Laboratorio di Software Security 2023/2024

Lezione 1

Marco Campion

marco.campion@univr.it

Goals

1) Understanding some well known software **vulnerabilities**

2) Understanding how to **exploit** such vulnerabilities as an attacker

3) Understanding the potential impact of such attacks and how to **prevent** them

This lab is a starting point!

- This is an introductory lab on software attacks/hacking/defenses!
- Good starting point for who is interested to study more complex software attacks/defenses
- How to learn more on this topic:
  - Cyberchallenge univr (https://cyberchallenge.it/)
  - Online tutorials (codearcana, roman, nightmare, …)
  - CTF contests (https://ctftime.org/)
  - …

Software vulnerabilities

- A *vulnerability* is a weakness which can be exploited by an *attacker* to perform *unauthorized actions* within your program
- To exploit a vulnerability, an attacker relies on tools and techniques related to a software weakness

What is a vulnerability?

- ***Question:*** Is the following code *vulnerable?*

```c
int authenticate() {
        char* password = "MyPassword!";
        char  name[10];
        char* psw = malloc(256);

        printf("Enter your name: ");
        scanf("%s", name);
        printf("Enter the password: ");
        scanf("%s", psw);

        if (strcmp(password,psw) == 0) {
                printf("Authenticated with name:\n");
                printf(name);
                return 1;
        } else {
                printf("The password is wrong! Please try again!\n");
                return 0;
        }
}
```

What is a vulnerability?

- ***Question:*** Is the following code *vulnerable*? ***Yes!***

There are (at least) **4 vulnerabilities** in this code

```c
int authenticate() {
        char* password = "MyPassword!";
        char  name[10];
        char* psw = malloc(256);

        printf("Enter your name: ");
        scanf("%s", name);
        printf("Enter the password: ");
        scanf("%s", psw);

        if (strcmp(password,psw) == 0) {
                printf("Authenticated with name:\n");
                printf(name);
                return 1;
        } else {
                printf("The password is wrong! Please try again!\n");
                return 0;
        }
}
```

What is a vulnerability?

- ***Question:*** Is the following code *vulnerable?* ***Yes!***

There are (at least) 4 vulnerabilities in this code:

- *Hardcoded password*

```c
int authenticate() {
        char* password = "MyPassword!";
        char  name[10];
        char* psw = malloc(256);

        printf("Enter your name: ");
        scanf("%s", name);
        printf("Enter the password: ");
        scanf("%s", psw);

        if (strcmp(password,psw) == 0) {
                printf("Authenticated with name:\n");
                printf(name);
                return 1;
        } else {
                printf("The password is wrong! Please try again!\n");
                return 0;
        }
}
```

What is a vulnerability?

- ***Question:*** Is the following code *vulnerable?* ***Yes!***

There are (at least) 4 vulnerabilities in this code:

- *Hardcoded password*
- *Potential stack-overflow*

```
int authenticate() {
        char* password = "MyPassword!";
        char  name[10];
        char* psw = malloc(256);

        printf("Enter your name: ");
        scanf("%s", name);
        printf("Enter the password: ");
        scanf("%s", psw);

        if (strcmp(password,psw) == 0) {
                printf("Authenticated with name:\n");
                printf(name);
                return 1;
        } else {
                printf("The password is wrong! Please try again!\n");
                return 0;
        }
}
```

What is a vulnerability?

- ***Question:*** Is the following code *vulnerable?* ***Yes!***

There are (at least) 4 vulnerabilities in this code:

- *Hardcoded password*
- *Potential stack-overflow*
- *Potential heap-overflow*

```c
int authenticate() {
        char* password = "MyPassword!";
        char  name[10];
        char* psw = malloc(256);

        printf("Enter your name: ");
        scanf("%s", name);
        printf("Enter the password: ");
        scanf("%s", psw);

        if (strcmp(password,psw) == 0) {
                printf("Authenticated with name:\n");
                printf(name);
                return 1;
        } else {
                printf("The password is wrong! Please try again!\n");
                return 0;
        }
}
```

What is a vulnerability?

- ***Question:*** Is the following code *vulnerable?* ***Yes!***

There are (at least) 4 vulnerabilities in this code:

- *Hardcoded password*
- *Potential stack-overflow*
- *Potential heap-overflow*
- *Potential format-string vulnerability*

```c
int authenticate() {
        char* password = "MyPassword!";
        char  name[10];
        char* psw = malloc(256);

        printf("Enter your name: ");
        scanf("%s", name);
        printf("Enter the password: ");
        scanf("%s", psw);

        if (strcmp(password,psw) == 0) {
                printf("Authenticated with name:\n");
                printf(name);
                return 1;
        } else {
                printf("The password is wrong! Please try again!\n");
                return 0;
        }
}
```

Sources of vulnerabilities

- Complexity, inadequacy, and (uncontrolled) changes
- Incorrect or changing assumptions (capabilities, inputs, outputs)
- Flawed specifications and designs
- Poor implementation of software interfaces (input validation, error and exception handling)
- Unintended, unexpected interactions with other components with the software’s execution environment
- ***Inadequate knowledge of secure coding practices***

Lab Software Security program overview

- Background on x86 architectures
- Debugging with gdb
- Reverse engineering
- Patching executables
- Python 3 library : pwntools
- Buffer overflow attacks: variables overriding and RA corruption
- Buffer overflow attacks: arbitrary code execution
- OS and Compiler-level defenses

Prerequisites for this Lab

- Basic knowledge of C/C++ and Unix-based OS

Prerequisites for this Lab

- Basic knowledge of C/C++ and Unix-based OS

- Desktop/laptop with *Intel x86/x86-64* microprocessor and Unix-based OS (e.g. Ubuntu)

  - If you have a machine with *Intel x86/x86-64* microprocessor :
    - Either you manually install on your Unix OS all the necessary tools
    - Or you can download our Ubuntu virtual machine with everything you need

  - If you don't have a machine with *Intel x86/x86-64* microprocessor :
    - Use Virtual Lab *(new from this year!)*

More infos at the end of this lecture!

Prerequisites for this Lab

- Basic knowledge of C/C++ and Unix-based OS
- Desktop/laptop computer with *Intel x86/x86-64* microprocessor and Unix-based OS (e.g. Ubuntu)
- **Very good knowledge of x86 architectures!**

In this lecture

- Background on x86 architectures
- Debugging with gdb
- Reverse engineering
- Patching executables
- Python 3 library : pwntools
- Buffer overflow attacks: variables overriding and RA corruption
- Buffer overflow attacks: arbitrary code execution
- OS and Compiler-level defenses

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

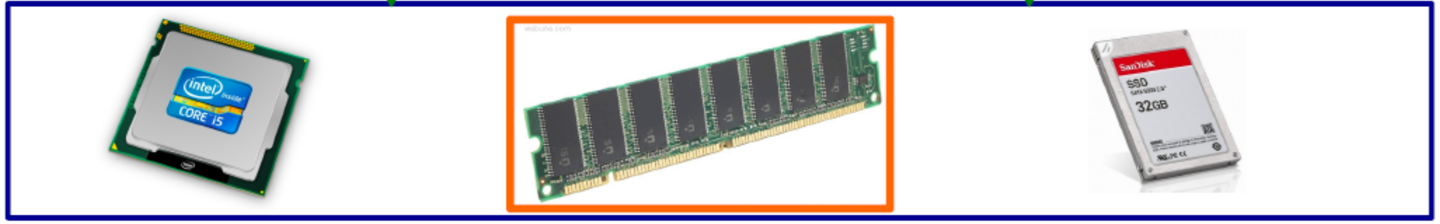
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows Mac

Computer system:



From code to programs

- Compiling a C program is a multi-stage process composed of four steps:
  1) preprocessing
  2) compilation
  3) assembly
  4) linking

From code to programs: preprocessing

- In the first phase some *preprocessor commands* (in C they start with ‘#’) are interpreted:

```c
#include <stdio.h>

#define MESSAGE "Hello world!"

int main() {
    printf(MESSAGE);
    return 0;
}
```

gcc –E hello.c

```c
# 2 "hello.c" 2



# 5 "hello.c"
int main() {
    printf("Hello world!");
    return 0;
}
```

From code to programs: compilation

- In the second phase, preprocessed code is translated into *assembly instructions*:

```
#include <stdio.h>

#define MESSAGE "Hello world!"

int main() {
    printf(MESSAGE);
    return 0;
}
```

gcc –s hello.c

```
# 2 "hello.c" 2



# 5 "hello.c"
int main() {
    printf("Hello world!");
    return 0;
}
```

