Software Security 02

Dal sorgente al codice eseguibile (e ritorno)

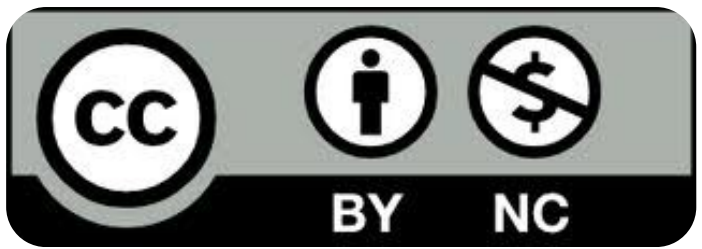
Gianluca Amato
Università di Chieti-Pescara

https://cybersecnatlab.it

License & Disclaimer

License Information

This presentation is licensed under the Creative Commons BY-NC License



To view a copy of the license, visit:

http://creativecommons.org/licenses/by-nc/3.0/legalcode

Disclaimer

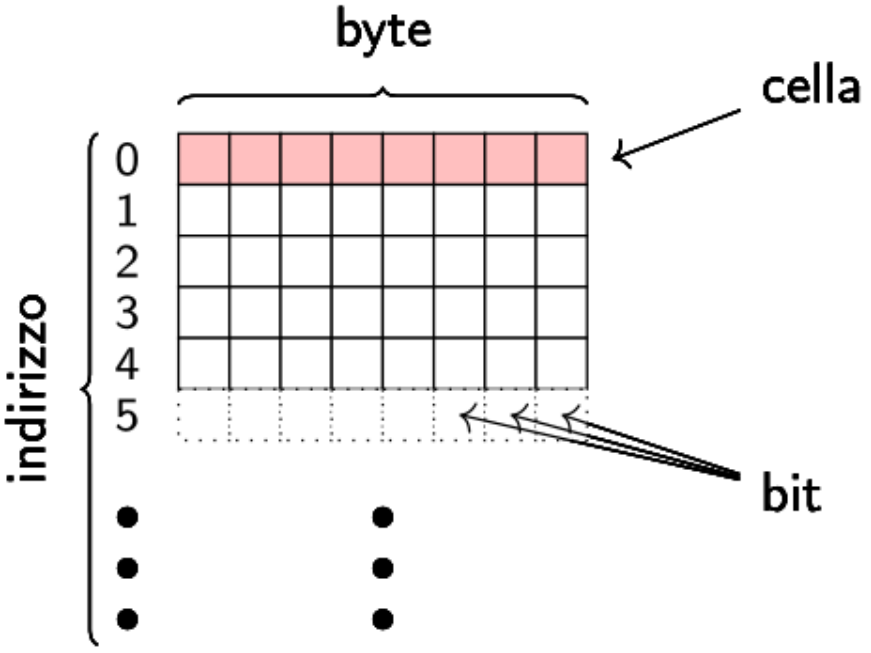
- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- 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.

CPU e memoria

- Per queste lezioni di software security, le parti di un computer su cui focalizzeremo l'attenzione sono:
  - la CPU (Central Processing Unity);
  - la memoria centrale (talvolta chiamata RAM).
- Sulla CPU diremo di più nelle prossime lezioni.
- Prima di iniziare la vera lezione, diciamo due parole sulla memoria centrale.
  - Ci interessa come un programma vede la memoria centrale.
  - In realtà le cose sono più complesse, ma questa complessità è gestita dal sistema operativo.

La memoria centrale

- La memoria centrale è una sequenza di celle:
  - Ogni cella ha un indirizzo
  - Ogni cella contiene un byte
    - numero binario di 8 bit
    - ovvero numero da 0 a 255
- Le istruzioni della CPU consentono di accedere a qualunque cella di memoria, partendo dall'indirizzo.



Numerazione in base 16

- Spesso (sempre) scriveremo i numeri in base 16 (esadecimale).
- Vi ricordo che i numeri in base 16 si scrivono con le cifre da 0 a 9 e le lettere da A (10) ad F (15).
  - Un byte in base 16 contiene un numero da 0 ad FF (255)
  - Su due byte possiamo mettere un numero da 0 ad FFFF (65535)
  - …e così via
- Se incontriamo un numero, come facciamo a capire se è in base 10 o in base 16?
  - Se contiene lettere, è in base 16 (1D5)
  - Talvolta useremo il prefisso 0x o il suffisso h per i numeri in base 16 (0x154)
  - Se il numero ha degli 0 iniziali, è in base 16 (0154)
    - A meno che non usi solo le cifre 0 ed 1, al ché potrebbe essere in base 2

Dal sorgente all'eseguibile

Il linguaggio macchina

- Il computer capisce un solo linguaggio: il linguaggio macchina (d'ora in poi LM)
- **Vantaggi**
  - è possibile sfruttare tutte le potenzialità dell'hardware.
- **Svantaggi**
  - estremamente complesso;
  - ogni istruzione è una sequenza di byte (difficile da ricordare);
  - cambia completamente da un famiglia di CPU all'altra
    - Intel/AMD a 64 bit
    - ARM (CPU usata per gli smartphone o i nuovi Mac)
    - ....
  - il programma è strettamente legato al sistema operativo per cui è scritto:
    - un programma in LM per Linux non funziona su Windows o Mac, neanche se la CPU è la stessa.

