# Software and Tools

**Michele LORETI**
Università di Camerino

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



- In this lecture we will present the main tools that will be used in the forthcoming modules of Software Security.

# Prerequisites



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

# Outline



- Reading ELF data
- GDB: The GNU Project Debugger
  - GEF: GDB Enhanced Features
- Pwntools
- Other tools:
  - Radare2
  - Ghidra

# Gathering info from binary files



Given a binary file we can…

- 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

# Outline



- Reading ELF data
- GDB: The GNU Project Debugger
  - GEF: GDB Enhanced Features
- Pwntools
- Other tools:
  - Radare2
  - Ghidra

# Gathering info from binary files



- Several tools are available to extract information from an ELF file, such as:
  - strings
  - objdump
  - readelf

# Gathering info from binary files



- Several tools are available to extract information from an ELF file, such as:
  - strings
  - objdump
  - readelf

# Strings



- This is a simple *terminal tool* that permits collecting all the *strings* occurring in a binary file
- For each given file, *strings* print the *printable character sequences* that are *at least 4 characters long* and are followed by an *unprintable character*
- Collected strings can give us info about *secrets* and used data

# Strings (an example)



- In a Linux system, we can use *strings* to collect data form */bin/bash* the *Bash shell:*

```
CC> strings /bin/bash | more
/lib64/ld-linux-x86-64.so.2
 $DJ
CDDB
E %
0`0
"BB1
B8:
0D@kB
 9E4
NR l
 "7$aD
`% H0A
Hap5
@ E<
($B
d> 7
`       0
A!I`
R       2(
        !C)
&51H
```

# Strings (some options)



- Relevant options are:
  - *–d* (or *--data*), strings are collected only from *data sections*
  - *-n <num>* (or *--bytes=<num>*), prints only strings with *<num>* characters (by default 4)
  - *-h* (or *--help*), prints program help

# Gathering info from binary files



- Several tools are available to extract information from an ELF file, such as:
  - strings
  - objdump
  - readelf

# objdump



- …displays information about one or more object files, such as:
  - information from the overall header of each of the *objfile* files:

```
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
```

# objdump



- information from the section headers of the object file (option *–h*):

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

/bin/bash:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000318  0000000000000318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.property 00000020  0000000000000338  0000000000000338  00000338  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000000358  0000000000000358  00000358  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .note.ABI-tag 00000020  000000000000037c  000000000000037c  0000037c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .gnu.hash     00004aac  00000000000003a0  00000000000003a0  000003a0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynsym       0000e418  0000000000004e50  0000000000004e50  00004e50  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .dynstr       00009740  0000000000013268  0000000000013268  00013268  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version  00001302  0000000000001c9a8  0000000000001c9a8  0001c9a8  2**1
```

# objdump



- Content of specific sections:

```
CC> objdump -s -j .rodata /bin/bash | more

/bin/bash:     file format elf64-x86-64

Contents of section .rodata:
 de000 01000200 474e5520 62617368 2c207665  ....GNU bash, ve
 de010 7273696f 6e202573 2d282573 290a0078  rsion %s-(%s)..x
 de020 38365f36 342d7063 2d6c696e 75782d67  86_64-pc-linux-g
 de030 6e750047 4e55206c 6f6e6720 6f707469  nu.GNU long opti
 de040 6f6e733a 0a00092d 2d25730a 00536865  ons:...--%s..She
 de050 6c6c206f 7074696f 6e733a0a 00092d25  ll options:...-%
 de060 73206f72 202d6f20 6f707469 6f6e0a00  s or -o option..
 de070 72756e5f 6f6e655f 636f6d6d 616e6400  run_one_command.
 de080 2d630072 62617368 00492068 61766520  -c.rbash.I have
 de090 6e6f206e 616d6521 003f3f68 6f73743f  no name!.??host?
 de0a0 3f004241 53485f45 4e560050 4f534958  ?.BASH_ENV.POSIX
 de0b0 4c595f43 4f525245 43540050 4f534958  LY_CORRECT.POSIX
 de0c0 5f504544 414e5449 43005c73 2d5c765c  _PEDANTIC.\s-\v\
 de0d0 2420003e 20007e2f 2e626173 68726300  $ .> .~/.bashrc.
 de0e0 25733a20 696e7661 6c696420 6f707469  %s: invalid opti
 de0f0 6f6e0025 6325633a 20696e76 616c6964  on.%c%c: invalid
 de100 206f7074 696f6e00 6c6f6769 6e5f7368   option.login_sh
```

# objdump



- This tool can be also used to disassemble binaries:

```
CC> objdump -D ./aprogram | grep "main."
    1081:       48 8d 3d c1 00 00 00    lea    0xc1(%rip),%rdi        # 1149 <main>
    1088:       ff 15 52 2f 00 00       callq  *0x2f52(%rip)        # 3fe0 <__libc_start_main@GLIBC_2.2.5>
0000000000001149 <main>:
CC>
```

- A detailed list of functionalities can be obtained by invoking the program with options *–H* (or *--help*)

# Gathering info from binary files



- Several tools are available to extract information from an ELF file, such as:
  - strings
  - objdump
  - readelf

# readelf



- Main info displayed with *readelf are:*
  - info at the header file (option -h)

```
CC> readelf -h /bin/bash
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x30430
  Start of program headers:          64 (bytes into file)
  Start of section headers:          1181528 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

# readelf



- info available in the segment headers (option –*l*):

```
CC> readelf -l /bin/bash

Elf file type is DYN (Shared object file)
Entry point 0x30430
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x000000000002ce70 0x000000000002ce70  R      0x1000
  LOAD           0x000000000002d000 0x000000000002d000 0x000000000002d000
                 0x00000000000b0705 0x00000000000b0705  R E    0x1000
  LOAD           0x00000000000de000 0x00000000000de000 0x00000000000de000
                 0x0000000000036198 0x0000000000036198  R      0x1000
  LOAD           0x0000000000114cf0 0x0000000000115cf0 0x0000000000115cf0
```

# readelf



- The entries in symbol table section of the file:

```
CC> readelf -s /bin/bash | more

Symbol table '.dynsym' contains 2433 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND endgrent@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __ctype_toupper_loc@GLIBC_2.3 (3)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND iswlower@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND sigprocmask@GLIBC_2.2.5 (2)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __snprintf_chk@GLIBC_2.3.4 (4)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND free@GLIBC_2.2.5 (2)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND getservent@GLIBC_2.2.5 (2)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND wcscmp@GLIBC_2.2.5 (2)
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND putchar@GLIBC_2.2.5 (2)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND tputs@NCURSES6_TINFO_5.0.19991023 (5)
    11: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND strcasecmp@GLIBC_2.2.5 (2)
    12: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND localtime@GLIBC_2.2.5 (2)
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND mblen@GLIBC_2.2.5 (2)
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __vfprintf_chk@GLIBC_2.3.4 (4)
    15: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND abort@GLIBC_2.2.5 (2)
    16: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __errno_location@GLIBC_2.2.5 (2)
    17: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND strncpy@GLIBC_2.2.5 (2)
```

# Outline



- Reading ELF data
- GDB: The GNU Project Debugger
  - GEF: GDB Enhanced Features
- Pwntools
- Other tools:
  - Radare2
  - Ghidra

# 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*

```
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



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

```
gdb <program>
```

- 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
  - *gdb* can be executed in different *modes:*
    - batch: executes a list of commands and waits for termination
    - quite: does not print introductory and copyright messages
- Load configuration files
  - Defines the behaviour of gdb at startup
  - Can be global at the system level or local to the current directory
  - Details are available at the gdb documentation page
- Load symbols of debugged program
- Waits for user commands

# GDB: Commands



- 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:*