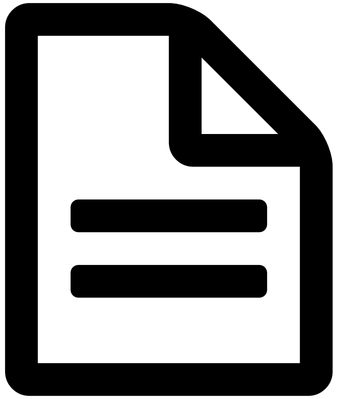
```
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
```

From code to programs: assembly

- In the *assembly* phase assembly instructions are translated into *machine* or *object code*:

```c
#include <stdio.h>

#define MESSAGE "Hello world!"

int main() {
    printf(MESSAGE);
    return 0;
}
```

gcc –c hello.c



hello.o

```
# 2 "hello.c" 2



# 5 "hello.c"
int main() {
    printf("Hello world!");
    return 0;
}
```

```
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
```

From code to programs: linking

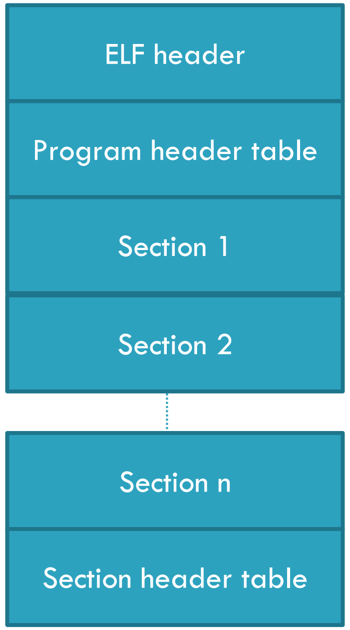
- In the last phase (multiple) *object code* are combined in a single executable
- In the generated file, references (links) to the used library are added

Static vs Dynamic linking

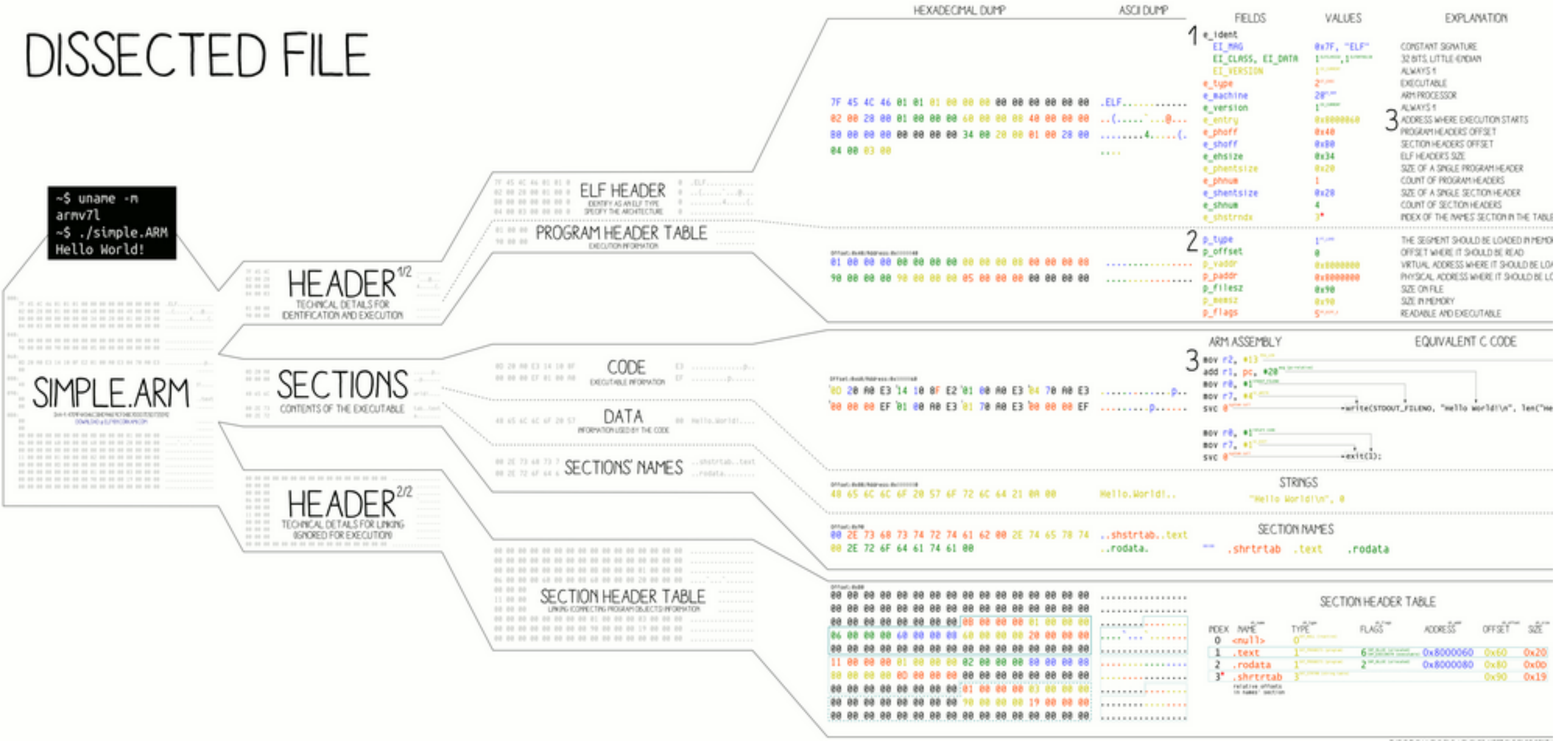
- Two approaches can be used in the linking phase:
  - Static Link
    - Binaries are *self-contained* and do not depend on any external libraries
  - Dynamic Link
    - Binaries rely on system libraries that are loaded when needed
    - Mechanisms are needed to *dynamically* relocate code

Executable and Linkable Format (ELF)

- Standard file format for executables in Unix-like systems
- Any ELF file is structured as:
  - an ELF header describing the file content for execution
  - a Program header table providing info about how to create a process image
  - a sequence of Sections containing what is needed for linking (instructions, data, symbol table, relocation information,…)
  - a Section header table with a description of previous sections
- Analogous format in Windows is *Portable Executable* (PE)
- Analogous format in macOS and iOS is *Mach-O*



ELF: Overview



ELF: Relevant sections

- .text: contains the executable instructions of a program
- .bss: contains uninitialised data that contribute to the program's memory image
- .data, .data1: contain initialized data that contribute to the program's memory image
- .rodata, .rodata1: are similar to .data and .data1, but refer to read-only data
- .symtab: contains the program's symbol table
- .dynamic: provides linking information

Recall storage sizes

Assembly word $\neq$ Processor word

- A bit can be either 0 or 1
- 1 BYTE = 8 bit, 1 WORD = 2 bytes, 1 DWORD = 4 bytes, 1 QUADWORD = 8 bytes
- The basic storage unit for all data in x86 architectures is 1 byte

- Least significant bit (LSB) = the right-most bit
- Most significant bit (MSB) = the left-most bit

LSB

0	0	1	0	1	1	0	1
$0*2^7$	$0*2^6$	$1*2^5$	$0*2^4$	$1*2^3$	$1*2^2$	$0*2^1$	$1*2^0$
		32		8	4		1