Un programma in LM

- Questo programma visualizza la scritta “Hello, world!” sullo schermo e poi termina
- Funziona su PC con Linux e CPU Intel/AMD a 64 bit

```
48 c7 c0 01 00 00 00 48 c7 c7 01 00 00 00 48 c7
c6 00 00 00 00 48 c7 c2 0f 00 00 00 0f 05 48 c7
c0 3c 00 00 00 48 c7 c7 00 00 00 00 0f 05 48 65
6c 6c 6f 2c 20 77 6f 72 6c 64 21 0a
```

Il linguaggio assembly


```
.text
  .global _start
_start:
  mov $1, %rax
  mov $1, %rdi
  mov $msg, %rsi
  mov $len, %rdx
  syscall
  mov $60, %rax
  mov $0, %rdi
  syscall
.data
  msg:
    .string "Hello, world!\n"
msgend:
  .equ len, msgend - msg
```

- Il LM non è fatto per gli umani e nessuno (oggi) lo usa direttamente.
- In qualche occasione si usa il linguaggio assembly.
  - Ogni istruzione corrisponde a una istruzione in LM.
  - Più facile da ricordare rispetto al LM.

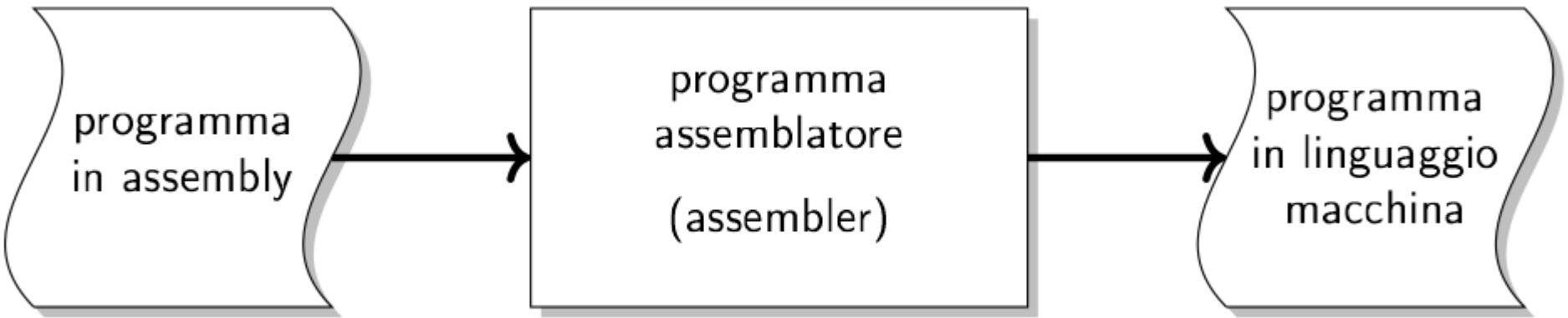
Assembly e linguaggio macchina


```
1                   .text
2                     .global _start
3                   _start:
4  0000 48C7C001      mov $1, %rax
4       000000
5  0007 48C7C701      mov $1, %rdi
5       000000
6  000e 48C7C600      mov $msg, %rsi
6       000000
7  0015 48C7C20F      mov $len, %rdx
7       000000
8  001c 0F05          syscall
8
9  001e 48C7C03C      mov $60, %rax
9       000000
10 0025 48C7C700      mov $0, %rdi
10      000000
11 002c 0F05          syscall
12                  .data
13                    msg:
14 0000 48656C6C        .string "Hello, world!\n"
14      6F2C2077
14      6F726C64
14      210A00
```

