode and execute
    - Today, complex instructions often implemented as microcode
  - RISC:
    - Small, simple instructions
    - Popular in 90’s workstations
    - Require more instructions
    - Can run at higher clock speed due to simplicity
    - Reflected in microcode approach of new x86 CISC chips

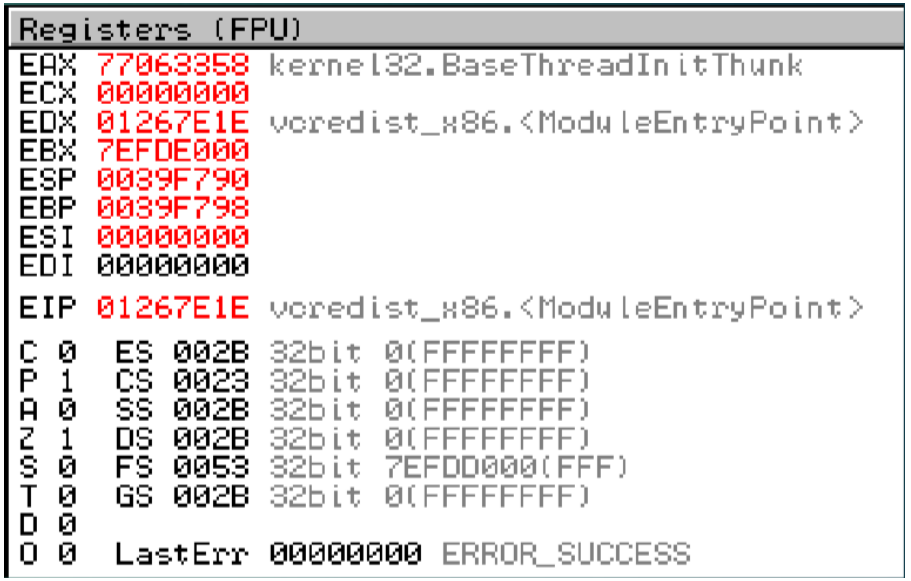
X86 Registers

- Overview – 16-bit integer registers
  - “General” purpose (with exceptions): AX, BX, CX, DX
  - Pointer registers: SP (Stack pointer), BP (Base Pointer)
  - For array indexing: DI, SI
  - FLAGS register to store flags, e.g. CF, OF, ZF
  - Instruction Pointer: IP
- Larger Registers (64 & 32bit) comprise the smaller ones lower half
  - 32 bit prefixed with “e”, 64 bit with “r”



