CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity National Lab

SPONSOR PLATINUM

accenturesecurity

aizoon AUSTRALIA EUROPE USA
TECHNOLOGY CONSULTING

B5

EY Building a better working world

eni

exprivia | ITALTEL

IBM

KPMG

LEONARDO

NTT DATA
Trusted Global Innovator

NUMERA
SISTEMI E INFORMATICA S.p.A.

Telsy

SPONSOR GOLD

bip.

cisco

MONTE DEI PASCHI DI SIENA
BANCA DAL 1472

negg

NOVANEXT
connecting the future

pwc

SPONSOR SILVER

DiGi ONE
the leading digital company

ICT CYBER CONSULTING

Programs and their Analysis

Michele LORETI
Univ. di Camerino
michele.loreti@unicam.it

CYBER CHALLENGE
CyberChallenge.IT

cini Cybersecurity National Lab

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.

Prerequisites

- Lecture:
  - *SS_1.1 – Secure programming*

Outline

- From code to Programs
- Executable and Linkable Format (ELF)
- From Programs to code
  - Decompilation
  - Disassembly
- Tools for static analysis
  - Extracting data from binaries
  - Code obfuscation
- Debugging and Tracing
  - Tools for dynamic analysis (an overview)
  - Memory leaks
  - Anti-debug protection techniques

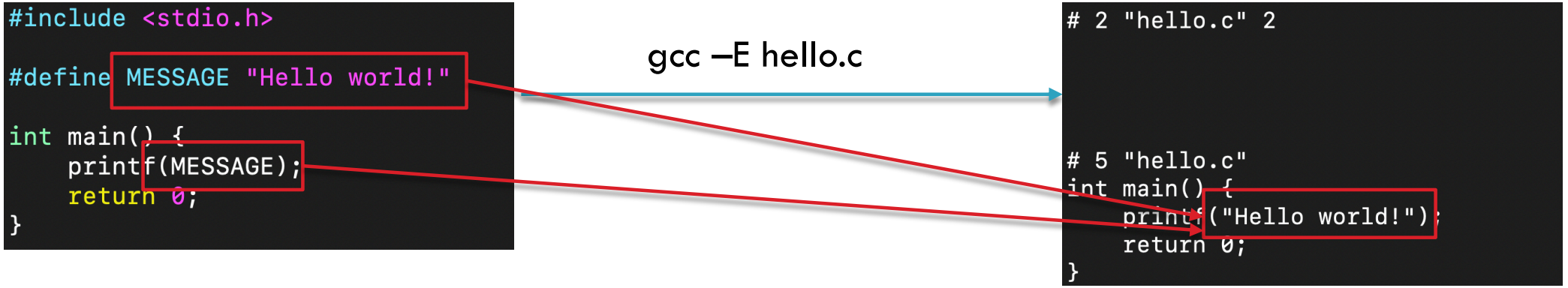
From code to programs

Compiling a C program is a multi-stage process composed by four steps:

- preprocessing
- compilation
- assembly
- linking

From code to programs: preprocessing

In the first phase *preprocessor commands (in C they start with '#') are interpreted:*



From code to programs: compilation

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

From code to programs: assembly

In the *assembly* phase assembly instructions are translated in *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.

Executable and Linkable Format

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

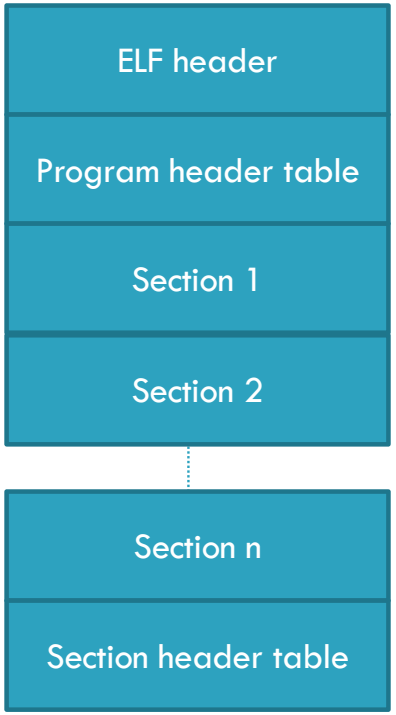
There are three types of object files:

- 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.

