Software Security

University of Insubria

From code to programs

Compiling a C program is a multi-stage process

- **preprocessing**: preprocessor commands are interpreted
- **compilation**: preprocessed code is translated into assembly instructions
- **assembly**: assembly instructions are translated into machine or object code
- **linking**: object code are combined in a single executable (links to the used library are) added

Linking

- **Static**: binaries are self-contained and do not depend on any external libraries
- **Dynamic**: binaries rely on system libraries that are loaded when needed (dynamic code relocation)

UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

ELF - Executable and Linkable Format

The Executable and Linkable Format (ELF) is a common file format for object files. Three types of object files

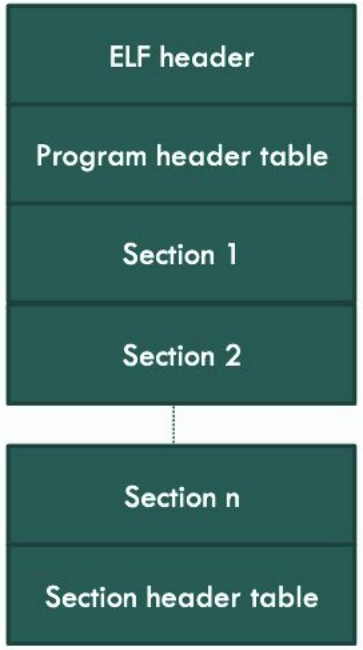
- **Relocatable file**: code and data that can be linked with other object files
- **Executable files**: program suitable for execution
- **Shared object files**
  - linked with other relocatable and shared object files
  - used by a dynamic linker to create a process image

UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

ELF - 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, ..)
- a Section header table with a description of previous sections



UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

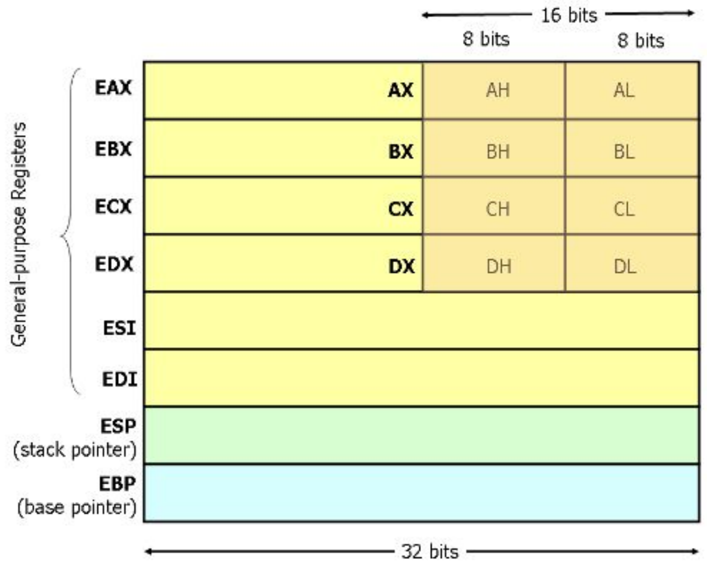
ELF - Executable and Linkable Format

- **.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

X86-32 Registers

x86-32 processors have eight 32-bit general purpose registers

- **EAX**: the accumulator (arithmetic operations)
- **ECX**: counter (loop index)
- **ESP**: stack pointer
- **EBP**: base pointer



UNIVERSITÀ DEGLI STUDI
DELL'INSUBRIA

x86 Instructions

Data Movement

- **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

Arithmetic and Logic

- **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** / ..

UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

x86 Instructions

Control Flow

- **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

Memory management

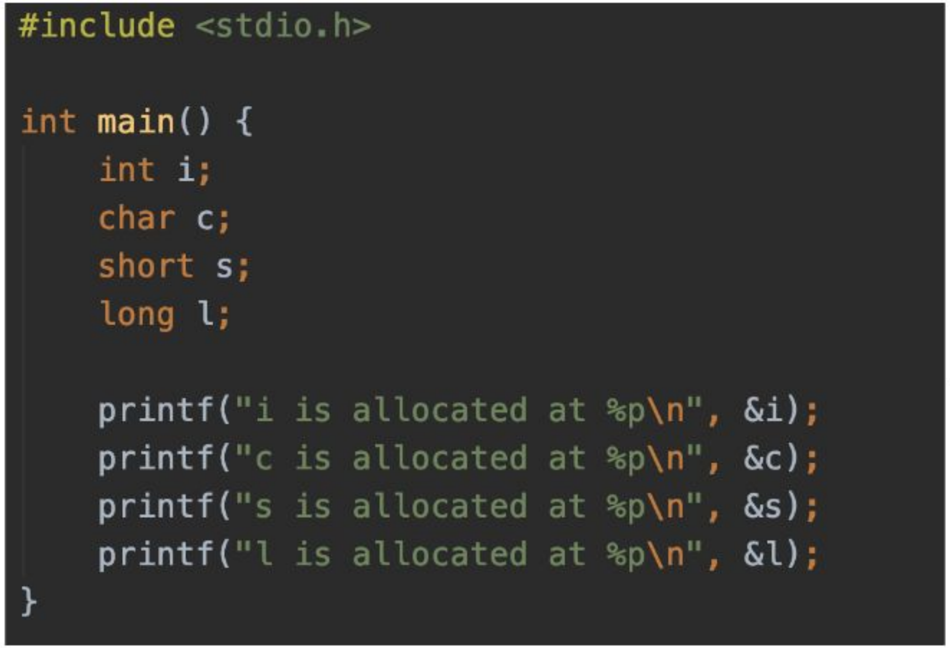
Modern programming languages allow programmers to use data types without having to know how they are represented. Programmers often ignore where data is allocated (compiler makes decisions).

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 (if x is a variable, &x denotes the pointer to x)

UNIVERSITÀ DEGLI STUDI
DELL'INSUBRIA

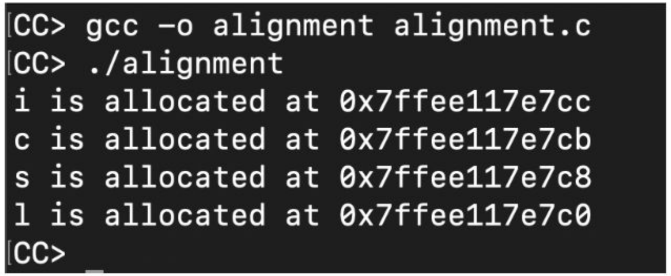
Memory allocation



```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);
}
```

C program:

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

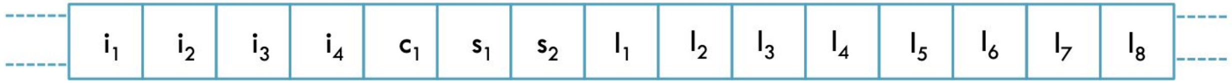


```
CC> gcc -o alignment alignment.c
CC> ./alignment
i is allocated at 0x7ffee117e7cc
c is allocated at 0x7ffee117e7cb
s is allocated at 0x7ffee117e7c8
l is allocated at 0x7ffee117e7c0
CC>
```

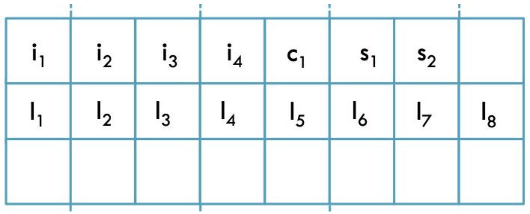
UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