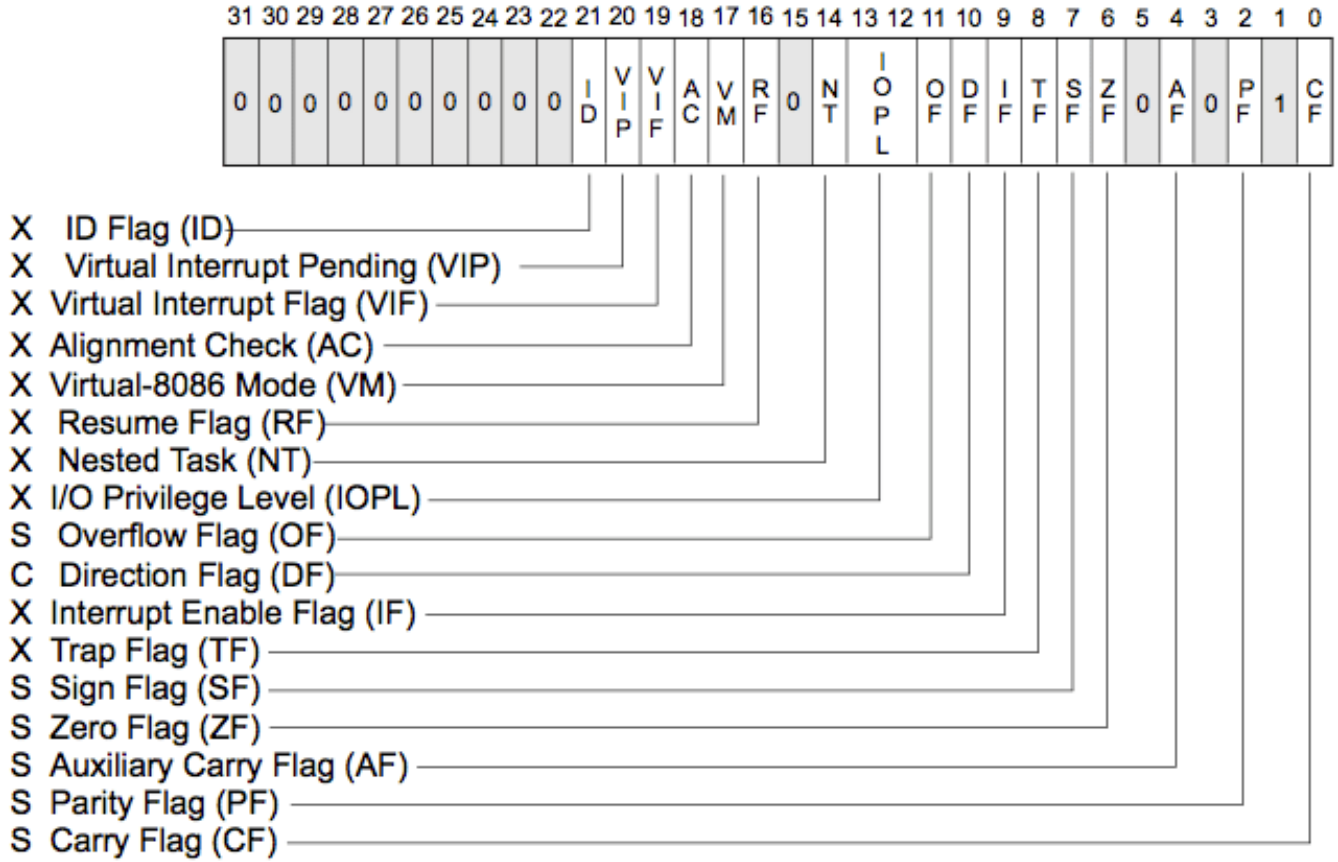
X86 Registers

- Special Purpose Registers
  - EIP – Instruction Pointer
  - EFLAGS - Flags
  - ESP – Stack Pointer
  - EBP – Base Pointer
- General Purpose Registers
  - EAX - Accumulator Register
  - EBX - Base Register
  - ECX - Counter Register
  - EDX - Data Register
  - ESI - Source Index
  - EDI - Destination Index
  - EBP - Base Pointer
  - ESP - Stack Pointer



OllyDbg Register Window

X86 EFLAGS Register




```
1   mov eax, 1
2   cmp eax, 2
3   jz  istwo
4   isnot:
5   mov eax, 2
6   istwo:
7   mov eax, 0
```

X86 EFLAGS Register

- **ZF** Zero flag
  - Set when the result of an operation is zero
- **CF** Carry flag
  - Set when result is too large or small for destination
- **SF** Sign Flag
  - Set when result is negative, or when most significant bit is set after arithmetic
- **TF** Trap Flag
  - Used for debugging—if set, processor executes only one instruction at a time

X86 EIP Register

- EIP = Extended Instruction Pointer

- Contains the memory address of the next instruction to be executed
- If EIP contains wrong data, the CPU will fetch non-legitimate instructions and crash
- Buffer overflows target EIP

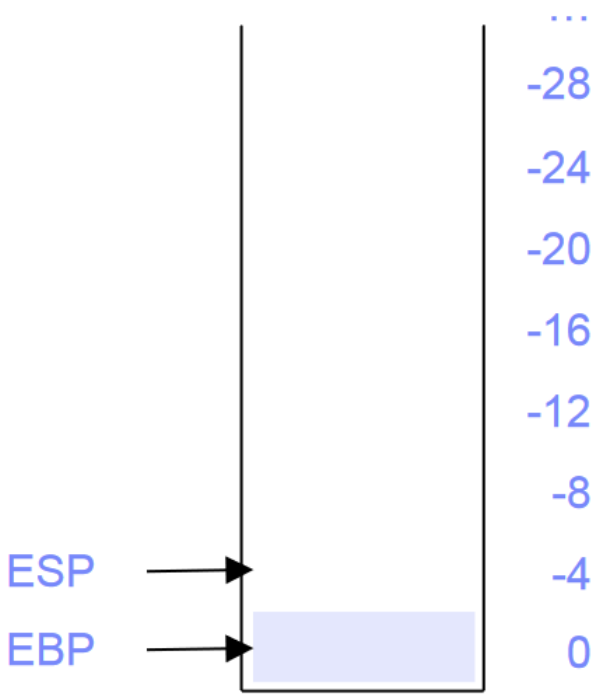
X86 Instructions & Addressing

- X86 instructions range from 1 to 15 bytes

- X86 has 8-bit addressability
  - For larger word size (most), address only specifies the low-order byte
  - The word size depends on the context (mode/instruction)
  - Addresses must align with word size
  - Addressable memory:
    - 8086 20-bit address space (1MB)
    - 80286 24-bit address space (16MB)
    - 80386 32-bit address space (4GB)
    - AMD Athlon/ Intel Core 2 64-bit address space, 48 bit implemented (256TB)