```
CC> gdb -q aprogram
Reading symbols from aprogram...
(gdb) run
Starting program: /home/loreti/CC/LECTURES/S0/aprogram
Hello World!
[Inferior 1 (process 36198) exited normally]
(gdb)
```

# GDB: Commands



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

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands.
breakpoints -- Making program stop at certain points.
data -- Examining data.
files -- Specifying and examining files.
internals -- Maintenance commands.
obscure -- Obscure features.
running -- Running the program.
stack -- Examining the stack.
status -- Status inquiries.
support -- Support facilities.
tracepoints -- Tracing of program execution without stopping the program.
user-defined -- User-defined commands.

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

# GDB: Commands



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

```
CC> gdb -q aprogram
Reading symbols from aprogram...
(gdb) set args 10 test /usr/bin
(gdb) show args
Argument list to give program being debugged when it is started is "10 test /usr/bin".
(gdb)
```

# GDB: Commands



- The *environment* consists of a set of *environment variables* storing info such as *user name*, *home directory*, *terminal type*, or the search *path* for programs to run.
- Environment variables can be accessed/changed within *gdb* via the following commands:
  - *path <directory>* — add the directory to the path
  - *show paths* — show the content of *path*
  - *show environment [<var>]* — show the environment (or the specific variable)
  - *set environment <var> <value>* — set value for the given variable
  - *unset environment <var>* — delete the given variable from the environment

# GDB: Stopping and Continuing



- 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
  - *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 function name
  - An address

# GDB: Example



- Let us consider the following simple c program:

```
#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)
```

# GDB: Example

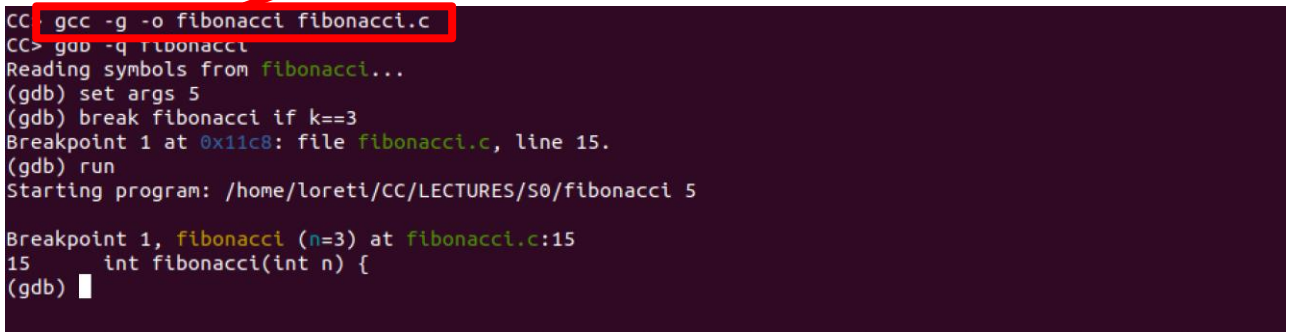


```
CC> gcc -g -o fibonacci fibonacci.c
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) set args 5
(gdb) break fibonacci if k==3
Breakpoint 1 at 0x11c8: file fibonacci.c, line 15.
(gdb) run
Starting program: /home/loreti/CC/LECTURES/S0/fibonacci 5