$= 45_{10}$

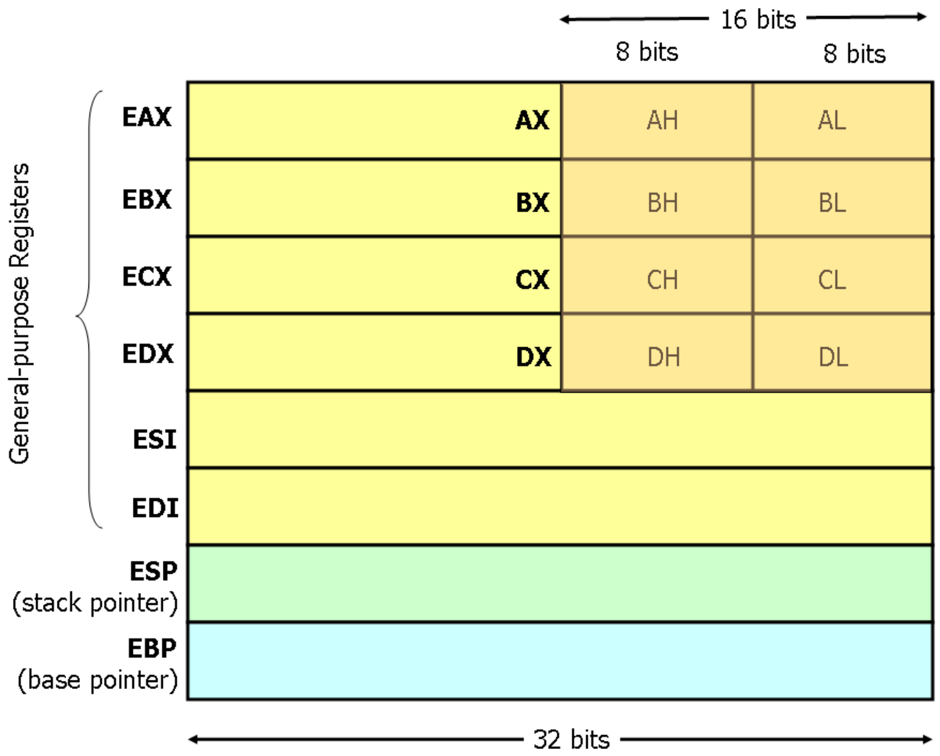
Hexadecimal integers

- In x86 assembly, hexadecimal (hex) values are used as a compact form for representing binary numbers
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
- Each digit in a hex integer represents 4 bits
- Two hex digits together represent a byte
- In C language they are written as 0x$\underbrace{\text{FA}}_{\text{1 byte}}\underbrace{\text{1D}}_{\text{1 byte}}$...

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

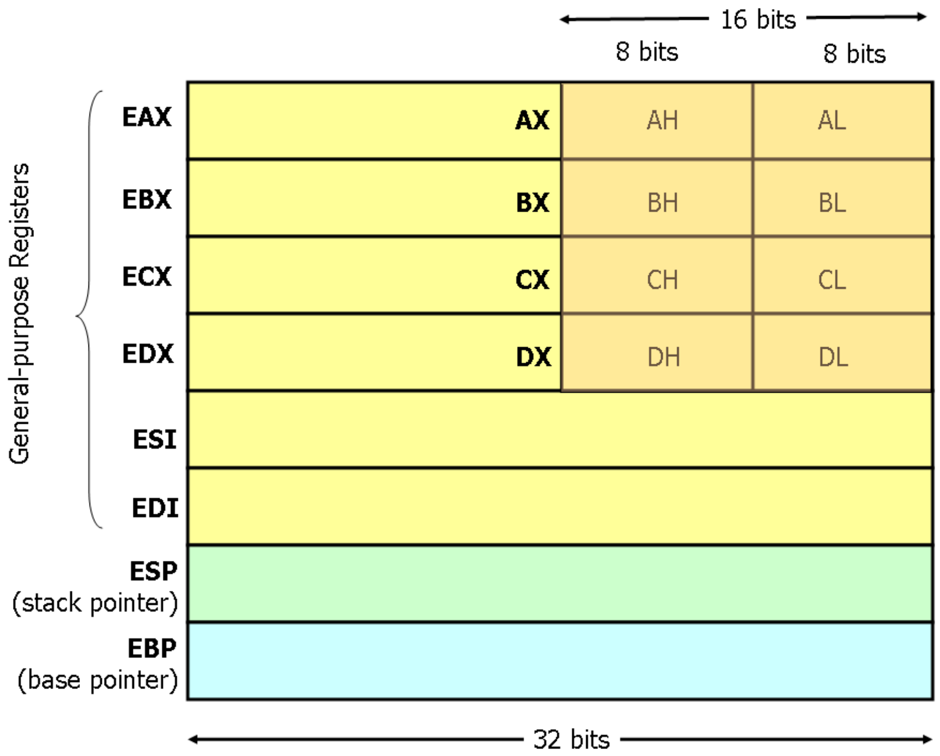
x86-32 Registers

- x86-32 processors have eight 32-bit general purpose registers
- The register names are mostly historical…
  - *EAX* used to be called *the accumulator* since it was used by a number of arithmetic operations
  - *ECX* was known as the *counter* since it was used to hold a loop index
- Two are reserved for special purposes:
  - the *stack pointer* (ESP)
  - the *base pointer* (EBP)



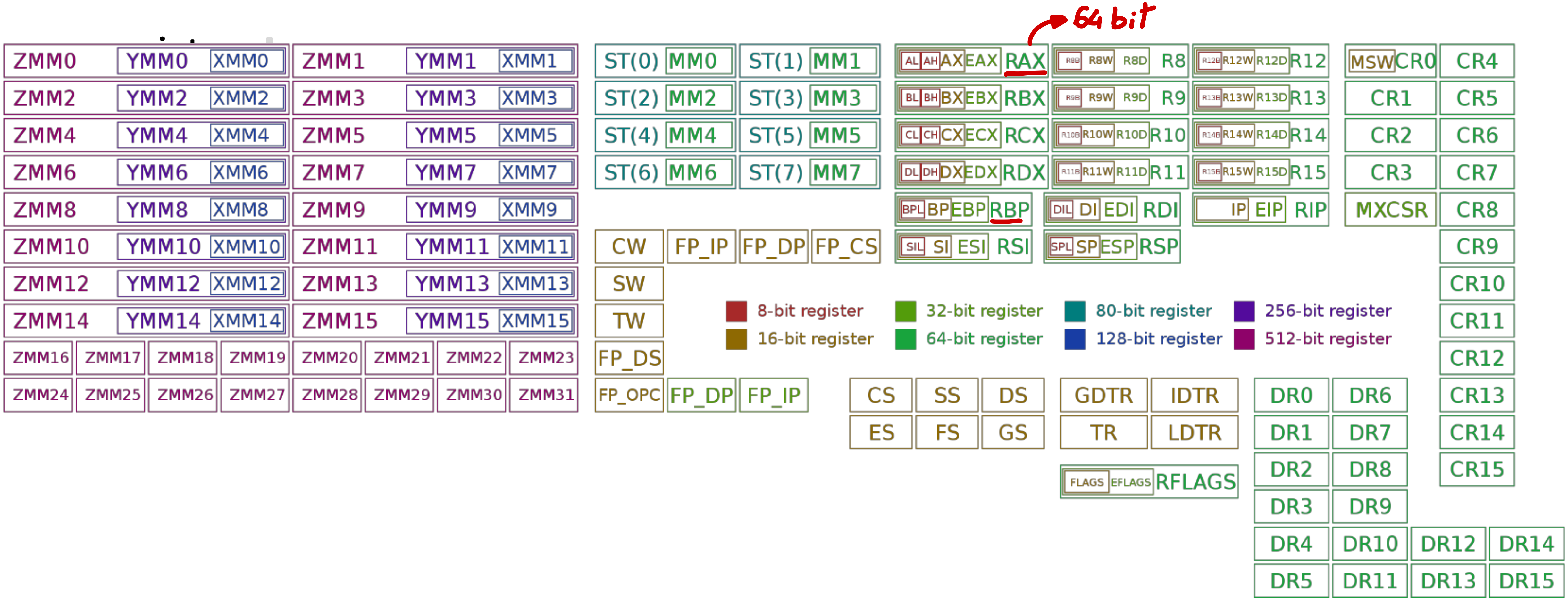
x86-32 Registers

- For each register, subsections may be used
- For example:
  - *AX* refers to the least significant 2 bytes of EAX
  - *AL* refers to the least significant byte of *AX*
  - *AH* refers to the most significant byte of *AX*
- These sub-registers are mainly hold-overs from older, 16-bit versions of the instruction set



x86-64 Registers

- x86-64 (also called amd64 or x64) architectures provide a larger set of registers



x86 Memory management

- In some programming languages, like C, memory management can be controlled by programmers:
  - memory can be dynamically allocated and deallocated
  - memory address of variables can be obtained (pointers)

- If *x* is a variable, *&x* denotes the pointer to x, i.e., the memory address where x is stored

Memory allocation

- Let us consider the following simple C program:

```
#include <stdio.h>

int main() {
    int i;
    char c;
    short s;
    long l;

    printf("i is allocated at %p\n", &i);
    printf("c is allocated at %p\n", &c);
    printf("s is allocated at %p\n", &s);
    printf("l is allocated at %p\n", &l);
}
```

Variable declarations

Variable addresses

We can assume that:

- *int* needs 4 bytes
- *char* needs 1 byte
- *short* needs 2 bytes
- *long* needs 8 bytes

Memory allocation

- Memory is just a sequence (array) of bytes each with a unique adress:

i_1	i_2	i_3	i_4	c_1	s_1	s_2	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8