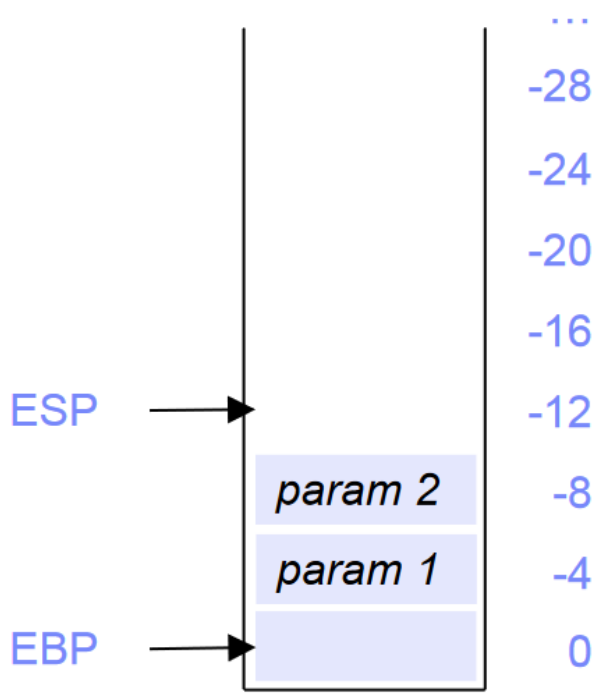
X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer



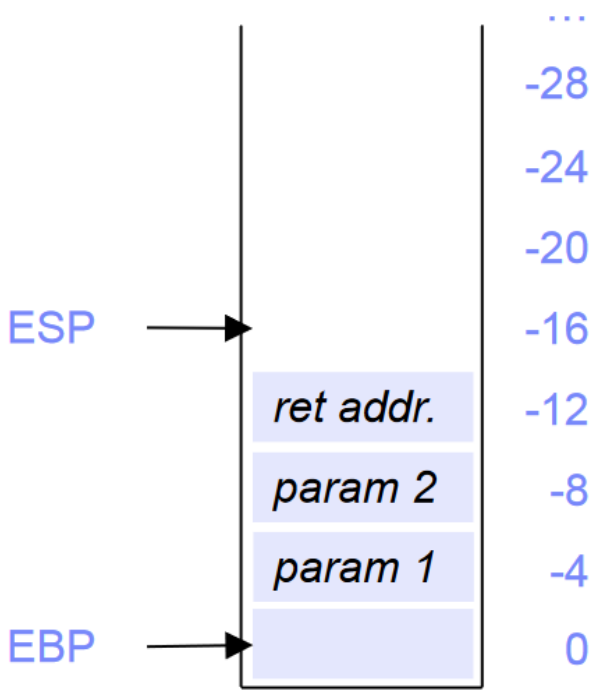
X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer
- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack



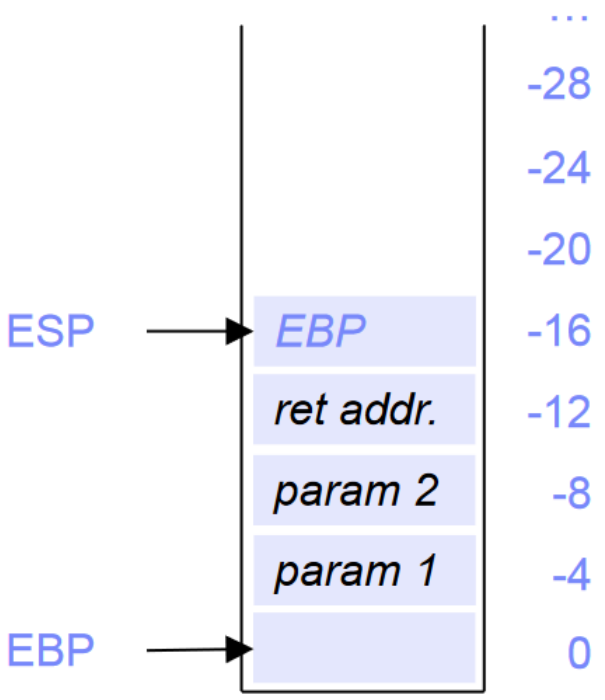
X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer
- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address