Data alignment

Memory is usually represented as a sequence of bytes



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



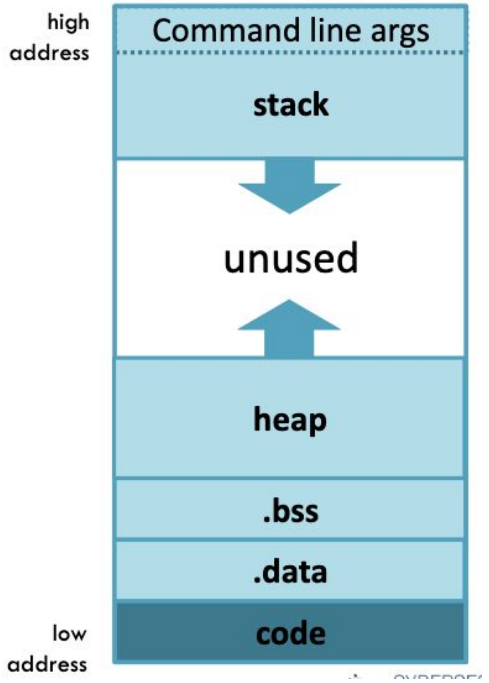
Compilers may introduce padding or change the order of data in memory (optional optimization)

Memory segments

Memory is allocated for each process (a running program) to store data and code.

Different segments

- **Stack**: for local variables
- **Heap**: for dynamic memory
- **Data segment**
  - global uninitialized variables (.bss)
  - global initialized variables (.data)
- **Code segment**



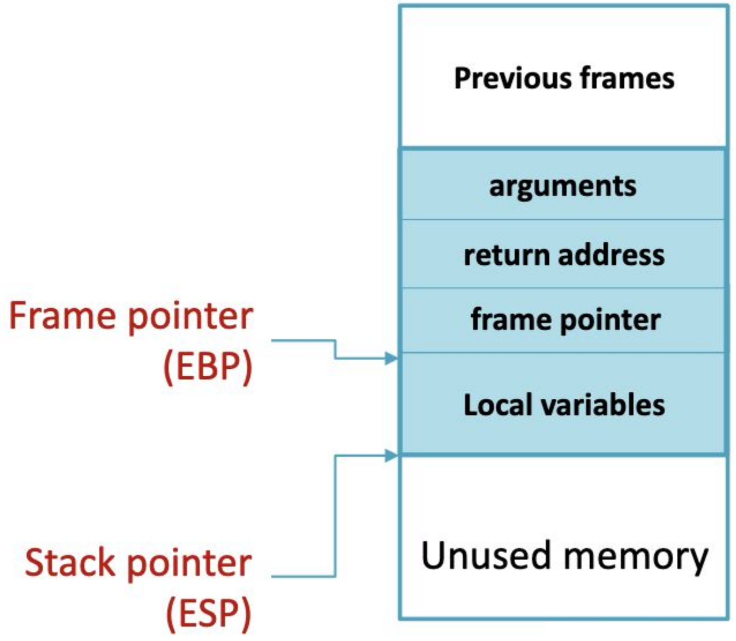
UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

The stack

The stack consists of a sequence of stack frames, each for each function call (allocated on call, de-allocated on return).

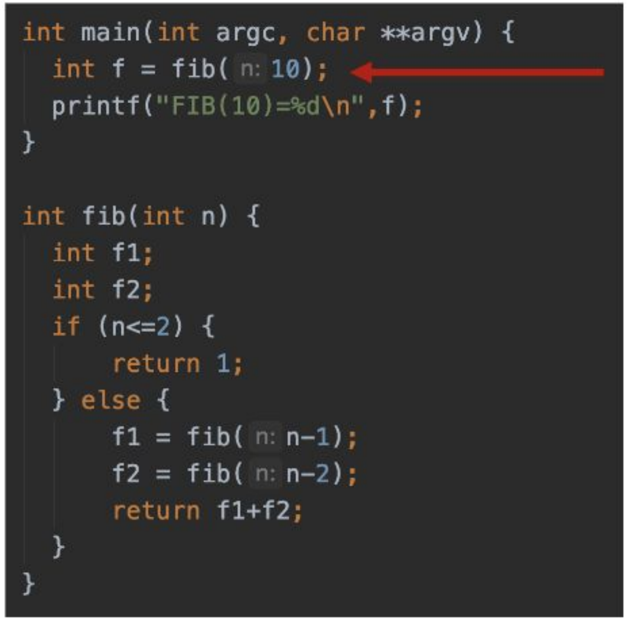
Pointers:

- **stack pointer (SP)**: refers to the last element on the stack
- **frame pointer (FP)**: provides a starting point to local variables