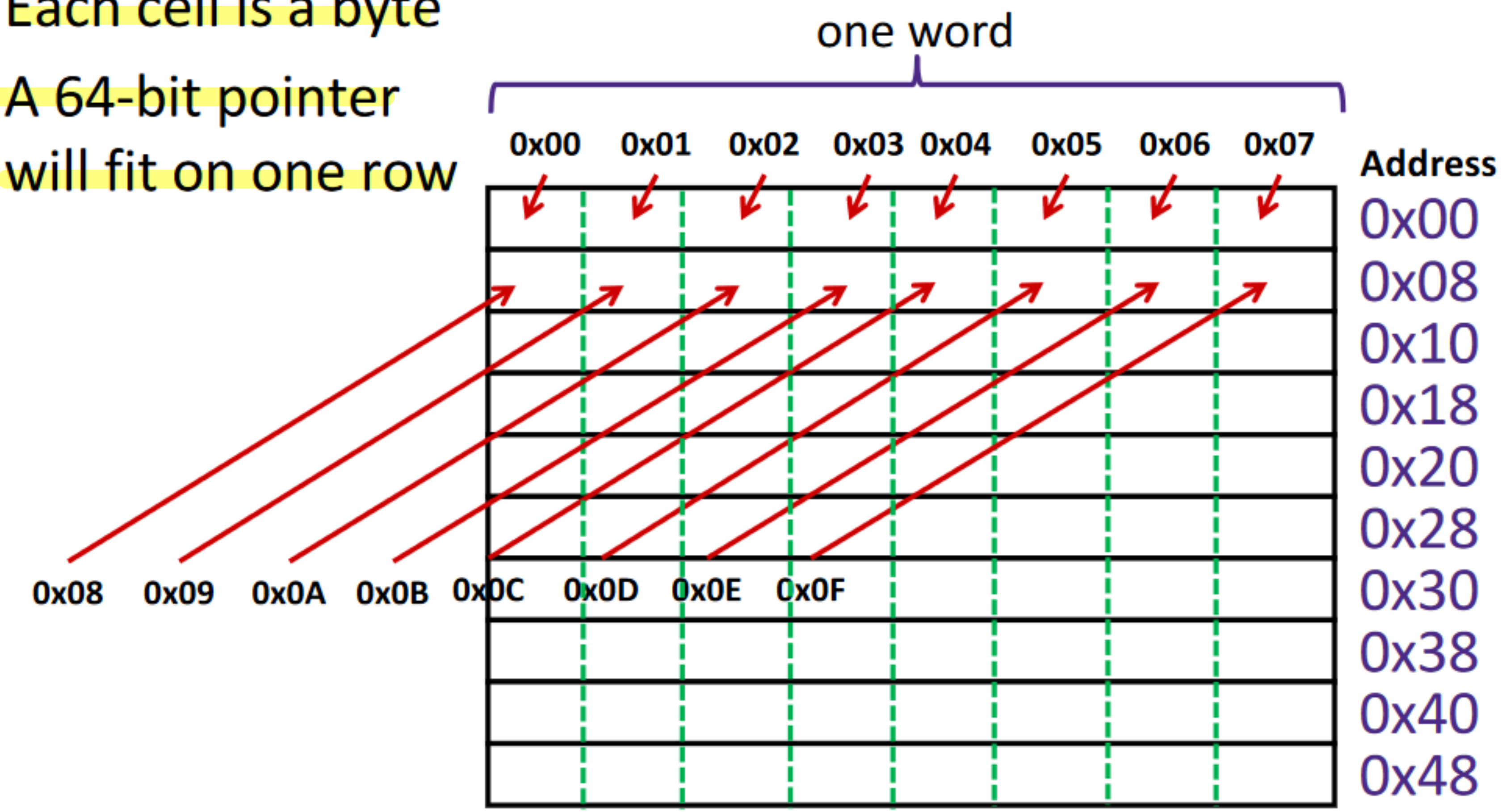
- Compilers may introduce padding or change the order of data in memory for optimization

- Memory is represented as groups of bytes (matrix) where each raw has n bytes with n = processor word (e.g. for 64bit architectures, n = 8 bytes)

i_1	i_2	i_3	i_4	c_1	s_1	s_2	
l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8

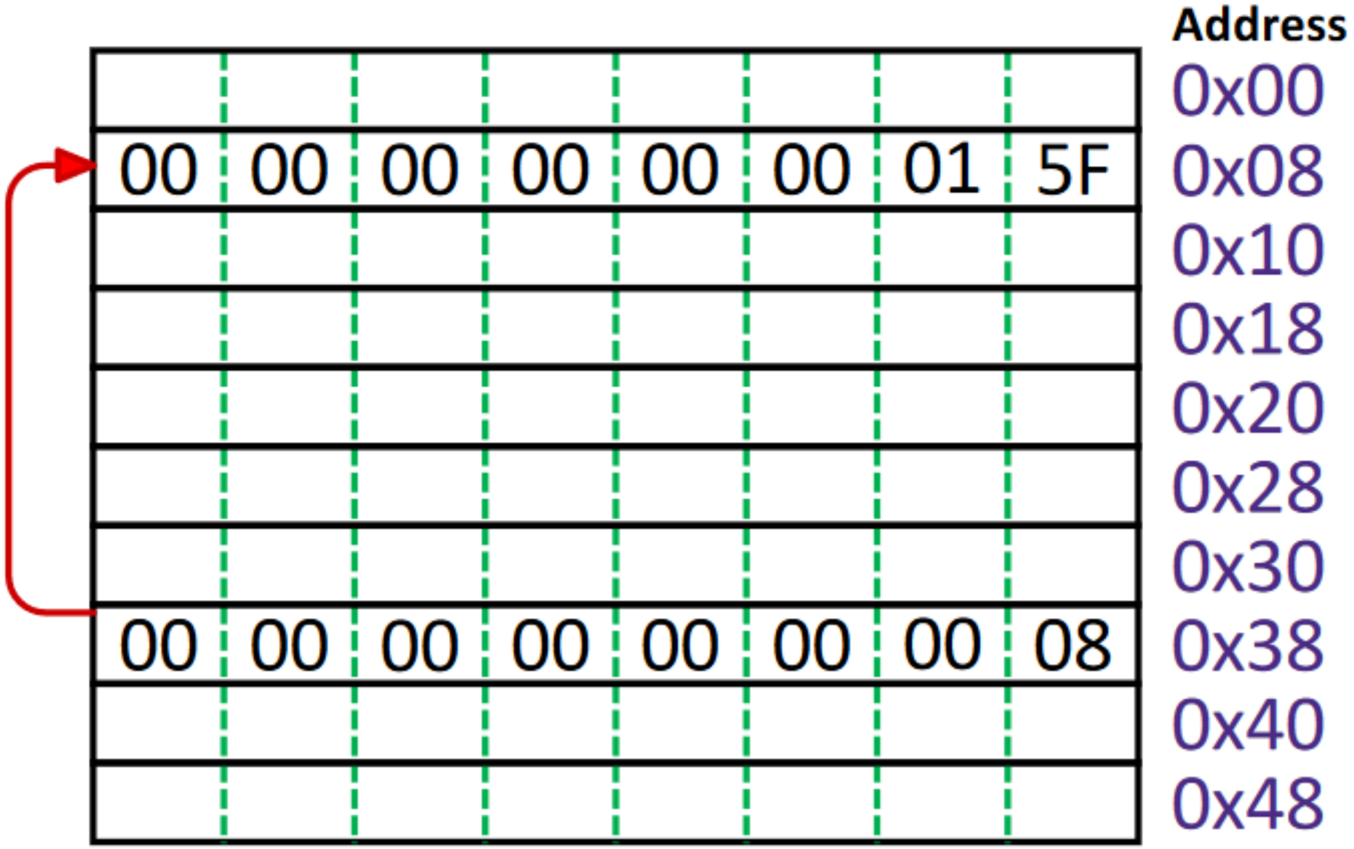
Memory allocation: example on a 64bit architecture

- A "64-bit (8-byte) word-aligned" view of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - A 64-bit pointer will fit on one row



Adresses and Pointers

- An *address* is a location in memory
- A *pointer* is a data object that holds an address
  - Address can point to *any* data
- Value 351 stored at address 0x08
  - $351_{10} = 15F_{16}$ = 0x 00 00 01 5F
- Pointer stored at 0x38 points to address 0x08