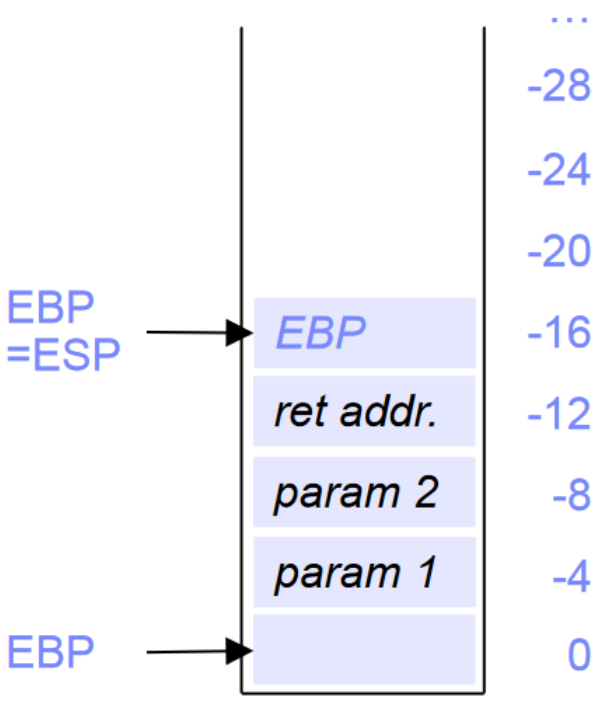
X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer
- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP



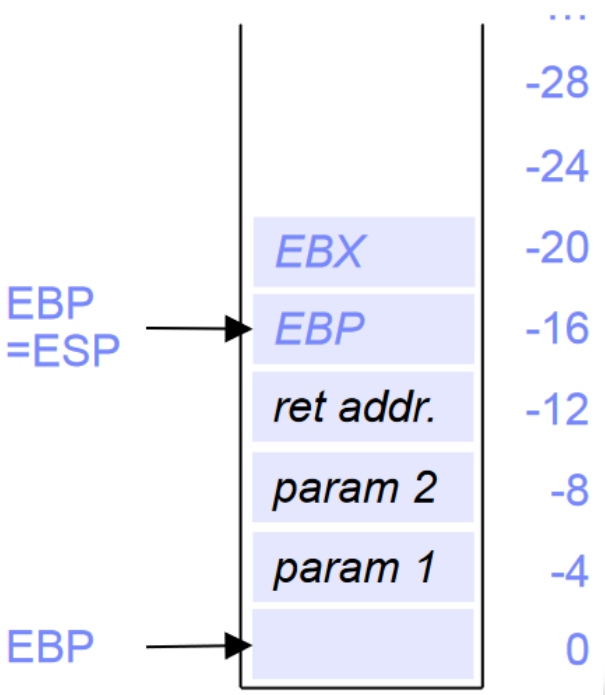
X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer
- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP



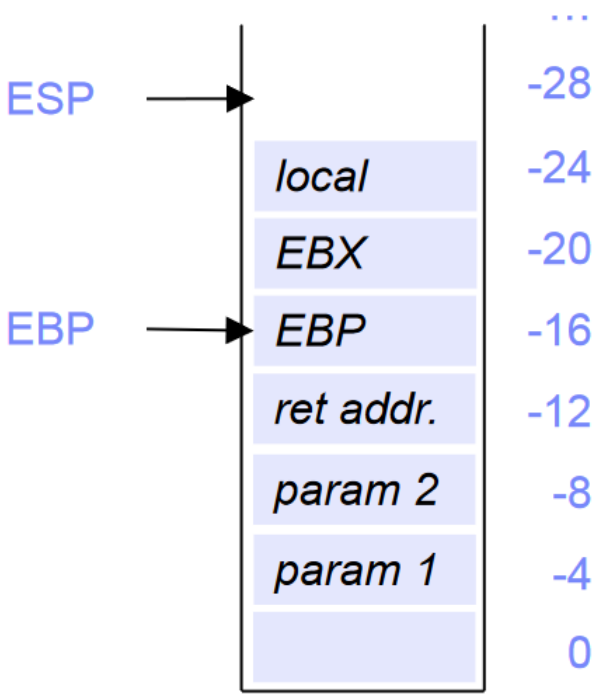
X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer
- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them



