Michele LORETI

Università di Camerino

Basic background

CYBER CHALLENGE.IT

CYBERSECURITY NATIONAL LABORATORY

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Goal

- This module provides an overview of the basic background used in the Software Security Area. The module is structured in two lectures.
- In this first lecture we will learn…
  - how binary files are stored;
  - the basic steps that transform a source code into programs;
  - principles of x86 architecture;
  - mechanisms used to support memory management.

Prerequisites

- Lecture:
  - Basic knowledge of C
  - Basic knowledge of Python

Outline

- From Code to Programs
- Executable and Linkable Format (ELF)
- x86 architectures
- Memory management and Calling conventions
- Debugging

Outline

- From Code to Programs
- Executable and Linkable Format (ELF)
- x86 architectures
- Memory management and Calling conventions
- Debugging

From code to programs

- Compiling a C program is a multi-stage process composed of four steps:
  - preprocessing
  - compilation
  - assembly
  - linking

GCC Compiler

- In this lecture *gcc* compiler will be used…
  - is the C compiler included in the *GNU Compiler Collection*
  - is available in the main operating systems
    - Standard compiler in all the UNIX/Linux distributions
- Detailed documentation is available at the following link:

https://gcc.gnu.org/onlinedocs/

From code to programs: preprocessing

- In the first phase *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*:

```c
#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*:

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

Outline

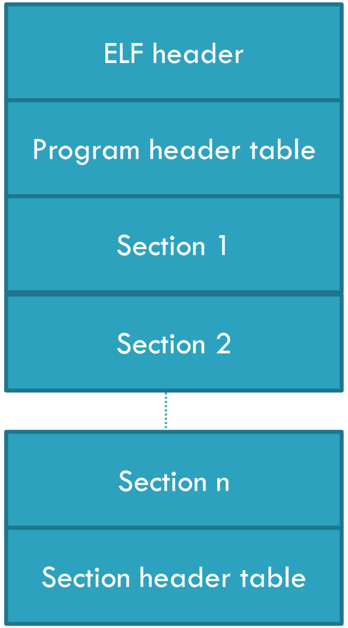
- From Code to Programs
- Executable and Linkable Format (ELF)
- x86 architectures
- Memory management and Calling conventions
- Debugging

Executable and Linkable Format

- The Executable and Linkable Format (ELF) is a common file format for object files
- There are three types of object files:
  - Relocatable file containing code and data that can be linked with other object files to create an executable or shared object file
  - Executable files holding a program suitable for execution
  - Shared object files that can be:
    - linked with other relocatable and shared object files to obtain another object file
    - used by a dynamic linker together with other executable files and object files to create a process image

Executable and Linkable Format

- Any ELF file is structured as:
  - an ELF header describing the file content