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

# GDB: Example

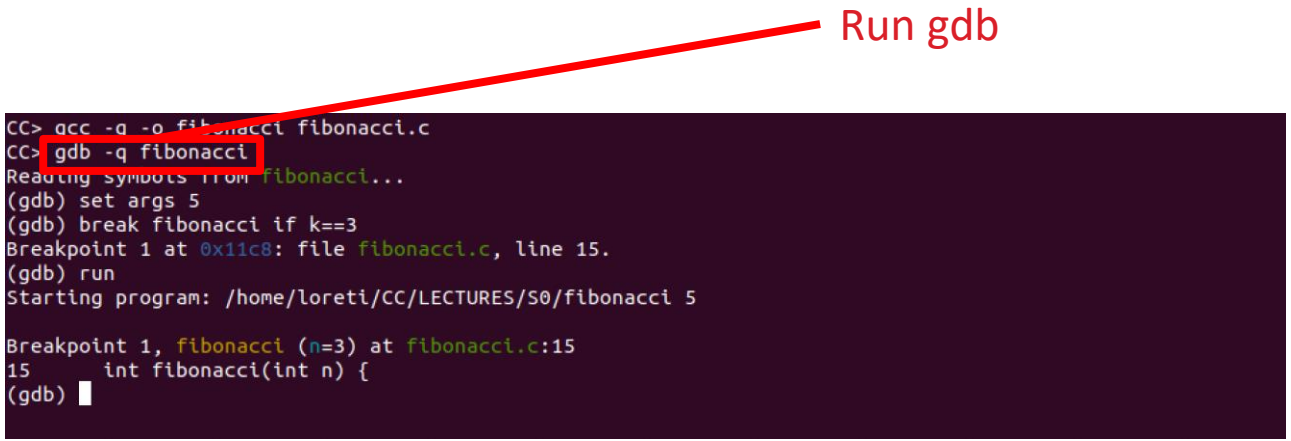


We compile it for debugging



# GDB: Example





# GDB: Example



Set program arguments

```
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: Example



```
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 0x11ca: 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)
```

Add a conditional breakpoint

# GDB: Example



```
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)
```

Run our program



# GDB: Example



```
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)
```

Computation is stopped when the breakpoint with the specific condition is reached



# GDB: Continue and Stepping



- 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
    - A step can be either an instruction of source code, or an instructions of the assembly

# GDB: Example



```
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) break main
Breakpoint 1 at 0x1169: file fibonacci.c, line 7.
(gdb) run
Starting program: /home/loreti/CC/LECTURES/S0/fibonacci

Breakpoint 1, main (argc=21845, argv=0x0) at fibonacci.c:7
7       int main(int argc , char** argv) {
(gdb) step
8               int n = 10;
(gdb) step
9               if (argc>1) {
(gdb) continue
Continuing.
fib(10) is 55
[Inferior 1 (process 36526) exited normally]
(gdb)
```

# GDB: Example



```
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) break main
Breakpoint 1 at 0x1169: file fibonacci.c, line 7.
(gdb) run
Starting program: /home/loreti/CC/LECTURES/S0/fibonacci

Breakpoint 1, main (argc=21845, argv=0x0) at fibonacci.c:7
7       int main(int argc , char** argv) {
(gdb) step
8               int n = 10;
(gdb) step
9               if (argc>1) {
(gdb) continue
Continuing.
fib(10) is 55
[Inferior 1 (process 36526) exited normally]
(gdb)
```

A single *step* is executed, *gdb* prints the *next* instruction

# GDB: Example



```
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) break main
Breakpoint 1 at 0x1169: file fibonacci.c, line 7.
(gdb) run
Starting program: /home/loreti/CC/LECTURES/S0/fibonacci

Breakpoint 1, main (argc=21845, argv=0x0) at fibonacci.c:7
7       int main(int argc , char** argv) {
(gdb) step
8           int n = 10;
(gdb) step
9           if (argc>1) {
(gdb) continue
Continuing.
fib(10) is 55
[Inferior 1 (process 36526) exited normally]
(gdb)
```

We can step again…

# GDB: Example



```
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) break main
Breakpoint 1 at 0x1169: file fibonacci.c, line 7.
(gdb) run
Starting program: /home/loreti/CC/LECTURES/S0/fibonacci

Breakpoint 1, main (argc=21845, argv=0x0) at fibonacci.c:7
7       int main(int argc , char** argv) {
(gdb) step
8               int n = 10;
(gdb) step
9               if (argc>1) {
(gdb) continue
Continuing.
fib(10) is 55
[Inferior 1 (process 36526) exited normally]
(gdb)
```

…or continue until the program terminates or another breakpoint is reached.

# 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 found:
  - the location of the call in your program
  - the arguments of the call
  - the local variables of the function being called

# GDB: Inspecting the stack



- 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: Example



```
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(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