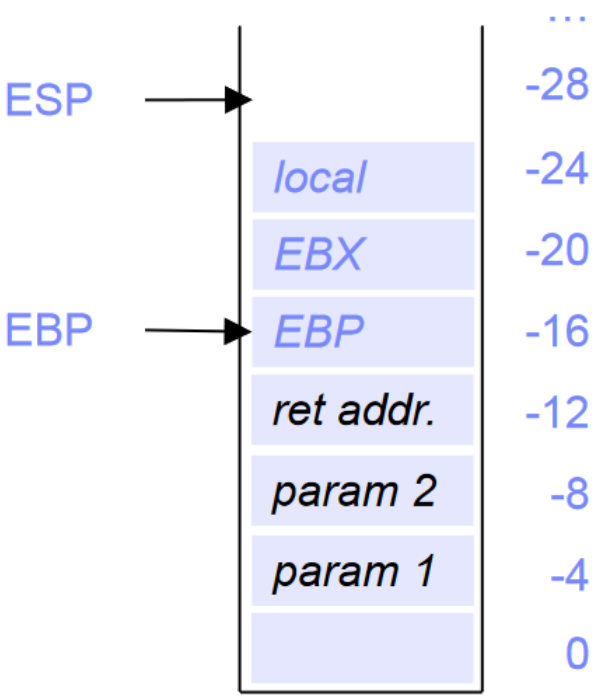
X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer
- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)



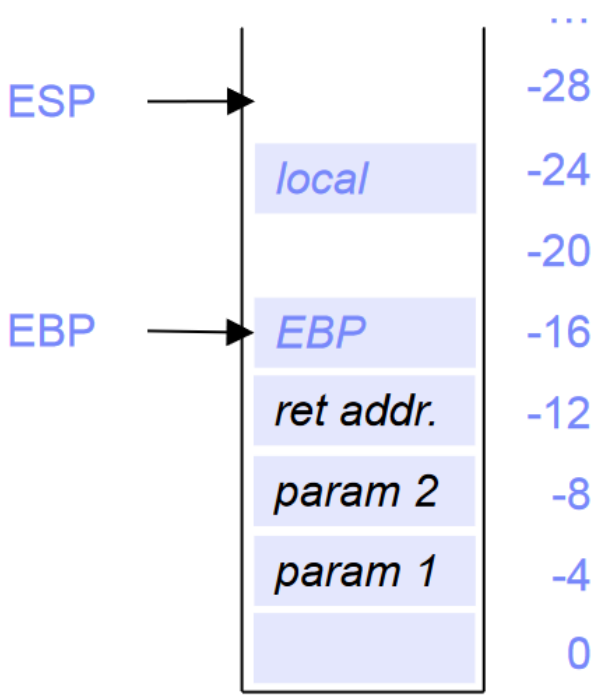
X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer
- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
  - At the end of a program/function



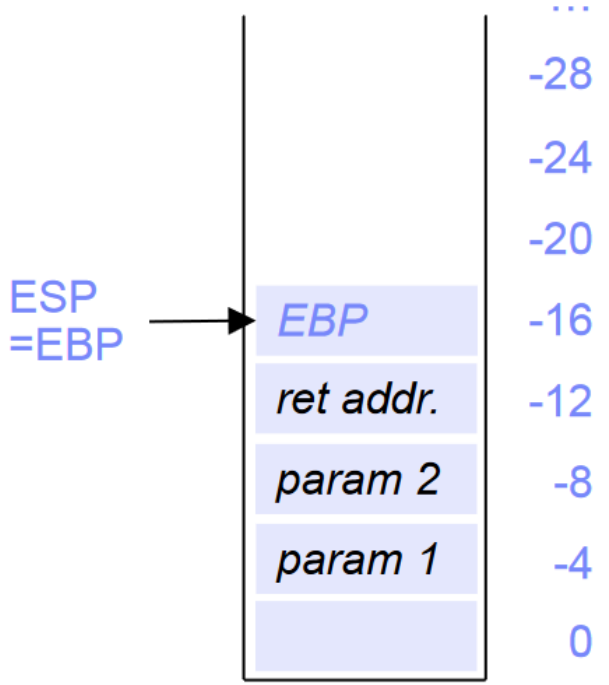
X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer
- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
  - At the end of a program/function
    - All registers are restored