Gathering info from binary files

- Several tools are available to extract information from an ELF file, such as:
  - objdump
  - readelf
- These tools can be used to gather information from binaries that can be used to discover vulnerabilities.
- This is static analysis
  - A program is not analyzed while it is running, but by only inspecting the static structure of its binary file.

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.

Disassembly

- Given a binary file we can use a disassembler to extract from a binary file info about the executed code.
- This can be done by using objdump:

objdump –d <file>

Disassembly in action


```c
#include <stdio.h>

#define MESSAGE "Hello world!"

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

gcc –s hello.c

```
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
```

gcc –o hello hello.c

hello.o

objdump -d hello

```
0000000000400526 <main>:
  400526:       55                      push   %rbp
  400527:       48 89 e5                mov    %rsp,%rbp
  40052a:       bf c4 05 40 00          mov    $0x4005c4,%edi
  40052f:       b8 00 00 00 00          mov    $0x0,%eax
  400534:       e8 c7 fe ff ff          callq  400400 <printf@plt>
  400539:       b8 00 00 00 00          mov    $0x0,%eax
  40053e:       5d                      pop    %rbp
  40053f:       c3                      retq
```


Decompilation

- A decompiler is a tool that, given a binary file, tries to reconstruct a high-level source file;
- The output of a decompiler is typically concise and the program logic is clear;
- Decompilation is quite easy in languages like Java, while it is complex for general binary files;
- Obfuscator can be used to make decompilation harder.

Identifying vulnerabilities

Information collected from binaries can be used to discover program vulnerabilities:

- where a program takes an input and if this is validated;
- if memory is safely managed;
- if up-to-date libraries or third-party frameworks are used;
- how the application is compiled;
- if (static) strings are used to record sensitive data.

Example

Let us assume we have a program named guessmyname. When executed this program just ask the user to guess a name:

```
CC> ./guessmyname
Guess my name: Paulo
I am sorry, this is not my name!
Please, try again!
CC>
```

We can use readelf to check if the name to guess is *hardcoded* in the file.

Example

We can read the content of .rodata section:

```
CC> readelf -x .rodata guessmyname

Hex dump of section '.rodata':
  0x00400760 01000200 00000000 43616c65 66004775 ........Calef.Gu
  0x00400770 65737320 6d79206e 616d653a 20002531 ess my name: .%1
  0x00400780 30730000 00000000 57656c6c 20646f6e 0s......Well don
  0x00400790 65212059 6f752067 75657373 6564206d e! You guessed m
  0x004007a0 79206e61 6d652100 4920616d 20736f72 y name!.I am sor
  0x004007b0 72792c20 74686973 20697320 6e6f7420 ry, this is not
  0x004007c0 6d79206e 616d6521 0a506c65 6173652c my name!.Please,
  0x004007d0 20747279 20616761 696e2100           try again!.
```


Example

We can now use the string we have discovered in our program:

```
CC> ./guessmyname
Guess my name: Calef
Well done! You guessed my name!
CC>
```

WARNING: To avoid information leakage, never hardcode secrets in your program!

Example

We can also use the tool strings to extract all the constant strings used in the code:

```
CC> strings guessmyname
/lib64/ld-linux-x86-64.so.2
libc.so.6
__isoc99_scanf
puts
printf
malloc
strcmp
__libc_start_main
__gmon_start__
GLIBC_2.7
GLIBC_2.2.5
UH-X
AWAVA
AUATL
[]A\A]A^A_
Calef
Guess my name:
%10s
Well done! You guessed my name!
I am sorry, this is not my name!
Please, try again!
```


Example

Looking at the code, we can observe that the name is in fact *hardcoded*:

```
int main() {
    char* name = "Calef";
    char* input = malloc(10);

    printf("Guess my name: ");
    scanf("%10s",input);
    if (strcmp(name,input)==0) {
        printf("Well done! You guessed my name!\n");
    } else {
        printf("I am sorry, this is not my name!\nPlease, try again!\n");
    }
    return 0;

}
```


From static to dynamic analysis

- We have seen how information can be collected from a binary file.
- We learnt that, to avoid information leakage, private data should not be hardcoded.
- This is not enough!
- Program data can be inspected at runtime!

Dynamic analysis

- Dynamic analysis is a program analysis technique performed either on virtual or on real processor.
- This kind of analysis is performed using tools that are able to observe program execution:
  - *dynamic memory analysis*
  - *dynamic disassembly*

Debugging

- One approach to perform dynamic analysis is via a debugger
- 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
```


GDB: The GNU Project Debugger

- GDB, the GNU Project debugger, is a debugger that:
  - Starts your program, specifying anything that might affect its behavior (parameters, value of registries,…).
  - Makes your program stop on specified conditions.
  - Examines what has happened, when your program has stopped.
  - Changes things in your program, to enable you to experiment with correcting the effects of one bug and to go on to learn about others.

GDB: The GNU Project Debugger

- Main gdb commands
  - file <filename>: loads a file
  - break <function>: sets a breakpoint at a given function
  - layout asm: displays disassembly and command info
  - layout regs: displays registers info
  - run: starts debugged program
  - stepi: execute next instruction

GDB in action

A new version of our *game* has been released where developers say they have fixed the previously identified vulnerabilities:

```
CC> ./guessmyname2
Guess my name: Calef
I am sorry, this is not my name!
Please, try again!
```

They also changed the name…

Example

We read the content of .rodata section:

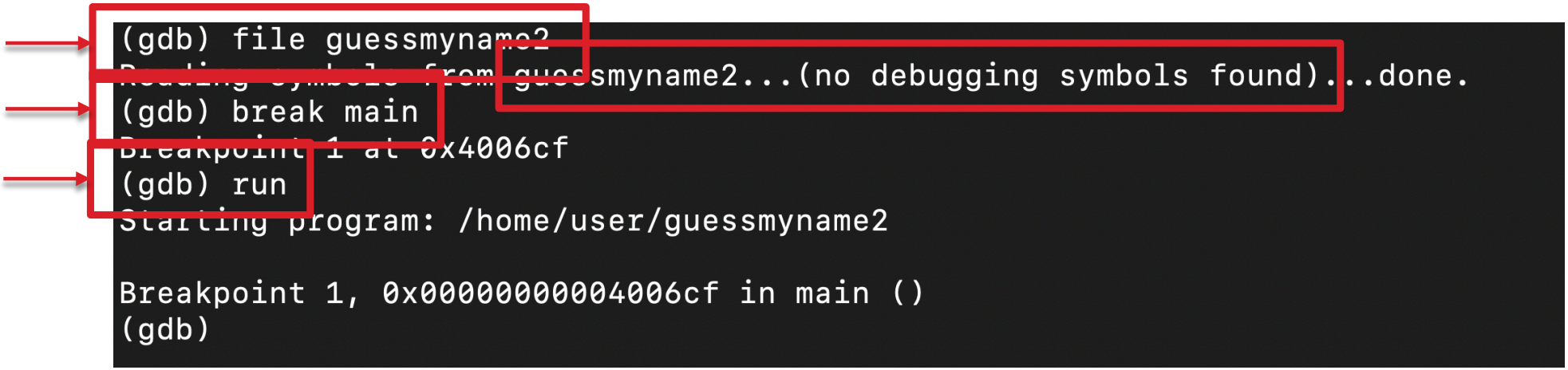
```
CC> readelf -x .rodata guessmyname2

Hex dump of section '.rodata':
  0x004007d0 01000200 00000000 47756573 73206d79 ........Guess my
  0x004007e0 206e616d 653a2000 25313073 00000000  name: .%10s....
  0x004007f0 57656c6c 20646f6e 65212059 6f752067 Well done! You g
  0x00400800 75657373 6564206d 79206e61 6d652100 uessed my name!.
  0x00400810 4920616d 20736f72 72792c20 74686973 I am sorry, this
  0x00400820 20697320 6e6f7420 6d79206e 616d6521  is not my name!
  0x00400830 0a506c65 6173652c 20747279 20616761 .Please, try aga
  0x00400840 696e2100                           in!.
```

- The lesson has been learnt… no name is evident in the program.
- We can use gdb to perform dynamic analylis of our code.