  - 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



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

Outline

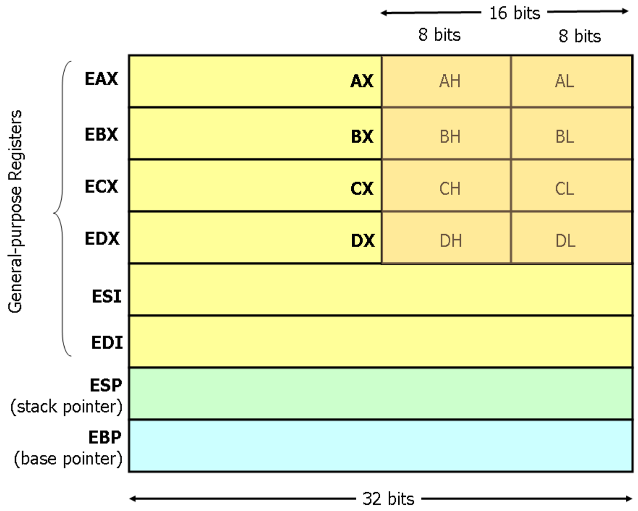
- From Code to Programs
- Executable and Linkable Format (ELF)
- x86 architectures
- Memory management and Calling conventions
- Debugging

x86 Instruction Sets

- We provide a short introduction of a small but useful subset of the available instructions and assembler directives of x86 assembly language
- A detailed description can be found at the following links:
  - Intel x86 Instruction Set Reference
    - http://www.felixcloutier.com/x86/
  - Intel's Pentium Manuals
    - http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

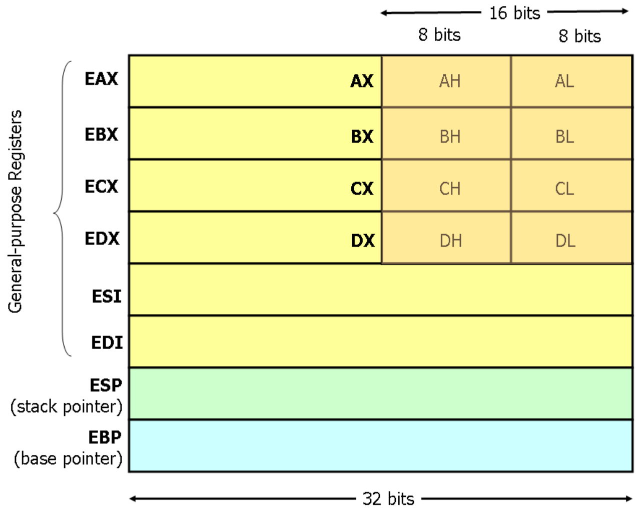
X86-32 Registers

- 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)



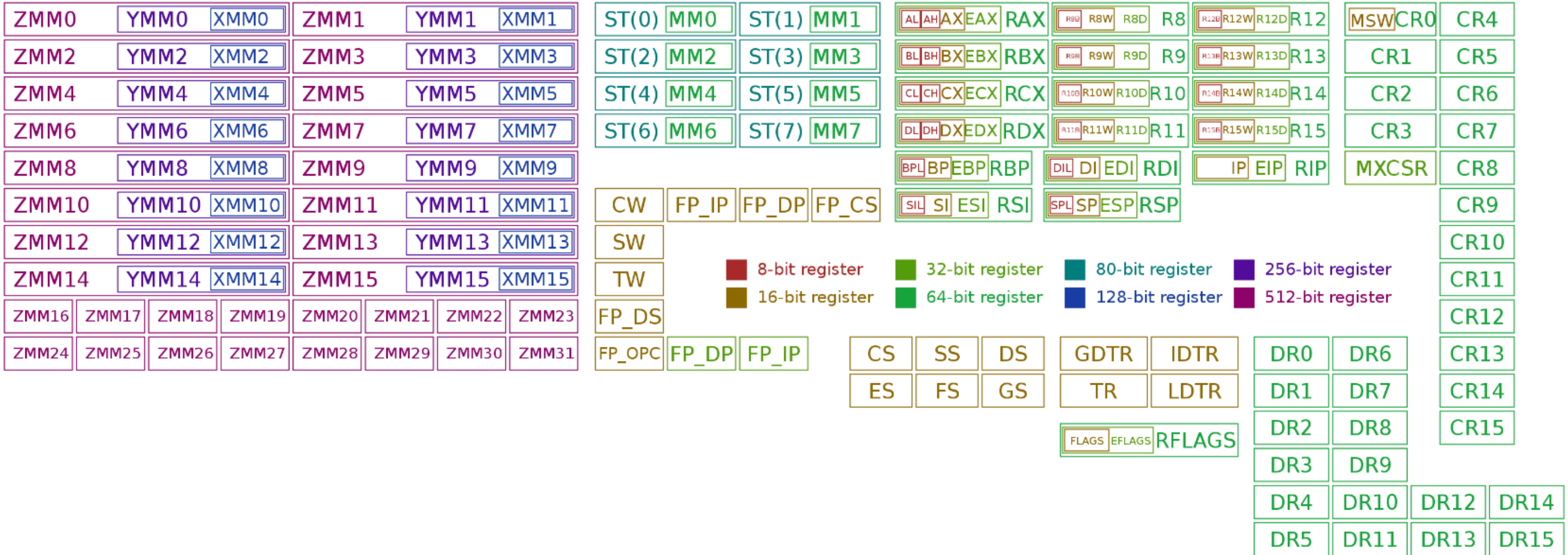
X86-32 Registers

- For the *EAX*, *EBX*, *ECX*, and *EDX* registers, 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 architecture provides a larger set of registers:



x86 Instructions

- Data Movement Instructions
  - *mov op1,op1:* copies the data item referred to by *op1* into the location referred to by *op2*
  - *push op1:* places its operand onto the top of the stack
  - *pop op1:* removes the 4-byte data element from the top of the stack
  - *lea op1,op2*: load the memory address indicated by *op2* into the register specified by *op1*

x86 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*
  - *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

x86 Instructions

- Control Flow Instructions
  - *jmp op:* jump to the instruction at the memory location specified by the operand *op*
  - *cmp op1,op2:* compares the values of the two specified operands and stores the result in the *machine status word*
  - *j<condition> op:* depending on the *<condition>* and on the context of *machine status word*, jumps to instruction at the memory location indicated by the operand

Outline

- From Code to Programs
- Executable and Linkable Format (ELF)
- x86 architectures
- Memory management and Calling conventions
- Debugging

Memory management

- Many of the modern programming languages allow programmers to use data types without having to know how they are represented
- Similarly, programmers often ignore where data is stored (allocated)…
  - compiler makes decisions, however at runtime allocation is under the control of Operating System and CPU

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:

```c
#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);
}
```

We can assume that:

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

Memory allocation...

- Let us consider the following simple C program:

```c
#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

We can assume that:

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

Memory allocation…

- Let us consider the following simple C program:

```
#include <stdio.h>

int main() {
    int i;
    char c;
    short s;
    long l;
                         Variable addresses
    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);
}
```

We can assume that:

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

Memory allocation…

- Memory is usually represented as a sequence of 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

In a n·8 architecture, bytes are arranged in groups of n:

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…


```c
#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);
}
```

We can run this simple program to observe how data are allocated in memory

Memory allocation…

- Different allocation policies can be used by *gcc*:

```
CC> gcc -o alignment alignment.c
CC> ./alignment
i is allocated at 0x7ffee117e7cc
c is allocated at 0x7ffee117e7cb
s is allocated at 0x7ffee117e7c8
l is allocated at 0x7ffee117e7c0
CC>
```

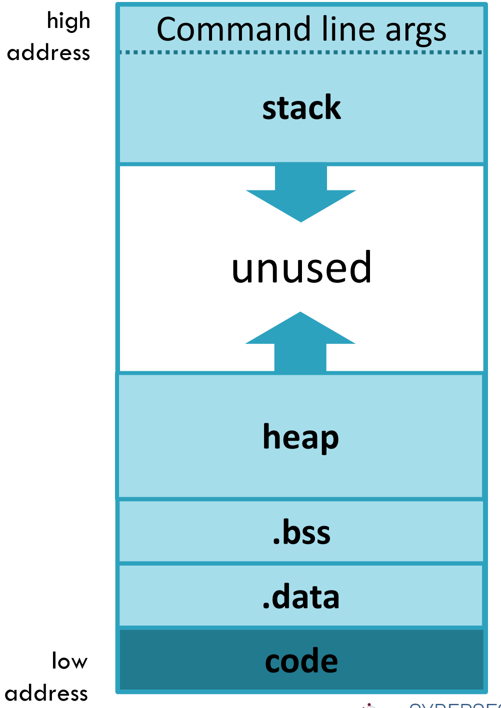
```
CC> gcc -o alignment alignment.c -O2
CC> ./alignment
i is allocated at 0x7ffee4dbb7c8
c is allocated at 0x7ffee4dbb7cf
s is allocated at 0x7ffee4dbb7cc
l is allocated at 0x7ffee4dbb7c0
CC>
```

Data alignment

- Compilers may introduce padding or change the order of data in memory
- There are trade-offs between speed and memory usage
- *C* compilers provide many optional optimization