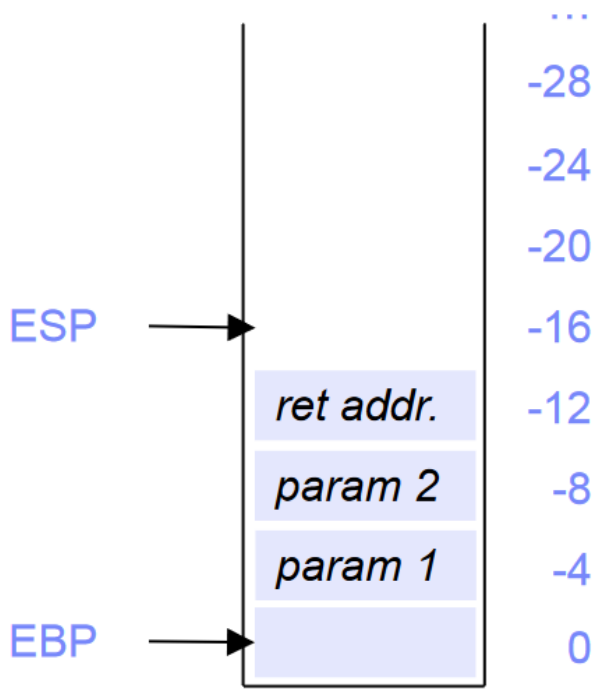
X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer
- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
  - At the end of a program/function
    - All registers are restored
    - LEAVE instruction clears the stack, i.e. sets ESP to EBP and pops EBP



X86 Instructions & Addressing

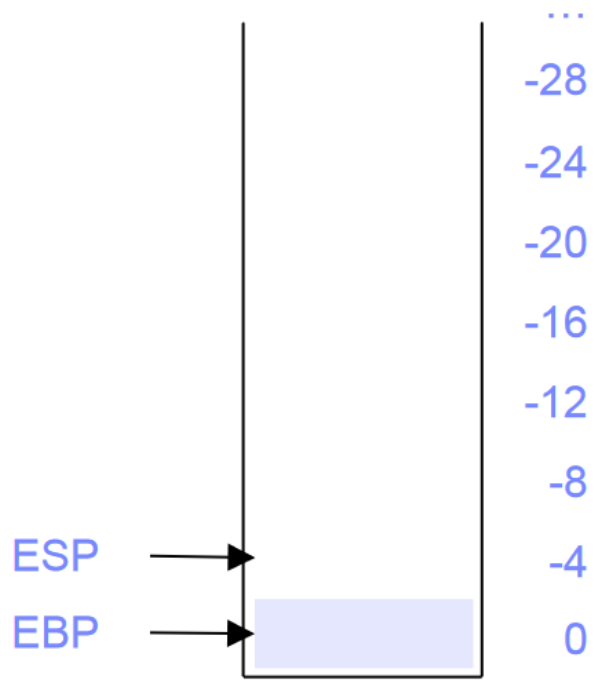
- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer
- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
  - At the end of a program/function
    - All registers are restored
    - LEAVE instruction clears the stack, i.e. sets ESP to EBP and pops EBP



X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

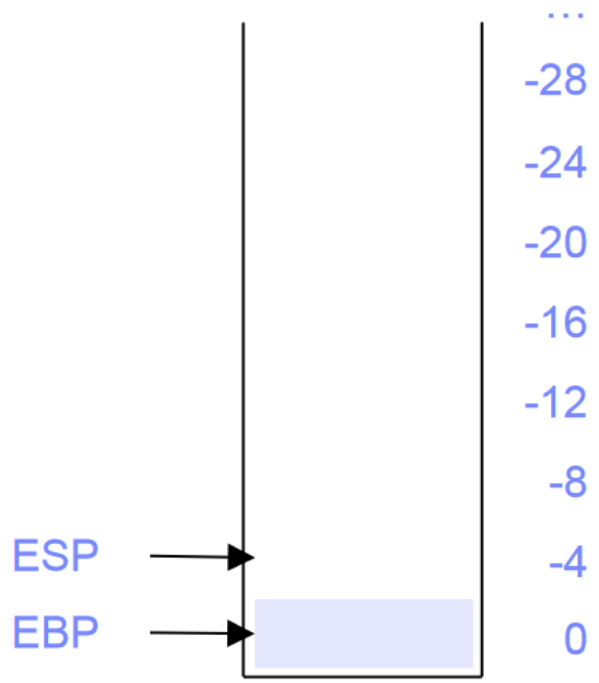
- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
  - At the end of a program/function
    - All registers are restored
    - LEAVE instruction clears the stack, i.e. sets ESP to EBP and pops EBP
    - RET instruction restores state in calling function …