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

# GDB: Example



```
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(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

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

Command *frame* is used to obtain a brief description of the current frame

# GDB: Example



```
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(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

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

Number o current frame

# GDB: Example

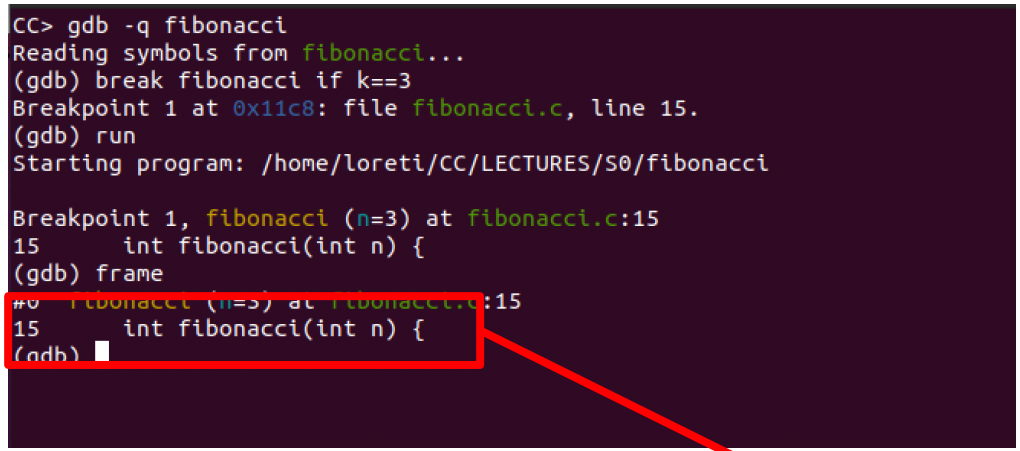


```
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) break fibonacci if k==3
Breakpoint 1 at 0x11c8: file fibonacci.c, line 15.
(gdb) run
Starting program: /home/loreti/CC/LECTURES/S0/fibonacci

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

Function name, its arguents and code line

# GDB: Example






Source code

# GDB: Example



- Command *info frame* permits getting detailed info about the current stack frame:

```
(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)
```

# GDB: Disassembly



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

```
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) break main
Breakpoint 1 at 0x1169: file fibonacci.c, line 7.
(gdb) run
Starting program: /home/loreti/CC/LECTURES/S0/fibonacci

Breakpoint 1, main (argc=21845, argv=0x0) at fibonacci.c:7
7       int main(int argc , char** argv) {
(gdb) disas
Dump of assembler code for function main:
=> 0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x20,%rsp
   0x0000555555555175 <+12>:    mov    %edi,-0x14(%rbp)
   0x0000555555555178 <+15>:    mov    %rsi,-0x20(%rbp)
   0x000055555555517c <+19>:    movl   $0xa,-0x4(%rbp)
   0x0000555555555183 <+26>:    cmpl   $0x1,-0x14(%rbp)
   0x0000555555555187 <+30>:    jle    0x55555555519f <main+54>
   0x0000555555555189 <+32>:    mov    -0x20(%rbp),%rax
   0x000055555555518d <+36>:    add    $0x8,%rax
   0x0000555555555191 <+40>:    mov    (%rax),%rax
   0x0000555555555194 <+43>:    mov    %rax,%rdi
   0x0000555555555197 <+46>:    callq  0x555555555070 <atoi@plt>
   0x000055555555519c <+51>:    mov    %eax,-0x4(%rbp)
   0x000055555555519f <+54>:    mov    -0x4(%rbp),%eax
   0x00005555555551a2 <+57>:    mov    %eax,%edi
   0x00005555555551a4 <+59>:    callq  0x5555555551c8 <fibonacci>
   0x00005555555551a9 <+64>:    mov    %eax,%edx
   0x00005555555551ab <+66>:    mov    -0x4(%rbp),%eax
   0x00005555555551ae <+69>:    mov    %eax,%esi
   0x00005555555551b0 <+71>:    lea    0xe4d(%rip),%rdi        # 0x555555556004
--Type <RET> for more, q to quit, c to continue without paging--
```

# GDB: Examining data



- The usual way to examine data in your program is with the *print* command (abbreviated *p*), or its synonym *inspect*