Sizes of data types in bytes

Java Data Type	C Data Type	32-bit (old)	x86-64
boolean	bool	1	1
byte	char	1	1
char		2	2
short	short int	2	2
int	int	4	4
float	float	4	4
	long int	4	8
double	double	8	8
long	long	8	8
	long double	8	16
(reference)	**pointer ***	**4**	**8**

Byte ordering

Endianes → convenzione su come salvare i byte in memoria

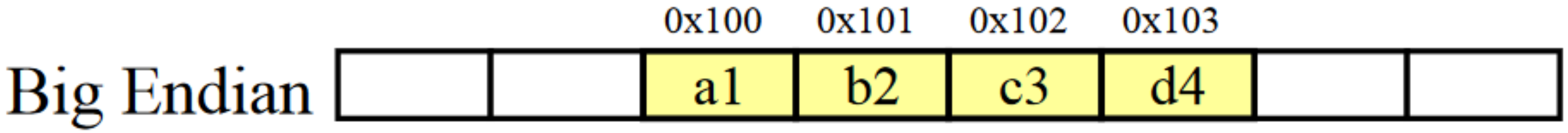
Bigendian
Little endian

- How should bytes within a word be ordered in memory?
  - **Example:** store the 4-byte (32-bit) `int`:
    `0x a1 b2 c3 d4`
- By convention, ordering of bytes called *endianness*
  - The two options are big-endian and little-endian

Byte ordering

- Big-endian (SPARC, z/Architecture)
  - Least significant byte has highest address

- **Example:** 4-byte data 0xa1b2c3d4 at address 0x100



Byte ordering

- Big-endian (SPARC, z/Architecture)
  - Least significant byte has highest address
- Little-endian (x86, x86-64)
  - Least significant byte has lowest address

- **Example:** 4-byte data 0xa1b2c3d4 at address 0x100

			0x100	0x101	0x102	0x103		
Big Endian			a1	b2	c3	d4		

			0x100	0x101	0x102	0x103		
Little Endian			d4	c3	b2	a1		

Byte ordering

- Big-endian (SPARC, z/Architecture)
  - Least significant byte has highest address
- Little-endian (x86, x86-64)
  - Least significant byte has lowest address
- Bi-endian (ARM, PowerPC)
  - Endianness can be specified as big or little

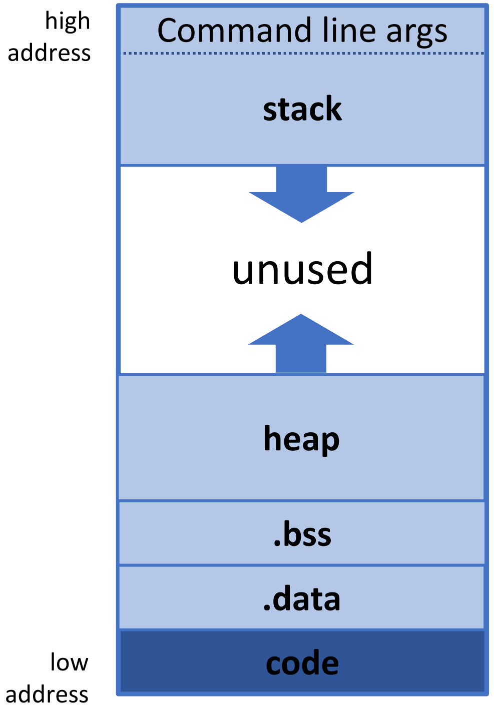
- **Example:** 4-byte data 0xa1b2c3d4 at address 0x100

			0x100	0x101	0x102	0x103		
Big Endian			a1	b2	c3	d4		

			0x100	0x101	0x102	0x103		
Little Endian			d4	c3	b2	a1		

Memory segments

- Memory is allocated for each process (a running program) to store data and code.
- This allocated memory consists of different segments:
  - *stack:* for local variables
  - *heap:* for dynamic memory
  - *data segment:*
    - *global uninitialized variables (.bss)*
    - *global initialized variables (.data)*
  - *code segment*



Intel x86 Instruction Sets

- We provide a short introduction of a small but useful subset of the available instructions and assembler directives of Intel x86 assembly language
- A detailed description can be found at the following links:
  - Short Assembly Guide
    - https://www.cs.virginia.edu/~evans/cs216/guides/x86.html
  - Online tutorial
    - https://www.tutorialspoint.com/assembly_programming/index.htm
  - Free e-book
    - http://mirror.easyname.at/nongnu/pgubook/ProgrammingGroundUp-1-0-booksize.pdf
  - Intel's Pentium Manuals
    - http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

x86 Instructions : Intel vs AT&T syntaxes

- x86 architectures (both 32- and 64-bit) has two alternative syntaxes available for assembly language

- AT&T syntax: `movl $1, %eax`

- Intel syntax: `mov eax, 1`

We will consider this syntax!

x86 Instructions : Data Movement Instructions

- MOV : copies data from right to left

Syntax

```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>
```

- Square brackets [val] are used to directly access memory adress contained in val

Examples

`mov eax, ebx` — copy the value in ebx into eax

`mov BYTE PTR [ebx], 2` *; Move 2 into the single byte at the address stored in EBX.*

x86 Instructions : Data Movement Instructions

- PUSH : places its operand onto the top of the stack

Syntax

```
push <reg32>
push <mem>
push <con32>
```

Examples

`push eax` — push eax on the stack

`push [var]` — push the 4 bytes at address *var* onto the stack

x86 Instructions : Data Movement Instructions

- POP : removes the 4-byte data from the top of the stack

Syntax

```
pop <reg32>
pop <mem>
```

Examples

`pop edi` — pop the top element of the stack into EDI.

`pop [ebx]` — pop the top element of the stack into memory at the four bytes starting at location EBX.

x86 Instructions : Data Movement Instructions

- LEA : load effective address

Syntax

`lea <reg32>,<mem>`

Examples

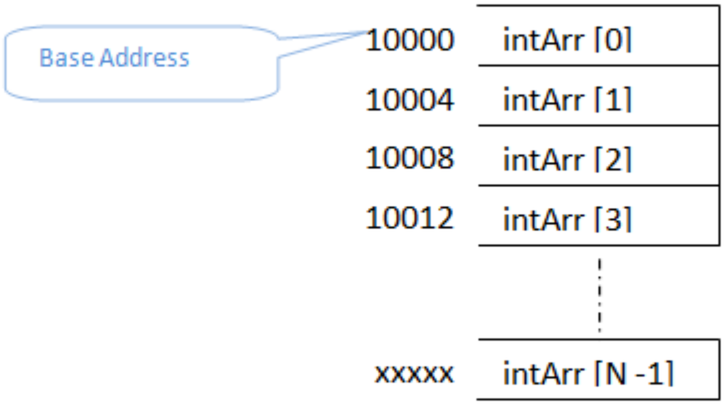
`lea edi, [ebx+4*esi]` — the quantity EBX+4*ESI is placed in EDI.

`lea eax, [var]` — the value in *var* is placed in EAX.

`lea eax, [val]` — the value *val* is placed in EAX.

C arrays in memory

- Each element of a C Array is allocated contiguously
- Initial adress of the array = adress of the first element
- Each element will occupy the memory space required to accomodate the values for its type, i.e., depending on elements datatype (1, 2, 4 or 8 bytes)
- Total memory allocated to an array = number of elements x size of one element



- MOV and LEA instructions used with arrays

Base adress
Size of each element
counter

mov eax, [ebx+4*esi] vs lea eax, [ebx+4*esi]

Load in eax the **value** at adress ebx+4*esi
eax = array[esi]

Load in eax the **adress** ebx+4*esi
eax = &array[esi]

C arrays in memory

- The order of the elements of a C array are unaffected by endianness!
  - The first element af an array (array[0]) is always at the lowest adress

char s[5] = {‘c’, ‘i’, ‘a’, ‘o’}

63	69	61	6F	00	
1000	1001	1002	1003	1004	Little-endian and big-endian

  - https://www.ascii-code.com/

- Each element of the array is affected by endianness!

int s[2] = {0x12345678, 0x9ABCDE}

78 56 34 12	DE BC 9A 00	Little-endian
1000	1004	

12 34 56 78	00 9A BC DE	Big-endian
1000	1004	

x86 Instructions : Arithmetic and Logic Instructions

- Arithmetic and Logic Instructions
  - *add op1,op2:* stores in *op1* the result of *op2+op1*
  - *sub op1,op2:* stores in *op1* the result of *op2-op1*
  - inc op: increments op by one
  - *and op1,op2*
  - *or op1,op2*
  - *xor op1,op2*
  - …

Perform the specified logical operation on the operands, storing the result in the first operand location

Control Flow

- x86 processor mantains an *Instruction Pointer* (IP) register of 32(or 64)-bit indicating the location in memory where the current instruction starts
- Normally, it increments to point to the next instruction in memory begins after execution an instruction
- Control flow instructions, like jumps, can update IP to point to specific labels

x86 Instructions: Control Flow Instructions

- JMP : unconditional jump

jmp — Jump

Transfers program control flow to the instruction at the memory location indicated by the operand.