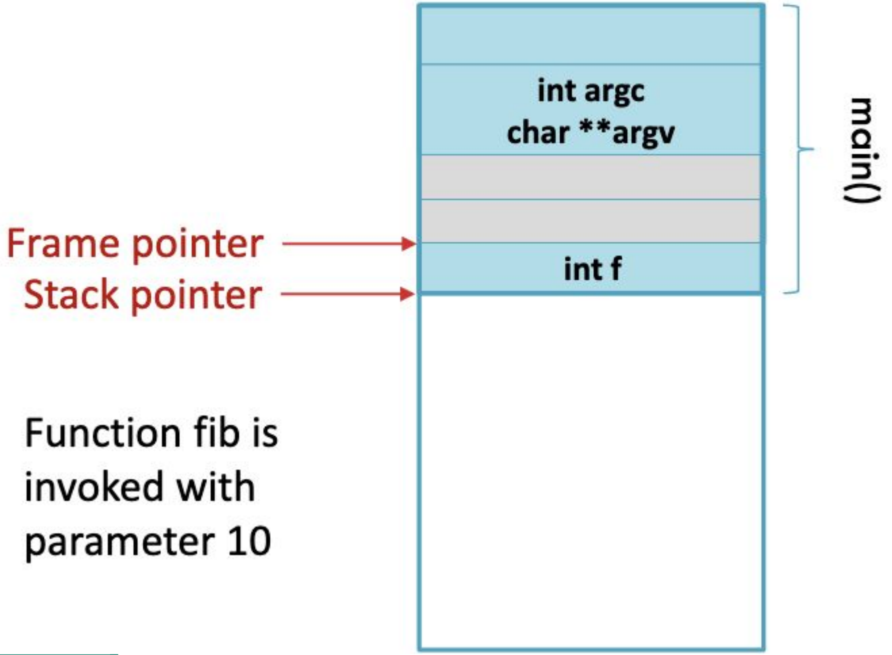


UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

Stack frame

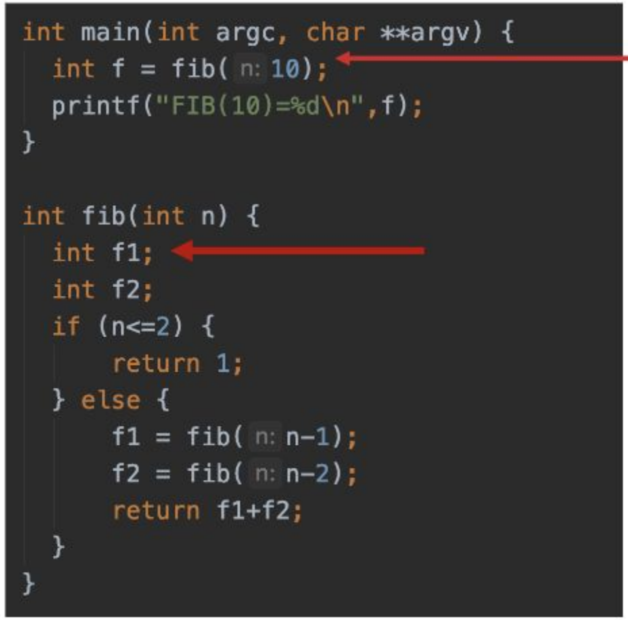


```
int main(int argc, char **argv) {
  int f = fib( n: 10);
  printf("FIB(10)=%d\n",f);
}

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;
  }
}
```