Example

We start gdb, load the binary, and set a breakpoint at function main:



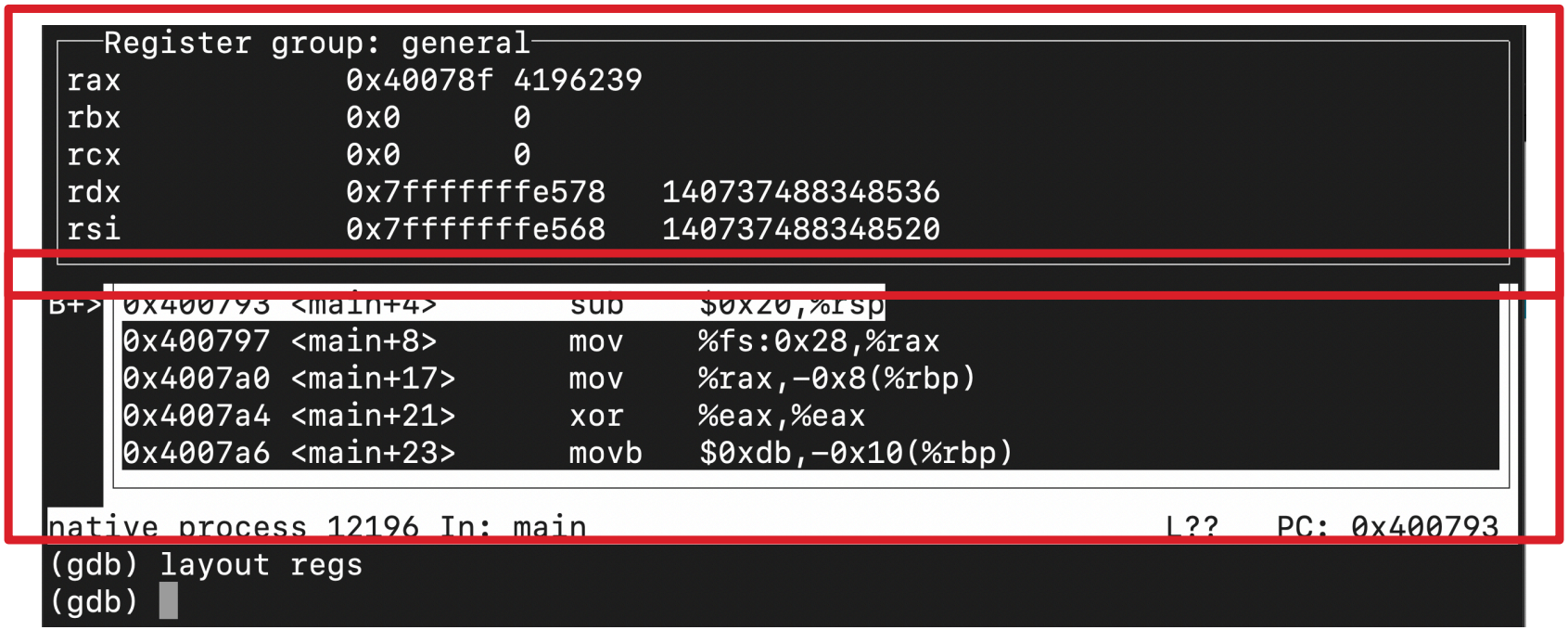
```
(gdb) file guessmyname2
Reading symbols from guessmyname2...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x4006cf
(gdb) run
Starting program: /home/user/guessmyname2

Breakpoint 1, 0x00000000004006cf in main ()
(gdb)
```

Note that we have not debug data in the object files.