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*



The stack

- 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;
  }
}
```

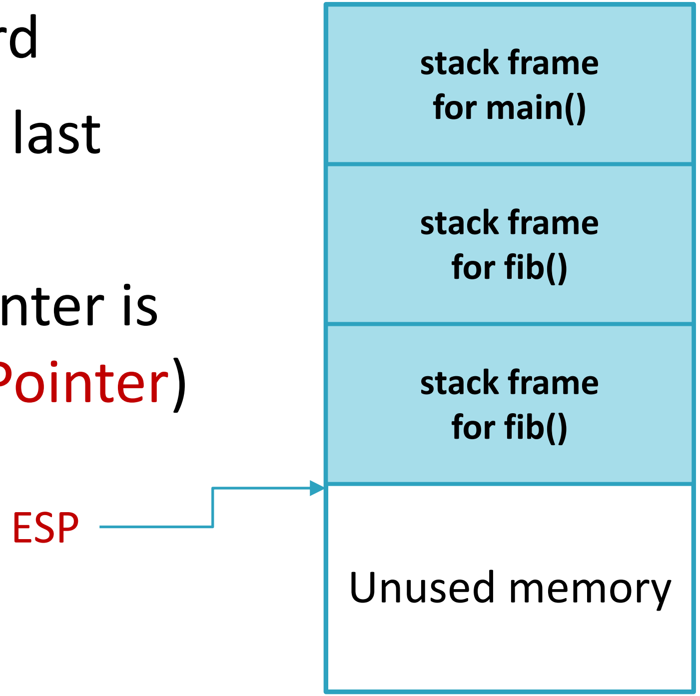
stack frame for main()
stack frame for fib()
stack frame for fib()
Unused memory

The stack

- The precise structure and organization of the stack depends on system architecture, operating system, and compilers we are using.
- For the sake of simplicity, in this lecture we will focus on *x86* architectures (32 and 64 bits) and on *gcc* compiler on *Linux*
- More detailed information are available at the following links:
  - https://docs.microsoft.com/en-us/cpp/cpp/argument-passing-and-naming-conventions
  - http://refspecs.linuxbase.org/

The stack

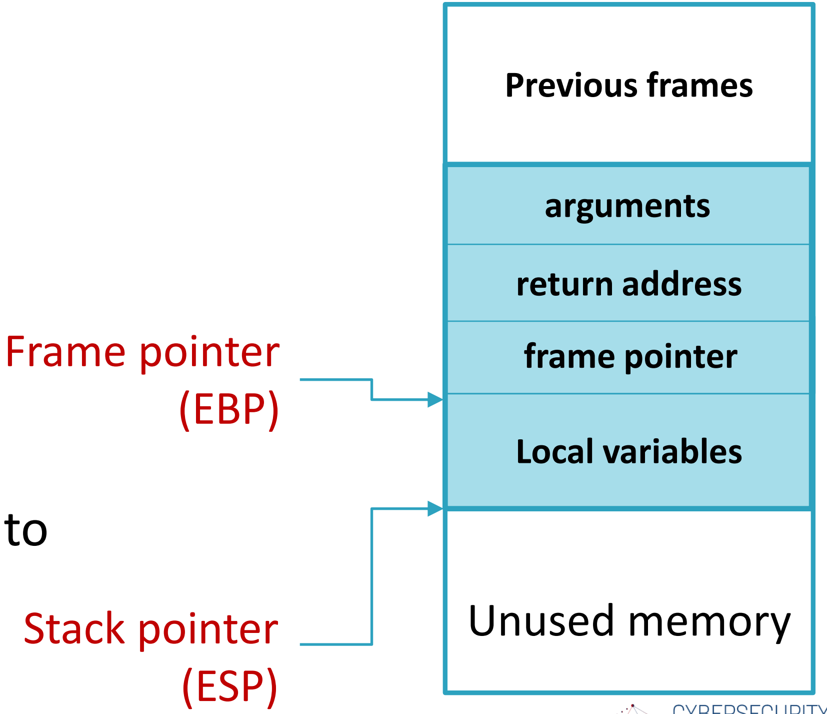
- Typically, the stack grows downward
- The stack pointer (SP) refers to the last element on the stack
- On *x86* architectures, the stack pointer is stored in the ESP (Extended Stack Pointer) register



Stack frame (for x86)

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

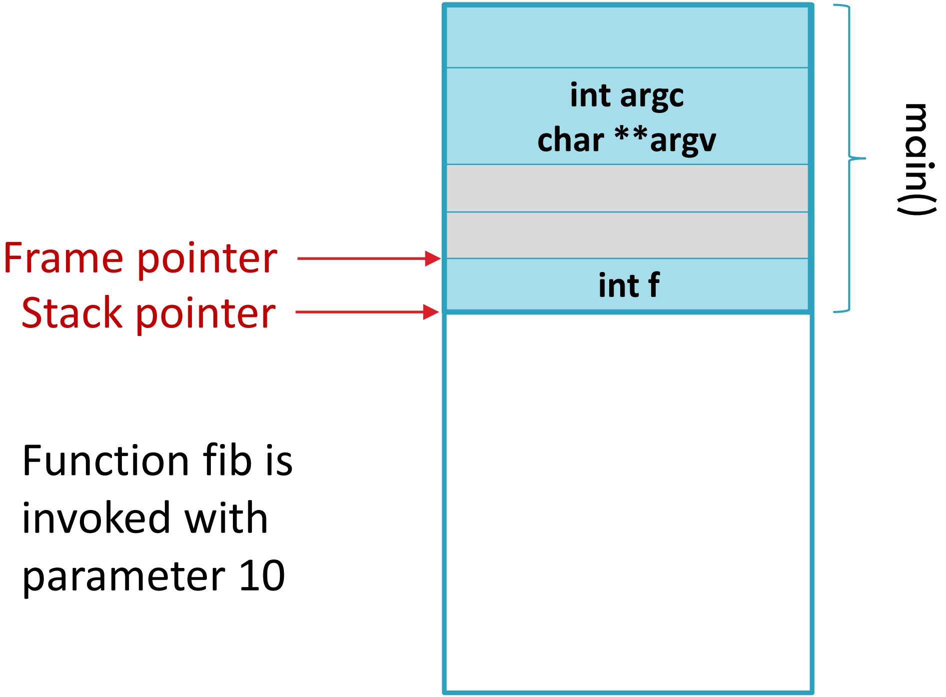
- Frame pointer, named Extended Base Pointer (EBP), provides a starting point to local variables



Stack frame: 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;
  }
}
```