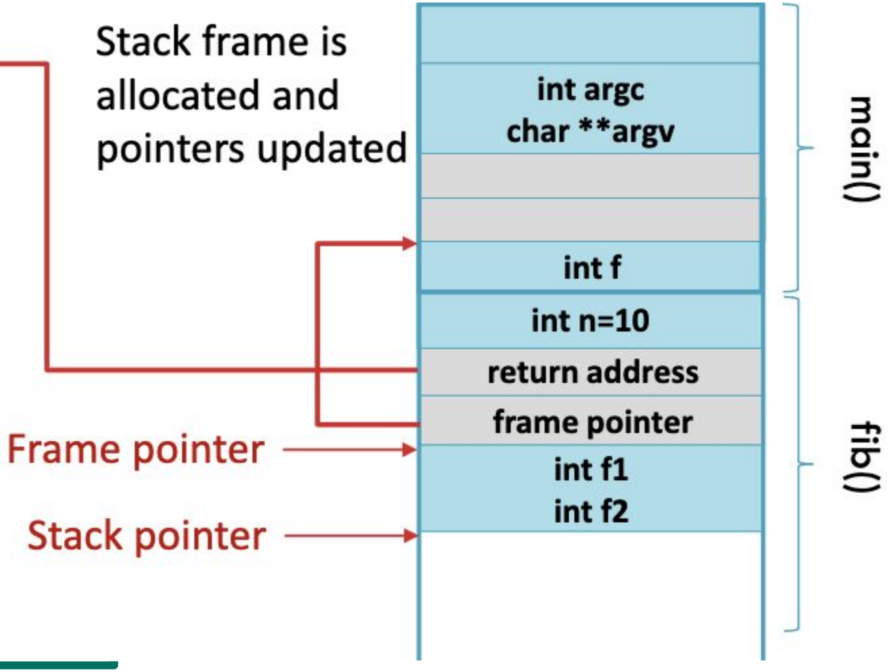


Function fib is invoked with parameter 10

Stack frame

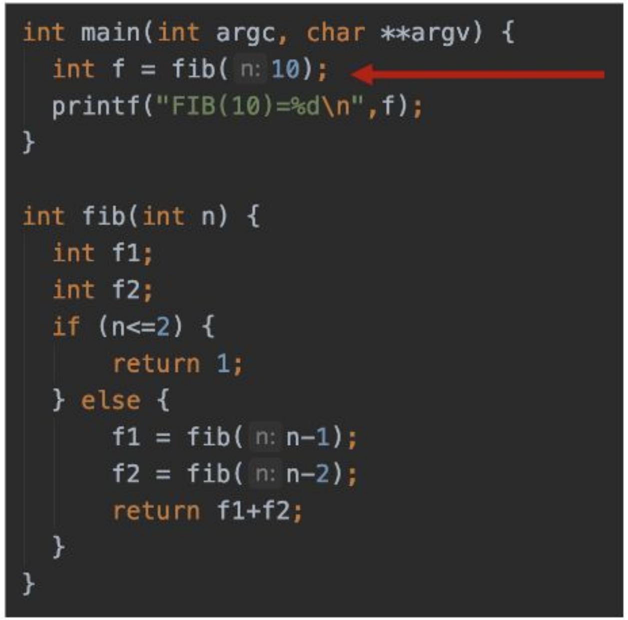


```
int main(int argc, char **argv) {
  int f = fib( n: 10);
  printf("FIB(10)=%d\n",f);
}

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;
  }
}
```