```
CC> gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) break main
Breakpoint 1 at 0x1169: file fibonacci.c, line 7.
(gdb) run
Starting program: /home/loreti/CC/LECTURES/S0/fibonacci

Breakpoint 1, main (argc=21845, argv=0x0) at fibonacci.c:7
7       int main(int argc , char** argv) {
(gdb) print n
$1 = 0
(gdb) step
8               int n = 10;
(gdb) print n
$2 = 0
(gdb) step
9               if (argc>1) {
(gdb) print n
$3 = 10
(gdb)
```

# Outline



- Reading ELF data
- GDB: The GNU Project Debugger
  - GEF: GDB Enhanced Features
- Pwntools
- Other tools:
  - Radare2
  - Ghidra

# 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/

# GEF: GDB Enhanced Features



- GEF has been designed with the following constraints:
  - Simple and fast to install
  - No external dependency
    - Some extensions are available to increase available features
  - Compatible with both Python2 and Python3
  - Abstract from specific architecture
  - Extensible
  - Well documented

# GEF: GDB Enhanced Features



- Pretty printing of registers (with automatic dereferencing)…

```
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
```

# GEF: GDB Enhanced Features



- Info about the *heap chuck…*

```
gef➤  heap chunks
Chunk(addr=0x555555756010, size=0x250, flags=PREV_INUSE)
    [0x0000555555756010     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x555555756260, size=0x110, flags=PREV_INUSE)
    [0x0000555555756260     61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61    aaaaaaaaaaaaaaaa]
Chunk(addr=0x555555756370, size=0x110, flags=PREV_INUSE)
    [0x0000555555756370     62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62    bbbbbbbbbbbbbbbb]
Chunk(addr=0x555555756480, size=0x110, flags=PREV_INUSE)
    [0x0000555555756480     63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63    cccccccccccccccc]
Chunk(addr=0x555555756590, size=0x110, flags=PREV_INUSE)
    [0x0000555555756590     64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64    dddddddddddddddd]
Chunk(addr=0x5555557566a0, size=0x20970, flags=PREV_INUSE)  ←  top chunk
```

# GEF: GDB Enhanced Features



- Security info…

```
gef➤  checksec
[+] checksec for '/home/user/mips-stack-bof'
Canary                            : No
NX                                : No
PIE                               : Yes
Fortify                           : No
RelRO                             : No
gef➤
```



# GEF: GDB Enhanced Features



- Hexdump view of a memory range…

```
gef➤  db $sp
0x00007fffffffdfa0     60 62 75 55 55 55 00 00 70 63 75 55 55 55 00 00     `buUUU..pcuUUU..
0x00007fffffffdfb0     80 64 75 55 55 55 00 00 90 65 75 55 55 55 00 00     .duUUU...euUUU..
0x00007fffffffdfc0     30 47 55 55 55 55 00 00 97 5b a0 f7 ff 7f 00 00     0GUUUU...[......
0x00007fffffffdfd0     01 00 00 00 00 00 00 00 a8 e0 ff ff ff 7f 00 00     ................
```

# GEF: GDB Enhanced Features



- GDB extensions:
  - *keystone-assemble* — Assembly engine
  - *capstone-disassemble* — Disassembly engine
  - *unirorn-emulate* — Emulation engine
  - *Ropper* — ROP Gadget generator
  - *RetDec* — Decompiler

(see GEF docs for details)

# Outline



- Reading ELF data
- GDB: The GNU Project Debugger
  - GEF: GDB Enhanced Features
- Pwntools
- Other tools:
  - Radare2
  - Ghidra

# Pwntools…



- …is a *CTF framework* and *exploit development library* that is…
  - written in Python
  - designed for rapid prototyping
  - intended to make exploit writing as simple as possible

# Pwntools: Tubes



- *Tubes* are I/O wrappers supporting the main type of connections:
  - Local processes (pipe)
  - Remote TCP or UDP connections
  - Process running on a remote server over SSH
  - Serial port I/O

# Pwntools: Tubes



- Via a *tube* one can:
  - Send data
    - *send(data)* — Sends data
    - *sendline(line)* — Sends data plus a newline
  - Receive data
    - *recv(n)* — Receive the given number of bytes
    - *recvline()* — Receive data until a newline is found
    - *recvuntil(delim)* — Receive data until a delimiter is found
    - *recvregex(pattern)* — Receive data until a regex pattern is satisfied
    - *recvrepeat(timeout)* — Keep receiving data until a timeout occurs
    - *clean()* — Discard all vuffered data
  - Manipulating integers
    - *pack(int)* — Sends a word-size packed integer
    - *unpack()* — Receives and unpacks word-size integer

# Pwntools: Processes



- A *tube* can be used to interact with a *process* that can be created by passing:
  - the name of the binary to execute