X86 Instructions & Addressing

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
  - At the end of a program/function
    - All registers are restored
    - LEAVE instruction clears the stack, i.e. sets ESP to EBP and pops EBP
    - RET instruction restores state in calling function …



The Reverse Engineering Chain

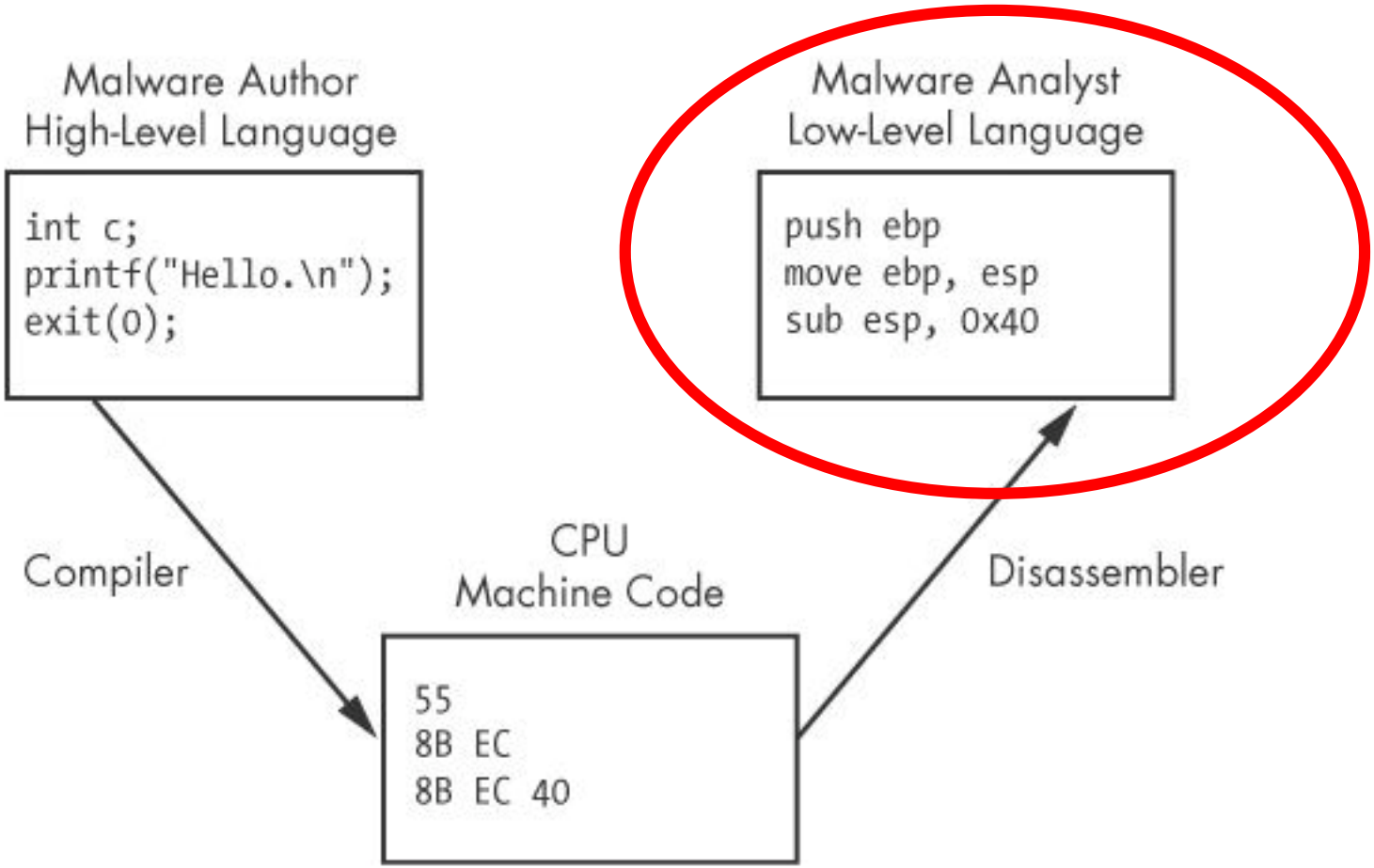
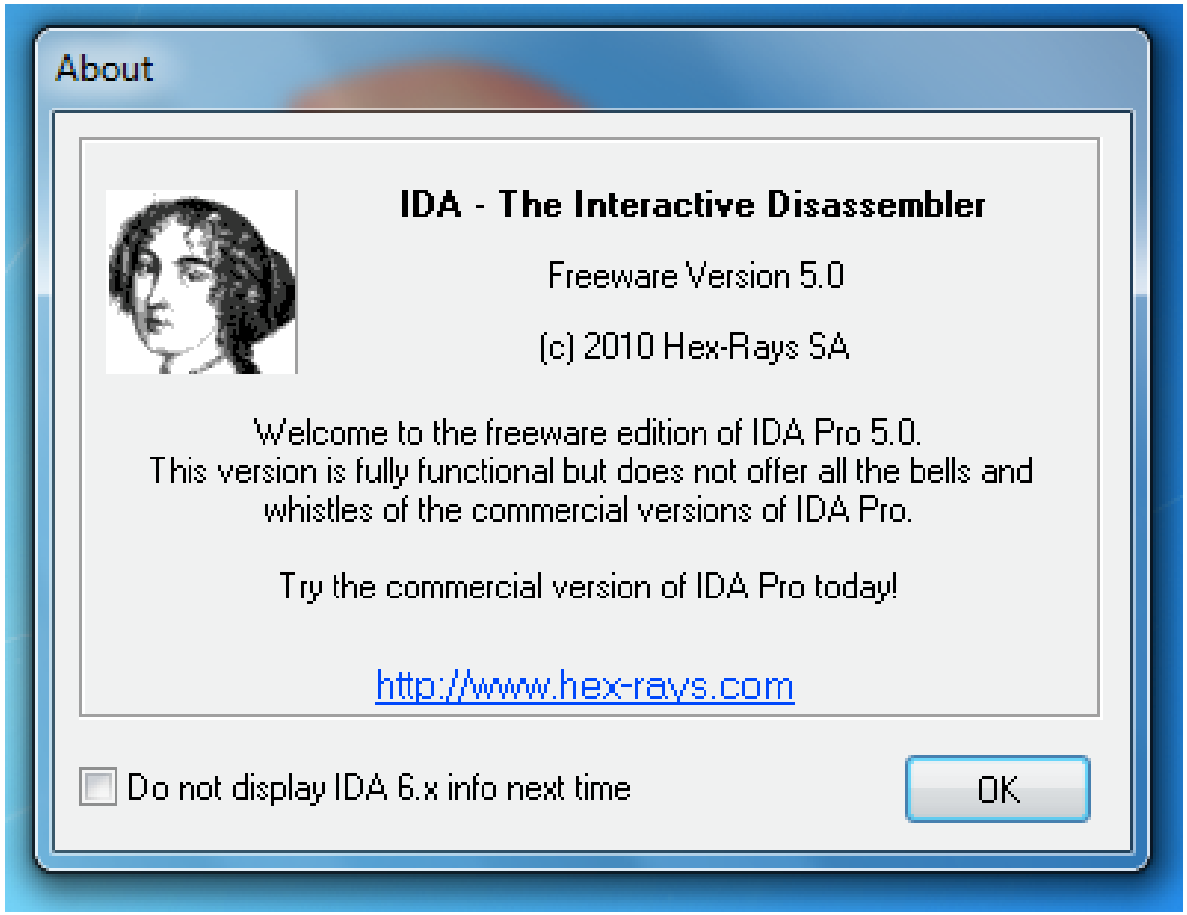


Figure 5-1. Code level examples

IDA Pro


About
IDA - The Interactive Disassembler
Freeware Version 5.0
(c) 2010 Hex-Rays SA
Welcome to the freeware edition of IDA Pro 5.0.
This version is fully functional but does not offer all the bells and whistles of the commercial versions of IDA Pro.
Try the commercial version of IDA Pro today!
http://www.hex-rays.com
Do not display IDA 6.x info next time
OK