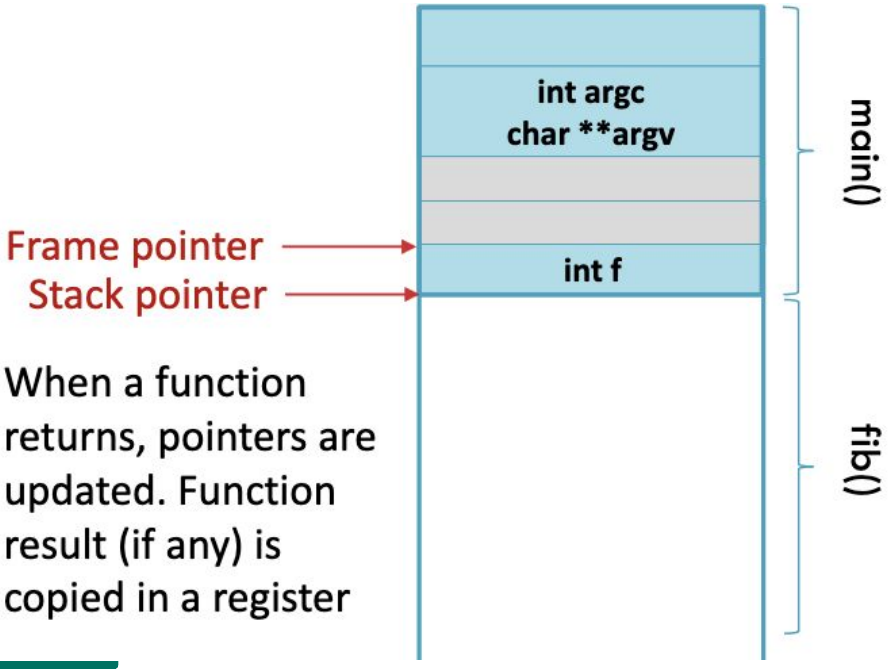


UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

Stack frame



```
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

Memory management functions

Basic C functions for memory management

- **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

Debugging is the process of finding and fixing bugs (defects or problems that prevent correct operation) within computer programs, software, or systems.

- compilers can be instrumented to emit extra (optional) data that debugger can use for a more informative input (-g for gcc)
- debugging information is stored in specific sections of ELF file
  - .debug: containing info for symbolic debugging
  - .line: containing line number information

UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

Tools

UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

Binary file

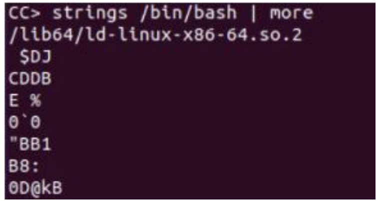
With a binary file it is possible to

- check if the file is executable or not
- discover the architecture for which the binary has been compiled
- collect symbols and strings used in the program
- check if there is a running process associated with the binary
- read the SHA of a file and check if it is associated with some malicious software
- identify function names and used libraries

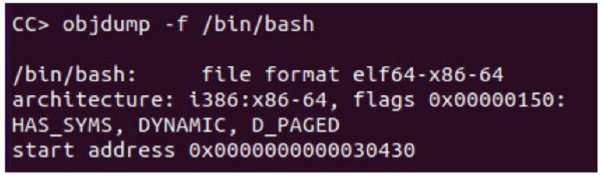
Reading ELF data

Tools to extract information from an ELF file

- **strings**: permits collecting all the strings occurring in a binary file
- **objdump**: displays information about one or more object files
  - info about the section headers
  - content of specific sections
  - disassemble binaries
- **readelf**: info at the header file



```
CC> strings /bin/bash | more
/lib64/ld-linux-x86-64.so.2
 $DJ
CDDB
E %
0`0
"BB1
B8:
0D@kB
```



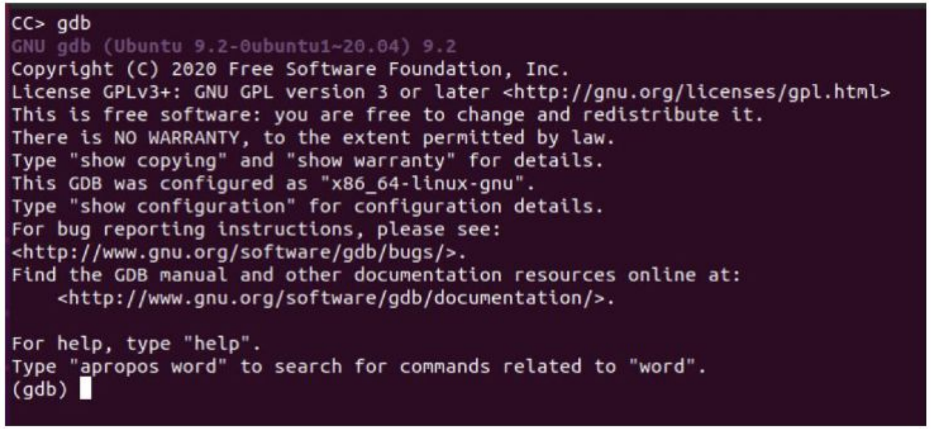
```
CC> objdump -f /bin/bash