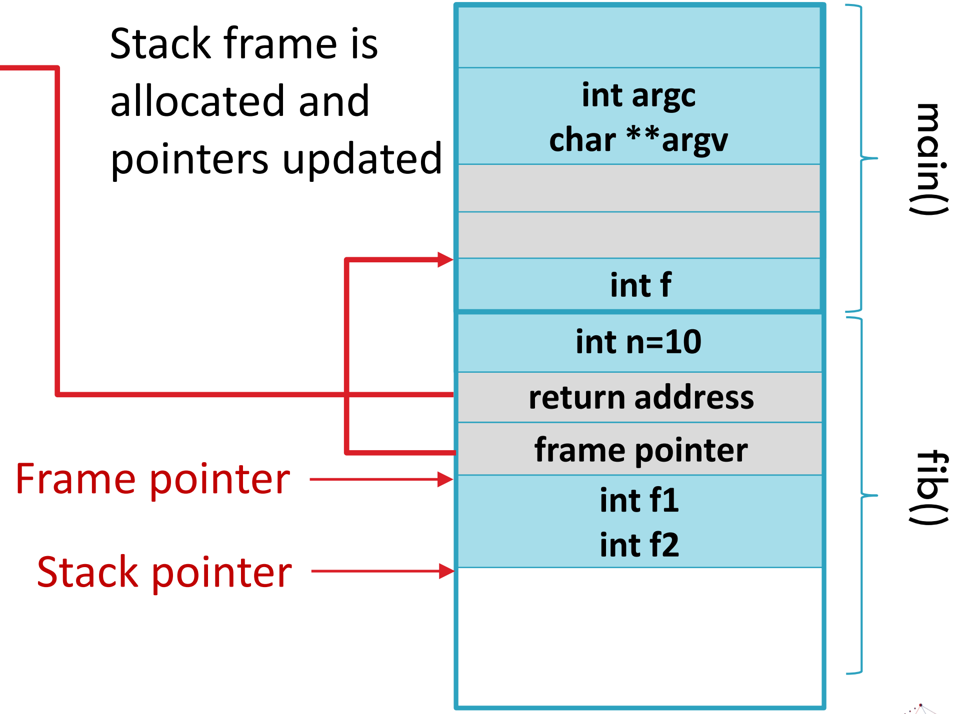
Function fib is invoked with parameter 10