```
io = process('sh')
```

  - the list of arguments and environment

```
io = process(['sh', '-c', 'echo $MYENV'], env={'MYENV': 'MYVAL'})
```

# Pwntools: Example



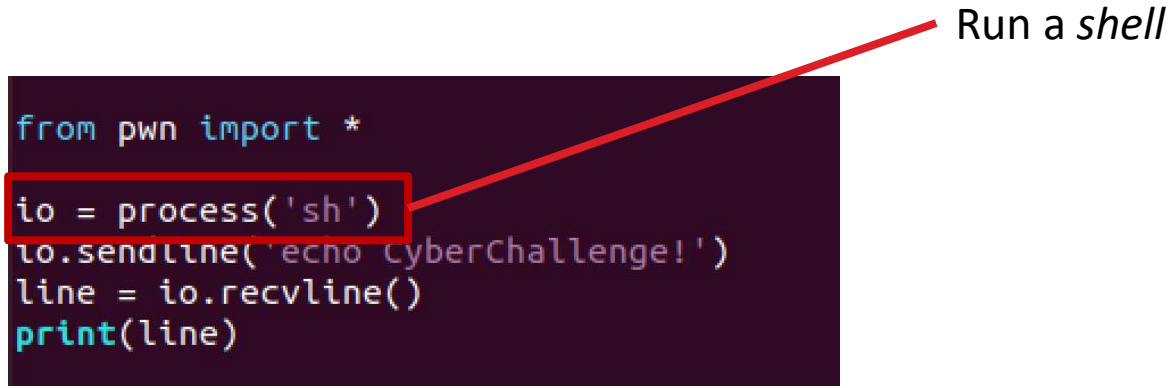
- Let us consider the following *python* script:

```
from pwn import *

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

# Pwntools: Example



- Let us consider the following *python* script:



# Pwntools: Example



- Let us consider the following *python* script:

```
from pwn import *

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

Send a command

# Pwntools: Example



- Let us consider the following *python* script:

```
from pwn import *

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

Receive the result

# Pwntools: Networking



- Via pwntools one can easily create network connections or waiting for incoming ones:
  - *remote* is used to open a connection with a remote server
  - *listen* is used to wait for an incoming connection