/bin/bash:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x0000000000030430
```

UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

GDB - The GNU Project Debugger

GDB is a debugging tool to

- Start and stop a program
- Examine what happened, when a program stopped
- Change memory o registers content while a program is running

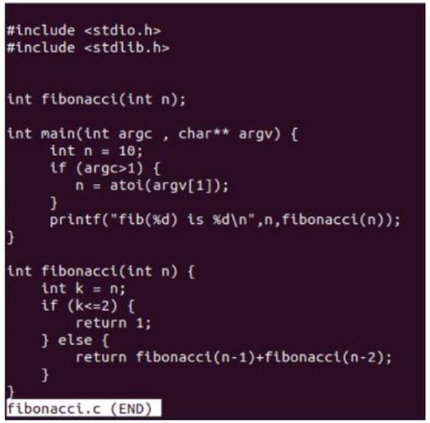


```
CC> gdb
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

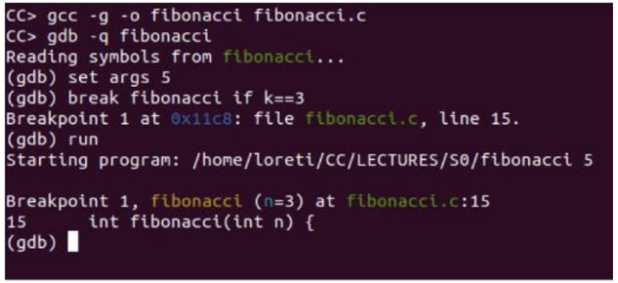
GDB - The GNU Project Debugger

A breakpoint makes your program stop whenever a certain point in the program is reached

- **watchpoint**: stops a program when the value of an expression changes
- **catchpoint**: stops your program when a certain kind of event occurs



```
#include <stdio.h>
#include <stdlib.h>


int fibonacci(int n);

int main(int argc , char** argv) {
     int n = 10;
     if (argc>1) {
        n = atoi(argv[1]);
     }
     printf("fib(%d) is %d\n",n,fibonacci(n));
}

int fibonacci(int n) {
    int k = n;
    if (k<=2) {
        return 1;
    } else {
        return fibonacci(n-1)+fibonacci(n-2);
    }
}
fibonacci.c (END)
```



```
CC> gcc -g -o fibonacci fibonacci.c
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) set args 5
(gdb) break fibonacci if k==3
Breakpoint 1 at 0x11c8: file fibonacci.c, line 15.
(gdb) run
Starting program: /home/loreti/CC/LECTURES/S0/fibonacci 5

Breakpoint 1, fibonacci (n=3) at fibonacci.c:15
15      int fibonacci(int n) {
(gdb) 
```

GDB - The GNU Project Debugger

When a program stops at a breakpoint, its execution can be resumed exploiting 2 functionalities

- **Continuing** means resuming program execution until your program completes normally
- **Stepping** means executing just one more “step” of your program

UNIVERSITÀ DEGLI STUDI
DELL'INSUBRIA

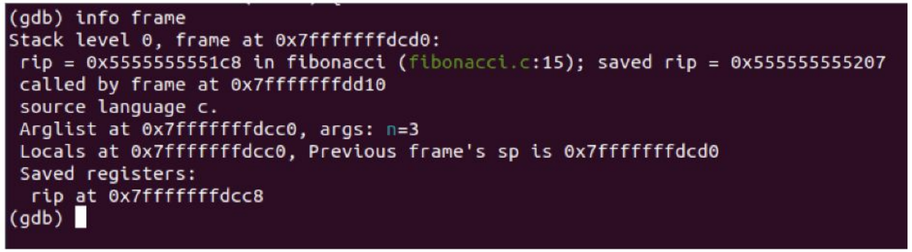
GDB - The GNU Project Debugger

When your program has stopped, the first thing we need to know is where it stopped and how it got there

- 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 found

- the location of the call in a program
- the arguments of the call
- the local variables of the function being called