Example

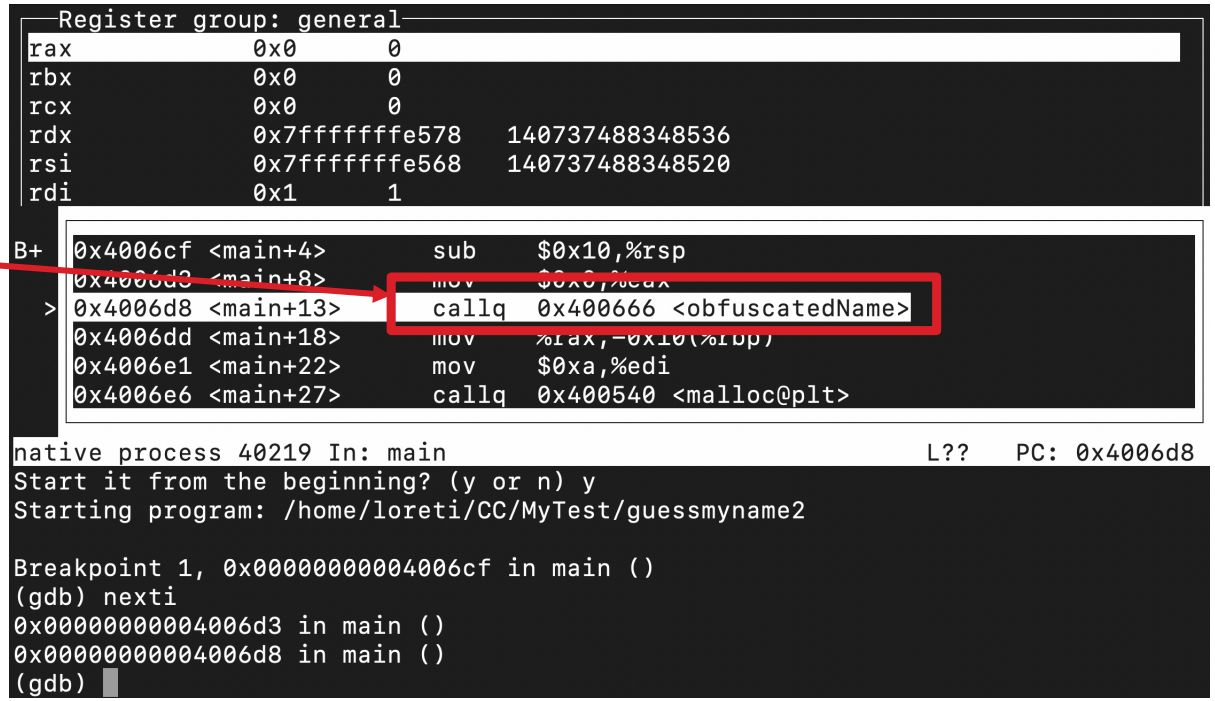
We setup the layout to see both registry and assembly:



Example

By using comman nexti we can interactively execute the program:

Function ‘obfuscated Name’ is called!



Example

In the next step, the result of invocation is copied from one registry ($rax) to a memory address:

```
B+  0x4006cf <main+4>       sub    $0x10,%rsp
    0x4006d3 <main+8>       mov    $0x0,%eax
>   0x4006d8 <main+13>      callq  0x400666 <obfuscatedName>
  > 0x4006dd <main+18>      mov    %rax,-0x10(%rbp)
    0x4006e1 <main+22>      mov    $0xa,%edi
    0x4006e6 <main+27>      callq  0x400540 <malloc@plt>
native process 40219 In: main                                  L??   PC: 0x4006dd
```


Example

We can inspect the content of registy $rax to read the value:

```
(gdb) x/s $rax
0x602010:       "Ulisse"
(gdb)
```