Syntax
`jmp <label>`

Example
`jmp begin` — Jump to the instruction labeled `begin`.

```
       mov esi, [ebp+8]
begin: xor ecx, ecx
       mov eax, [esi]
```

x86 Instructions: Control Flow Instructions

➢ Jcondition : conditional jump

Syntax
`je <label>` (jump when equal)
`jne <label>` (jump when not equal)
`jz <label>` (jump when last result was zero)
`jg <label>` (jump when greater than)
`jge <label>` (jump when greater than or equal to)
`jl <label>` (jump when less than)
`jle <label>` (jump when less than or equal to)

Example
```
cmp eax, ebx
jle done
```

If the contents of EAX are less than or equal to the contents of EBX, jump to the label *done*. Otherwise, continue to the next instruction.

x86 Instructions: Control Flow Instructions

- CMP : compare the values of the two specified operands, setting the condition codes in the machine status word appropriately

Syntax

```
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
```

Example

```
cmp DWORD PTR [var], 10
jeq loop
```

If the 4 bytes stored at location *var* are equal to the 4-byte integer constant 10, jump to the location labeled *loop*.

x86 Instructions: Control Flow Instructions

- CALL : pushes the current code location onto the stack and then performs an unconditional jump to the code location indicated by the label operand. It is used to call functions.

- RET : implements a return from a function. It first pops a code location off the stack and then performs an unconditional jump to the retrieved code location

Syntax

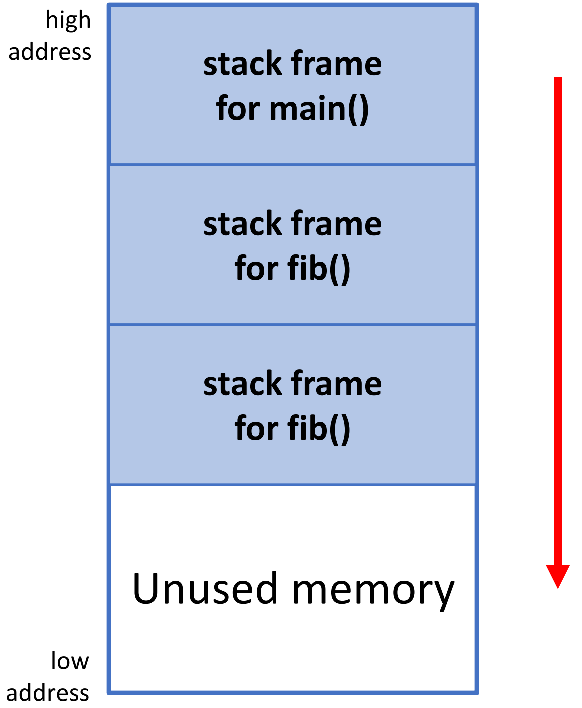
```
call <label>
ret
```

The stack

diviso in frame logici, cresce verso il basso.

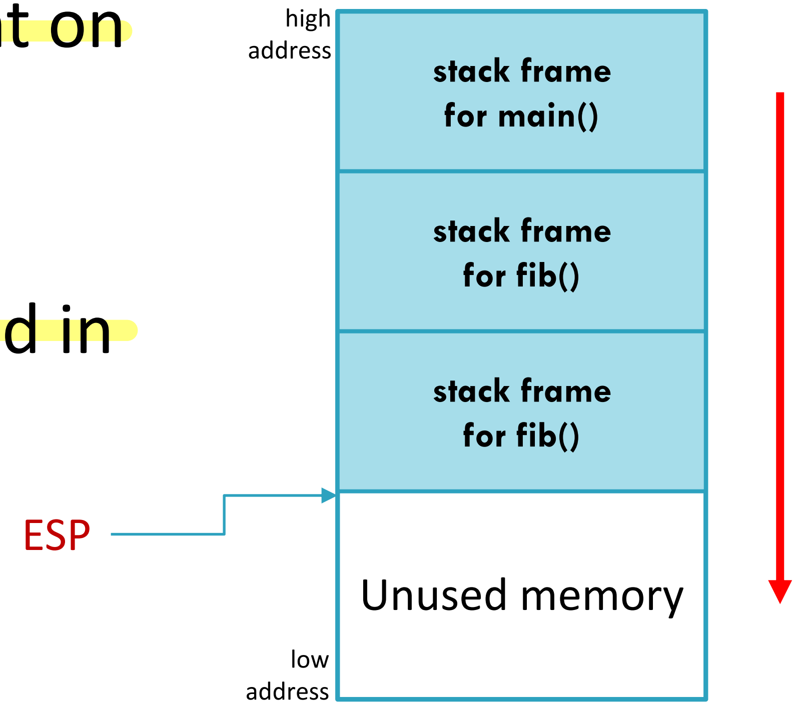
- The stack consists of a sequence of *stack frames* (or activation records), each for each function call:
  - allocated on *call*
  - de-allocated on *return*

```
int main(int argc, char **argv) {
  int f = fib( n: 10);
  printf("FIB(10)=%d\n",f);
}

int fib(int n) {
  int f1;
  int f2;
  if (n<=2) {
      return 1;
  } else {
      f1 = fib( n: n-1);
      f2 = fib( n: n-2);
      return f1+f2;
  }
}
```



The stack

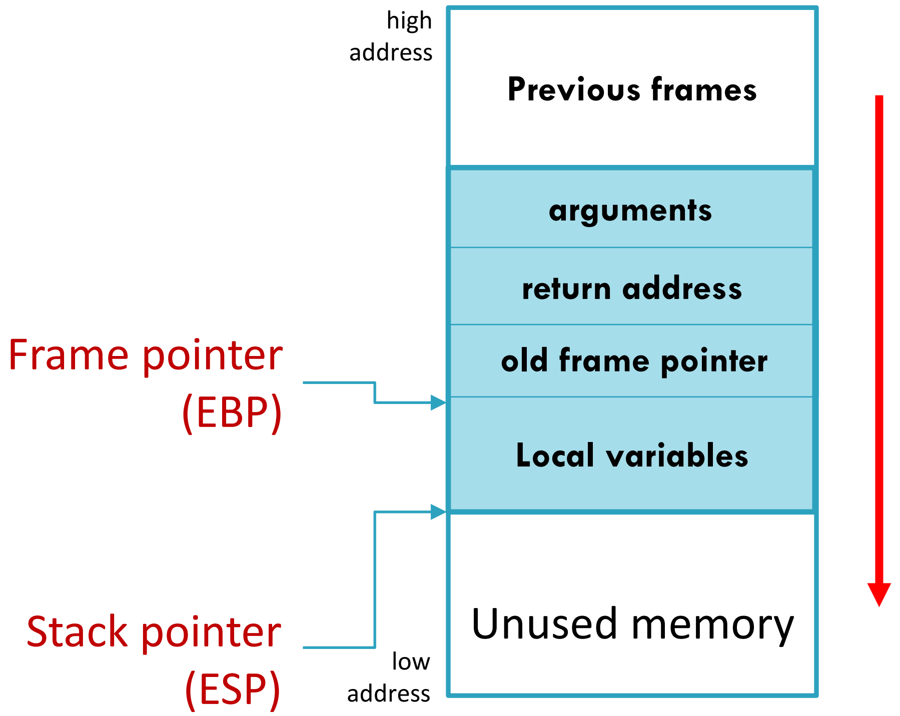
- The stack pointer (SP) refers to the last element on the stack
- On *x86* architectures, the stack pointer is stored in the ESP (Extended Stack Pointer) register



Stack frame (for x86)

- In *x86* architecture, each stack frame contains (in order):
  - Function arguments
  - Copies of registries that must be restored:
    - return address
    - previous frame pointer
  - Local variables