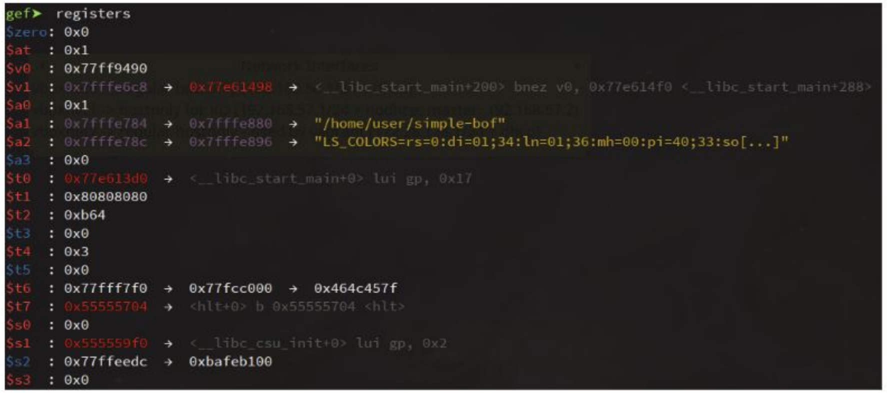


```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdcd0:
 rip = 0x5555555551c8 in fibonacci (fibonacci.c:15); saved rip = 0x555555555207
 called by frame at 0x7fffffffdd10
 source language c.
 Arglist at 0x7fffffffdcc0, args: n=3
 Locals at 0x7fffffffdcc0, Previous frame's sp is 0x7fffffffdcd0
 Saved registers:
  rip at 0x7fffffffdcc8
(gdb) 
```

GEF - GDB Enhanced Features

GEF consists of a set of commands that extends GDB with additional features for dynamic analysis and exploit development

- Embedded hexdump view
- Automatic dereferencing of data and registers
- Heap analysis
- Display ELF information



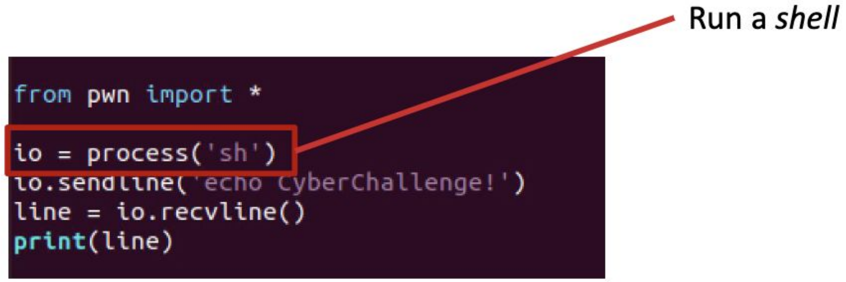
```
gef➤  registers
$zero: 0x0
$at  : 0x1
$v0  : 0x77ff9490
$v1  : 0x7fffe6c8  →  0x77e61498  →  <__libc_start_main+200> bnez v0, 0x77e614f0 <__libc_start_main+288>
$a0  : 0x1
$a1  : 0x7fffe784  →  0x7fffe880  →  "/home/user/simple-bof"
$a2  : 0x7fffe78c  →  0x7fffe896  →  "LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so[...]"
$a3  : 0x0
$t0  : 0x77e613d0  →  <__libc_start_main+0> lui gp, 0x17
$t1  : 0x80808080
$t2  : 0xb64
$t3  : 0x0
$t4  : 0x3
$t5  : 0x0
$t6  : 0x77fff7f0  →  0x77fcc000  →  0x464c457f
$t7  : 0x55555704  →  <hlt+0> b 0x55555704 <hlt>
$s0  : 0x0
$s1  : 0x555559f0  →  <__libc_csu_init+0> lui gp, 0x2
$s2  : 0x77ffeedc  →  0xbafeb100
$s3  : 0x0
```

UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

Pwntools

Tubes are I/O wrappers supporting the main type of connections

- Local processes
- Remote TCP or UDP connections
- Process running on a remote server over SSH
- Serial port I/O



```
from pwn import *

io = process('sh')
io.sendline('echo CyberChallenge!')
line = io.recvline()
print(line)
```

Other tools

Radare2: toolchain for easing task like

- Software Reverse Engineering
- Exploiting
- Debugging

Ghidra: is a Software Reverse Engineering (SRE) tool including

- an interactive disassembler and decompiler
- a set of tools to support code analysis