This is the name that we have to guess!

```
CC> ./guessmyname2
Guess my name: Ulisse
Well done! You guessed my name!
CC>
```


Example

If we examine the code we can observe that in this case the password is obfuscated, however it is revealed when saved in a variable:

```
int main() {
    char* name = obfuscatedName();
    char* input = malloc(10);

    printf("Guess my name: ");
    scanf("%10s",input);
    if (strcmp(name,input)==0) {
        printf("Well done! You guessed my name!\n");
    } else {
        printf("I am sorry, this is not my name!\nPlease, try again!\n");
    }
    return 0;

}
```


Example

The same information can be extraced by using the tool ltrace that allows to trace calls to libraries:

```
CC> ltrace ./guessmyname2
__libc_start_main(0x4006cb, 1, 0x7ffc04c1b0f8, 0x400750 <unfinished ...>
malloc(7)                                                = 0x25a9010
malloc(10)                                               = 0x25a9030
printf("Guess my name: ")                                = 15
__isoc99_scanf(0x4007e8, 0x25a9030, 0x7f3b635e9780, 15Guess my name: Harry
) = 1
strcmp("Ulisse", "Harry")                                = 13
puts("I am sorry, this is not my name!"...I am sorry, this is not my name!
Please, try again!
)      = 52
+++ exited (status 0) +++
CC>
```


How can we patch this program?


```
int main() {
    char* name = readEncryptedName();
    char* input = malloc(10);

    printf("Guess my name: ");
    scanf("%10s",input);
    if (strcmp(name,encrypt(input))==0) {
        printf("Well done! You guessed my name!\n");
    } else {
        printf("I am sorry, this is not my name!\nPlease, try again!\n");
    }
    return 0;

}
```


How can we patch this program?


```
int main() {
    char* name = readEncryptedName();
    char* input = malloc(10);

    printf("Guess my name: ");
    scanf("%10s",input);
    if (strcmp(name,encrypt(input))==0) {
        printf("Well done! You guessed my name!\n");
    } else {
        printf("I am sorry, this is not my name!\nPlease, try again!\n");
    }
    return 0;

}
```

Read encrypted name from a file.

How can we patch this program?


```
int main() {
    char* name = readEncryptedName();
    char* input = malloc(10);

    printf("Guess my name: ");
    scanf("%10s",input);
    if (strcmp(name,encrypt(input))==0) {
        printf("Well done! You guessed my name!\n");
    } else {
        printf("I am sorry, this is not my name!\nPlease, try again!\n");
    }
    return 0;

}
```

Encrypt the name read from input and compare it with the one we have!

How can we patch this program?


```
int main() {
    char* name = readEncryptedName();
    char* input = malloc(10);

    printf("Guess my name: ");
    scanf("%10s",input);
    if (strcmp(name,encrypt(input))==0) {
        printf("Well done! You guessed my name!\n");
    } else {
        printf("I am sorry, this is not my name!\nPlease, try again!\n");
    }
    return 0;

}
```

The implementation of ‘readEncryptedName’ and ‘encrypt’ is left as an exercise!

Programs and their Analysis

Michele LORETI
Univ. di Camerino
michele.loreti@unicam.it

CYBER CHALLENGE
CyberChallenge.IT

cini Cybersecurity National Lab

https://cybersecnatlab.it

CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity National Lab

SPONSOR PLATINUM

accenturesecurity

aizoon AUSTRALIA EUROPE USA
TECHNOLOGY CONSULTING

B5

EY Building a better working world

eni

exprivia | ITALTEL

IBM

KPMG

LEONARDO

NTT DATA
Trusted Global Innovator

NUMERA
SISTEMI E INFORMATICA S.p.A.

Telsy

SPONSOR GOLD

bip.

cisco

MONTE DEI PASCHI DI SIENA
BANCA DAL 1472

negg

NOVANEXT
connecting the future

pwc

SPONSOR SILVER

DiGi ONE
the leading digital company

ICT CYBER CONSULTING