Stack frame: 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 frame is allocated and pointers updated



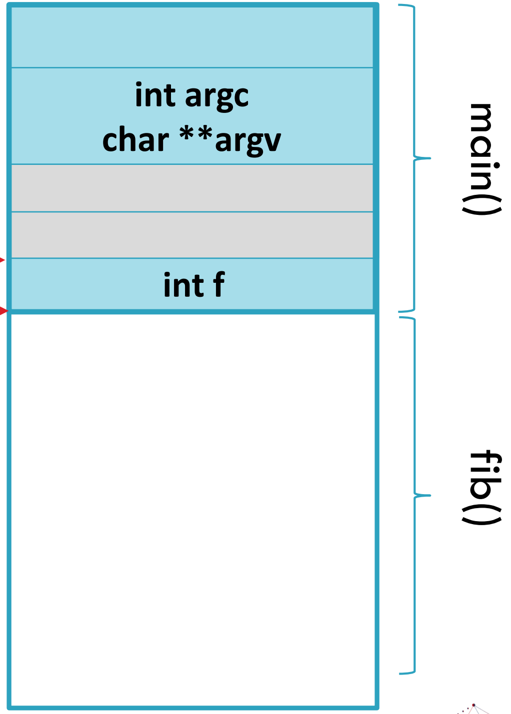
Stack frame: example


```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;
  }
}
```

Frame pointer

Stack pointer



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

Stack frame (for x86-64)

- In the *64-bit* version of *x86:*
  - Registers are extended to 64 bits
  - A new set of 8 registers is added
- The arguments are not all in the stack:
  - The first 6 are passed via registers
  - The remaining are placed in the stack (like for *x86*)
- Pointers *EBP* and *ESP* are named *RBP* and *RSP*
- A *red zone* of 128 bytes is placed in the stack just under RSP
  - This can be used to store extra local variables and is not modified by interrupt/exception/signal handlers

C declaration: cdecl

- The cdecl (C declaration) is a calling convention used in many compilers for *x86*
  - A calling convention describes how functions receive their parameters from caller and how they return they results
- In cdecl arguments are passed on the stack, while values are returned via registers
- Function arguments are pushed on the stack in the *right-to-left order*
- The caller *cleans* the stack after the RETURN from the called function

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

Stack and Heap pointers

- We have always to preserve reference to allocated areas in the heap
  - Otherwise, we cannot use them!
- These references (pointers) can be stored in the stack or in the heap
- We could store pointers to the stack in the heap
  - This can be dangerous! Referenced memory could be released!

Outline

- From Code to Programs
- Executable and Linkable Format (ELF)
- x86 architectures
- Memory management and Calling conventions
- Debugging

Debugging

- Debugging is the process of finding and fixing bugs (defects or problems that prevent correct operation) within computer programs, software, or systems.
- Debugging can be based on different techniques and methodologies such as:
  - interactive debugging
  - control flow analysis
  - unit and integration testing
  - log file and output analysis
  - memory dumps
  - profiling
- Many programming languages and software development tools also offer programs to aid in debugging, known as *debuggers*

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 (optional) data that debugger can use for a more informative input
- In the case of *gcc* compiler, the parameter –*g* can be used
- Debugging information is stored in specific sections of ELF file:
  - .debug, containing info for symbolic debugging
  - .line, containing line number information

Example


```
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
```

Michele LORETI

Università di Camerino

Basic background

CYBER CHALLENGE.IT

CYBERSECURITY NATIONAL LABORATORY

https://cybersecnatlab.it