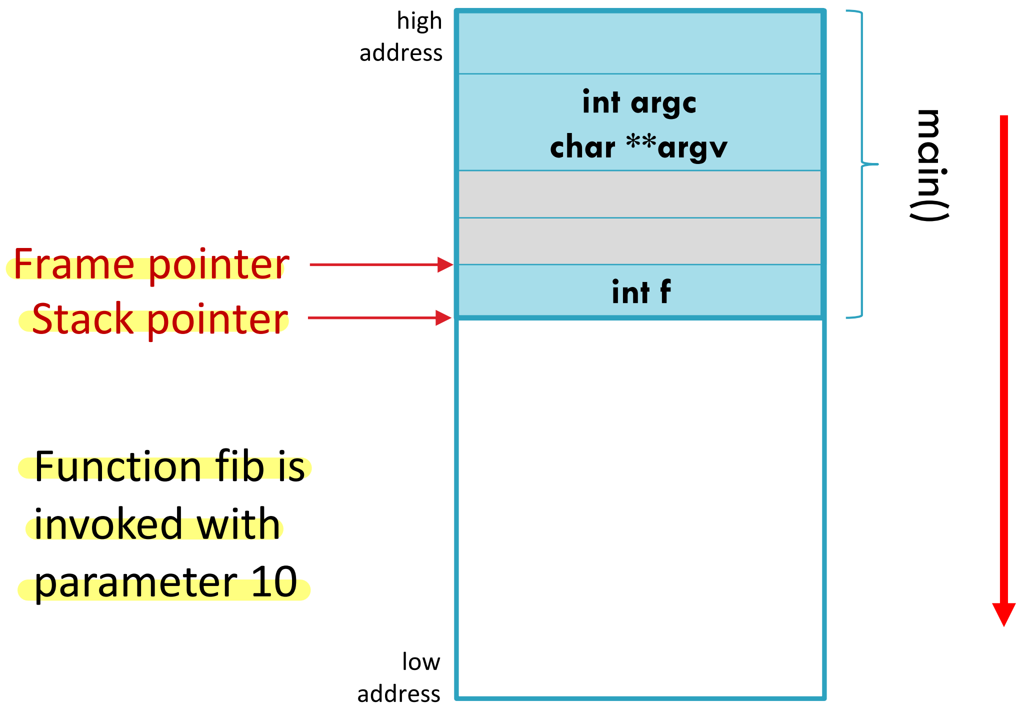
- Frame pointer, named Extended Base Pointer (EBP), provides a starting point to local variables



Stack example

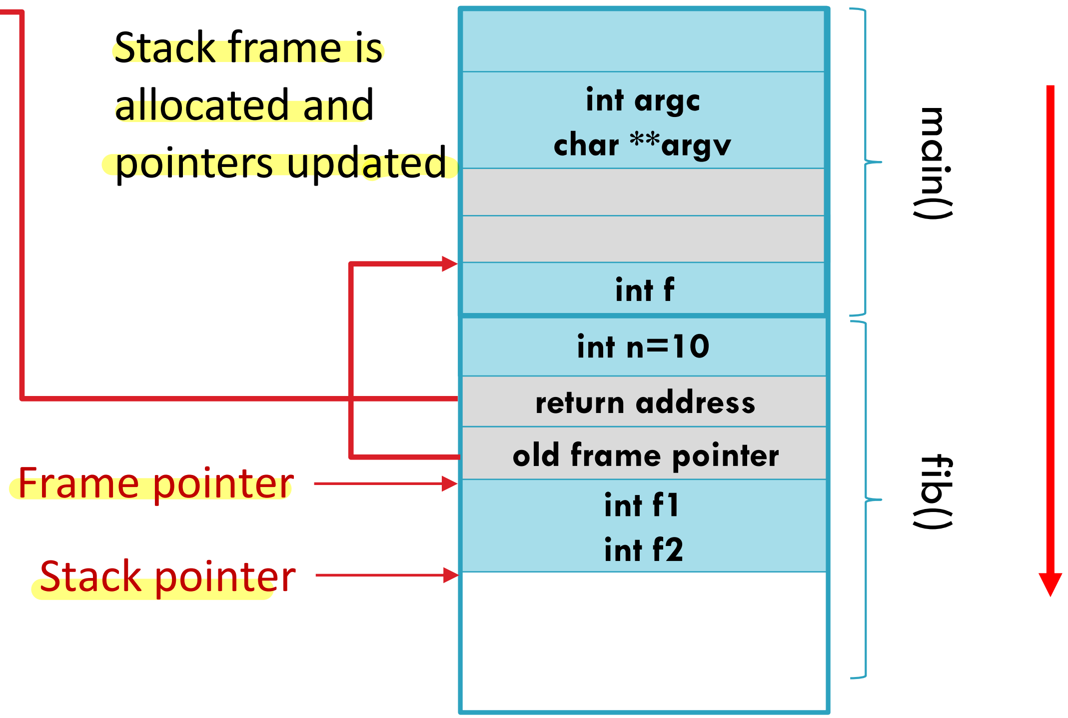
```c
int main(int argc, char **argv) {
  int f = fib( n: 10);
  printf("FIB(10)=%d\n",f);
}

int fib(int n) {
  int f1;
  int f2;
  if (n<=2) {
      return 1;
  } else {
      f1 = fib( n: n-1);
      f2 = fib( n: n-2);
      return f1+f2;
  }
}
```

Function fib is invoked with parameter 10



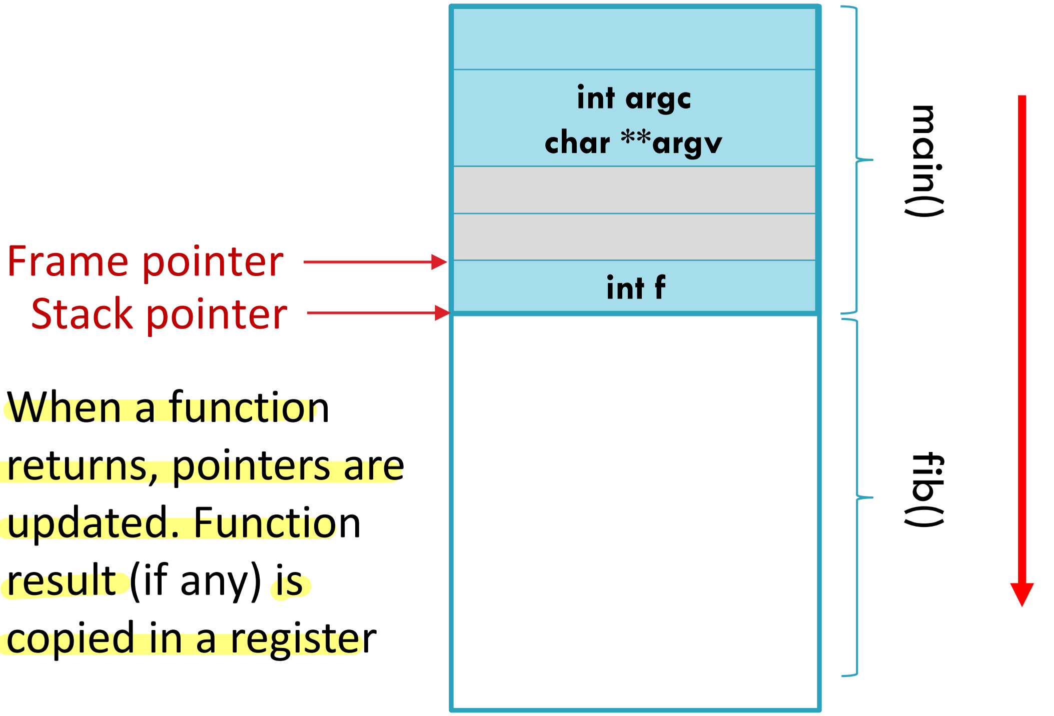
Stack example

```c
int main(int argc, char **argv) {
  int f = fib( n: 10);
  printf("FIB(10)=%d\n",f);
}

int fib(int n) {
  int f1;
  int f2;
  if (n<=2) {
      return 1;
  } else {
      f1 = fib( n: n-1);
      f2 = fib( n: n-2);
      return f1+f2;
  }
}
```