- Both *tcp* and *udp* protocols can be used

# Pwntools: Example



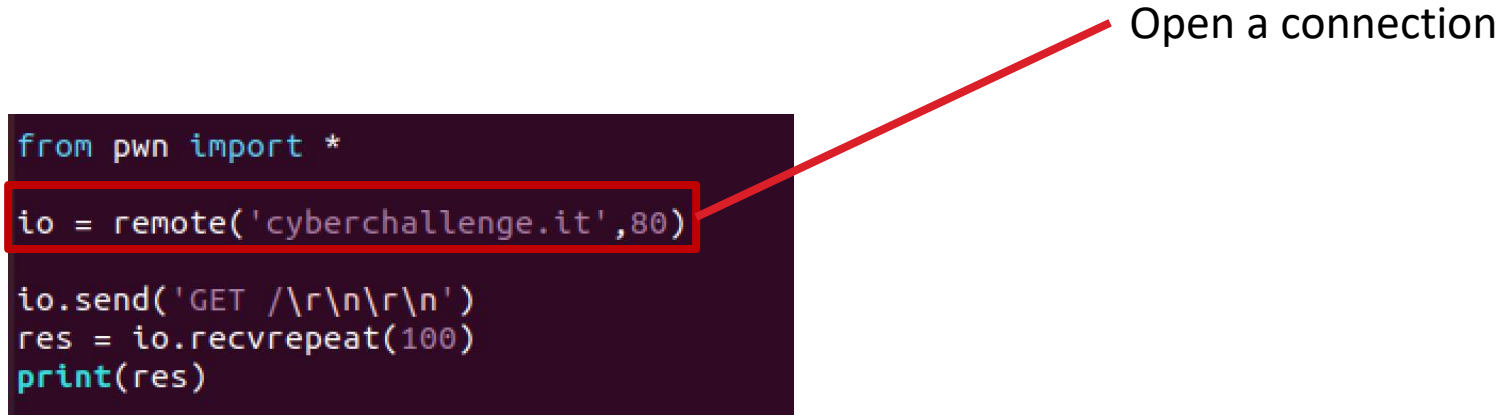
- Let us consider the following *python* script:

```
from pwn import *

io = remote('cyberchallenge.it',80)

io.send('GET /\r\n\r\n')
res = io.recvrepeat(100)
print(res)
```

# Pwntools: Example



- Let us consider the following *python* script:



# Pwntools: Example



- Let us consider the following *python* script:

```
from pwn import *

io = remote('cyberchallenge.it',80)

io.send('GET /\r\n\r\n')
res = io.recvrepeat(100)
print(res)
```

Send a request

# Pwntools: Example



- Let us consider the following *python* script:

```
from pwn import *

io = remote('cyberchallenge.it',80)

io.send('GET /\r\n\r\n')
res = io.recvrepeat(100)
print(res)
```

Receive a responce

# Pwntools: SSH



- *Pwntools* supports SSH sessions:

```
session = ssh('username', 'hostname', password='password')
```

- Given a session, one can execute *processes* as we have already seen in the previous slides:

```
io = session.process('sh')
```

# Pwntools: Other Features



- A *utility* module with functions for
  - *Packaging* and *unpackaging* integers
  - File I/O
  - Hashing and Encoding
  - Pattern Generation
- ELF files manipulation
- Assembling and disassembling

# Outline



- Reading ELF data
- GDB: The GNU Project Debugger
  - GEF: GDB Enhanced Features
- Pwntools
- Other tools:
  - Radare2
  - Ghidra

# Radare2



- Radare2 is a *toolchain* for easing task like:
  - Software Reverse Engineering
  - Exploiting
  - Debugging
- Radare2 is *open source* available at

  *https://github.com/radareorg/radare2*

# Radare2



- The framework consists of a number of *command-line* tools, among which we mention here:
  - *radare2:* a powerful *hexadecimal editor* and *debugger*
  - *rabin2:* a program that permits extracting information from executable binaries
  - *rasm2:* a command line *assembler* and *disassembler*
  - *rahash2:* an implementation of a *block-based* hash tool

# Outline



- Reading ELF data
- GDB: The GNU Project Debugger
  - GEF: GDB Enhanced Features
- Pwntools
- Other tools:
  - Radare2
  - Ghidra

# Ghidra



- Ghidra is a Software Reverse Engineering (SRE) tool developed by *National Security Agency* (NSA)
- Ghidra is a *platform independent* and includes:
  - an interactive *disassembler* and *decompiler*
  - a set of tools to support *code analysis*
- The tool is *open source* and available at:
  - https://ghidra-sre.org
- More details about Ghidra and its use will be given in the next module

# Software and Tools

**Michele LORETI**
Università di Camerino



*https://cybersecnatlab.it*