Stack example

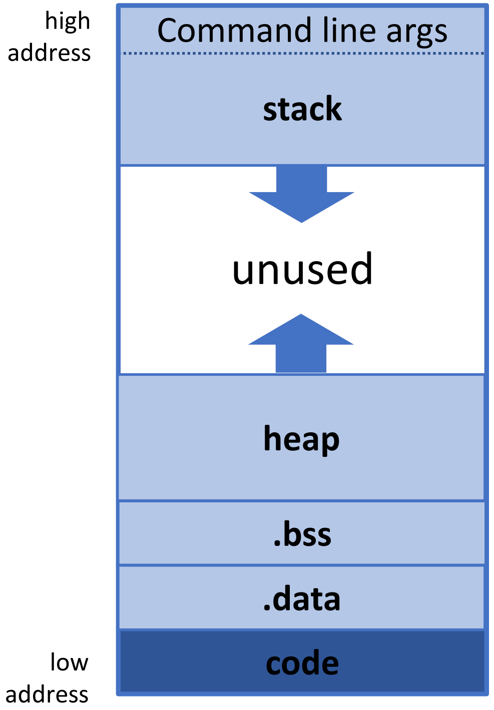
```c
int main(int argc, char **argv) {
    int f = fib( n: 10);
    printf("FIB(10)=%d\n",f);
}

int fib(int n) {
    int f1;
    int f2;
    if (n<=2) {
        return 1;
    } else {
        f1 = fib( n: n-1);
        f2 = fib( n: n-2);
        return f1+f2;
    }
}
```



When a function returns, pointers are updated. Function result (if any) is copied in a register

The heap

- Memory allocation and de-allocation in the stack is very fast
  - However, this memory cannot be used after a function returns
- The heap is used to store dynamically allocated data that outlive function calls:
  - This area is under programmer's responsibility



Memory management functions

- Basic *C* functions for memory management are:
  - *malloc(int),* given an integer *n* allocates an area of *n* (continuous) byes and returns a pointer to that area
  - *free(void*),* deallocates the memory associated with a pointer

Debugging

- A debugger is a software tool that allows to:
  - run the target program under controlled conditions
  - track program operations in progress
  - monitor changes in computer resources
  - display the contents of memory
  - modify memory or register contents

Debugging

- Compilers can be instrumented to emit extra data that debugger can use for a more informative execution
- In the case of gcc compiler, the parameter $-g$ can be used
- Debugging information is stored in specific sections of ELF file:
  - .debug : contains info for symbolic debugging
  - .line : contains line number informations

```
CC> gcc -o hello -g hello.c
CC> readelf --debug-dump hello
Contents of the .debug_aranges section:

  Length:                   44
  Version:                  2
  Offset into .debug_info:  0x0
  Pointer Size:             8
  Segment Size:             0

    Address            Length
    0000000000400526 000000000000001a
    0000000000000000 0000000000000000

Contents of the .debug_info section:

  Compilation Unit @ offset 0x0:
   Length:        0x8d (32-bit)
   Version:       4
   Abbrev Offset: 0x0
```

GDB : The GNU Project Debugger

- GDB is a debugging tool that can be used to…
  - Start your program, specifying anything that might affect its behavior
  - Stop your program at the occurrence of specified conditions
  - Examine what happened, when your program stopped
  - Change memory o registers content while your program is running

GDB : The GNU Project Debugger

- GDB supports multiple languages:
  - Assembly (x86, ARM, MIPS…)
  - C
  - C++
  - Rust
  - …

GDB : The GNU Project Debugger

- GDB is a terminal tool and can be launched by executing program *gdb*

- You can invoke *gdb* by passing the program to debug

```
gdb &lt;program&gt;
```

- you can pass the *process ID* as a second argument to debug a running process:

```
gdb <program> <pid>
```

- Or equivalently use:

```
gdb -p <pid>
```

GDB : Startup

When executed, gdb performs the following main steps:

- sets up the command interpreter as specified by the command line
- Load configuration files
- Load symbols of debugged program
- Waits for user commands

GDB : running and arguments

- A *gdb* command consists of a single line of input, containing:
  - a *command name*
  - a sequence of *parameters*

- Command `run` can be used to `start` a program in *gdb*

- Command `help` can be used to access the list of available commands

- Commands `set args` and `show args` can be used to set and show program arguments

GDB : Stopping and Counting

- The principal reason to use a debugger is that we can stop a program before it terminates to check its status and, if we experience some problems, investigate and find out why.

- Inside *gdb,* a program may stop for:
  - a *breakpoint*
  - a *signal*
  - the completion of the execution of a *step*

GDB : Breakpoints

- A *breakpoint* makes your program stop whenever a certain point in the program is reached
  - details can be added to the breakpoint to control in finer detail whether your program stops
- A *watchpoint* is a special breakpoint that stops your program when the value of an expression changes
- A *catchpoint* is another special breakpoint that stops your program when a certain kind of event occurs

GDB : Breakpoints

- Breakpoints are set with the *break* command (abbreviated *b*):
  - `break location` set break point at the given location (addresses must be preceded by *)
  - `break` set break point at the next instruction
  - `break [location] if <cond>` set break point with the given condition

- A breakpoint location can be:
  - A line number
  - A label/function name
  - An address (must be preceded by `*`)

GDB : Continuing and Stepping

- When a program stops at a breakpoint, its execution can be resumed exploiting 2 functionalities:
  - Continuing with `continue` or `c` means resuming program execution until another breakpoint is found or your program completes normally
  - Next instruction `nexti` or `ni` means executing just one instruction

GDB : Inspecting the stack

- When your program has stopped, the first thing we need to know is where it *stopped* and *how* it got there
- The first thing to consider is the content of the *stack*
- gdb commands are available to examine the stack and to read the content of the *stack* and to read any of the stored *stack frames*
- In the *stack* frame you can find:
  - the location of the call in your program
  - the arguments of the call
  - the local variables of the function being called

GDB :

- The following commands can be used to read the content of the stack:
  - `frame [<selection>]`
    - prints a brief description of the selected stack frame.
  - `info frame [<selection>]`
    - prints a verbose description of the selected stack frame

- See *gdb* documentation for a (long) list of options

GDB : Other commands

- Command *disas* can be used to disassemble a given function

- The *print* command (abbreviated *p*) prints the content of an address or register (registers must be preceded by *$*)

  Example: `p *0x0809aaf6`

- Command *x* treats the content of a memory location or register as an address and prints its content

  Example: `x $eax`

- Command *call* calls a function of our inspected program

  Example: `call (void) function()`

- Command *set* modifies the value of a memory location or a register

  Example: `set $pc=0x400344`

GEF : GDB Enhanced Features

- GEF consists of a set of commands that extends GDB with additional features for *dynamic analysis* and *exploit development*.
- GEF is based on GDB Python API
- Main GEF features:
  - Embedded hexdump view
  - Automatic dereferencing of *data* and *registers*
  - Heap analysis
  - Display ELF information
- Detailed GEF documentation is available at

  https://gef.readthedocs.io/en/master/

Now…

…hands on!

Tools for the lab

If you already have a computer with Intel processor then, in order to be able to succesfully do the exercises, you can either:

a) manually download and install each tool we need:

- gcc-multilib (`sudo apt install gcc-multilib`)
- GEF (https://gef.readthedocs.io/en/master/)
- Ghidra (https://ghidra-sre.org/)
- Python 3 (https://www.python.org/downloads/)
- Pwntools (https://docs.pwntools.com/en/latest/index.html)

Tools for the lab

b) download our Ubuntu virtualbox with everything already installed:

https://univr-my.sharepoint.com/:u:/g/personal/marco_campion_univr_it/ESoL6ITf6YBMibrti16PfjoBXGQHVmUrKrtM6DxrQGK5IQ?e=jDbyVw

Username : swsec
Password : password

Import it with Oracle Virtual Box (https://www.virtualbox.org/)

Tools for the lab

If you don't have a machine with Intel processor then:

1) Connect to the univr intranet (either by connecting to the univr wifi or through vpn)
2) Visit https://virtualab.univr.it
3) Use your GIA credentials to authenticate
4) Select «*aula_virtuale*»
5) Select any virtual pc and start using the Ubuntu machine!

- Python 3 is already installed
- You can manually install GEF, Ghidra and pwntools with no root permissions
- gcc-multilib may not be installed!

Remember to logout at the end of your work!

Tutorial Exercise

- Disable ASLR on your system (it will be automatically reactivated on next restart):

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

- Run and analyze the program *tutorial* and answer the following questions:
  1) How many bytes are saved in the stack for the local variables of function `fun()`?
  2) What are the adresses (in the form register ± offset) of the two integer variables of function `fun()`?
  3) What is the adress of the third letter in the string `name[11]`?
  4) What are the adresses (in the form register ± offset) where the two static strings «`Insert your name:` » and «`You won!\n`» have been saved?
  5) How many bytes are there between the starting of `name[11]` and the old base pointer saved on the stack?
  6) What is the adress (in the form register ± offset) of the variable `n` given as argument to the function `fun()`?

- Can you execute the function secret_fun() ?

Thanks to…

CYBERCHALLENGE.IT

cini consorzio interuniversitario nazionale per l'informatica