istruzione in assembly

istruzione in lingiaggio macchina

Dal linguaggio assembly al linguaggio macchina

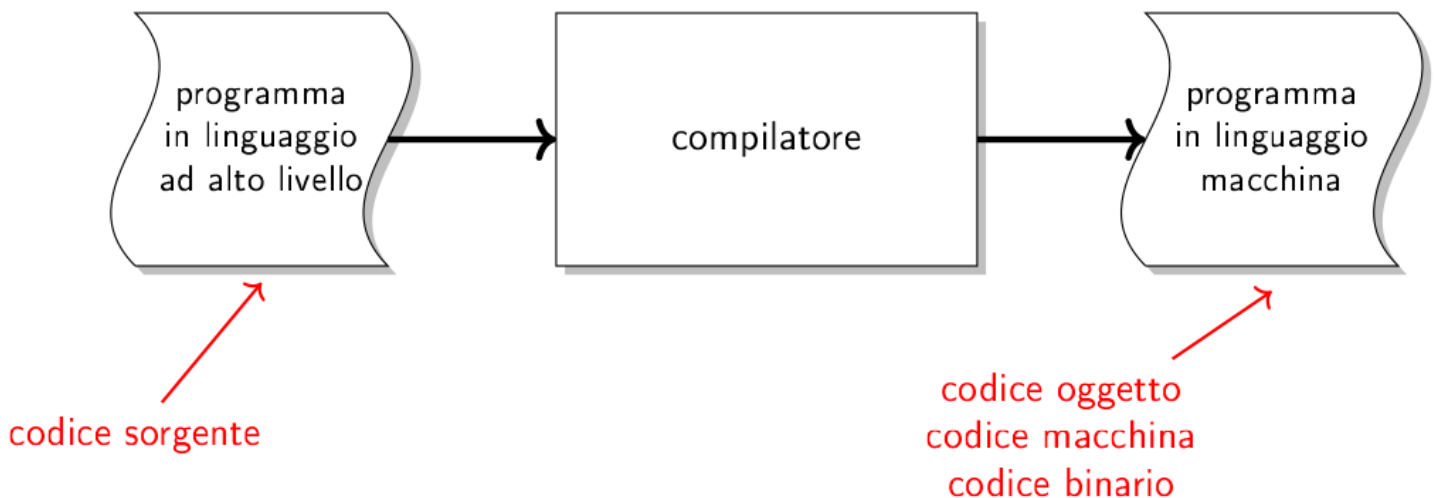


Linguaggi ad alto livello

- Sia il LM che l'assembly sono linguaggi a basso livello.
- Normalmente si programma con linguaggi ad alto livello (C, Python, Java, ...):
  - più semplici da comprendere per un essere umano;
  - non dipendono dalla CPU utilizzata;
  - non dipendono (almeno per le cose semplici) dal sistema operativo;
  - non sono eseguibili direttamente dalla CPU.
- Come si fa a far eseguire un programma scritto in linguaggio ad alto livello?
  - compilatore
  - interprete
  - soluzioni ibride

Compilatore

- Legge il programma in linguaggio ad alto livello e lo traduce in linguaggio macchina tutto in una volta. Una volta tradotto il compilatore non serve più
  - Esempi di linguaggi tipicamente compilati: C, C++, Rust

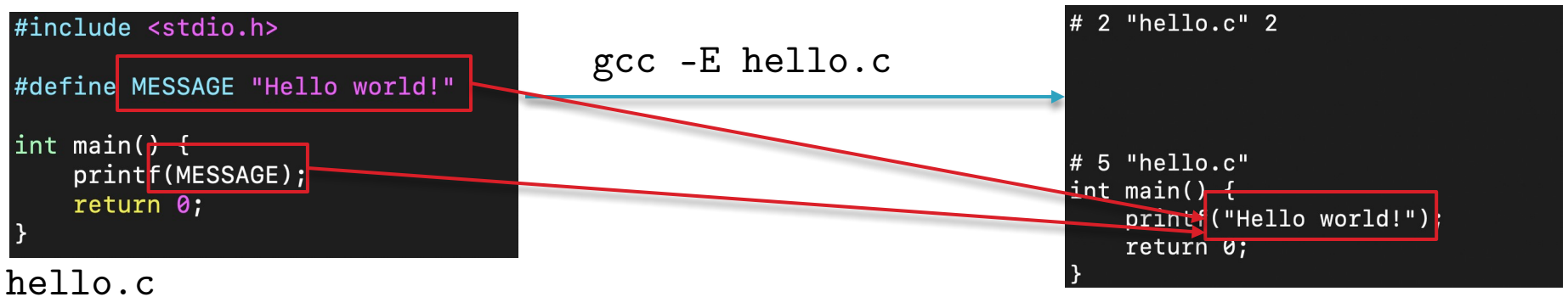


Compilazione del linguaggio C

- Ci occuperemo nelle prossime lezioni del linguaggio C
- Fasi della compilazione di un programma in:
  - preprocessing
  - compilazione vera e propria
  - assemblaggio (assembly)
  - collegamento (linking)
- Nelle lezioni useremo il compilatore GCC (GNU Compiler Collection)
  - disponibile per molti sistemi operativi;
  - lo standard per i sistemi Linux.

Fase di preprocessing

- Si occupa di eseguire le direttive presenti nel codice sorgente, ovvero le righe che iniziano con # come `#include`, `#define`, etc…
- Si può dire al gcc di fermarsi alla fase di preprocessing con l'opzione `-E`



Fase di compilazione vera e propria

- Il codice dalla fase precedente è trasformato in istruzioni assembly.
- Si può dire al gcc di fermarsi alla fase di compilazione vera e propria con l'opzione `-S`

Fase di assemblaggio

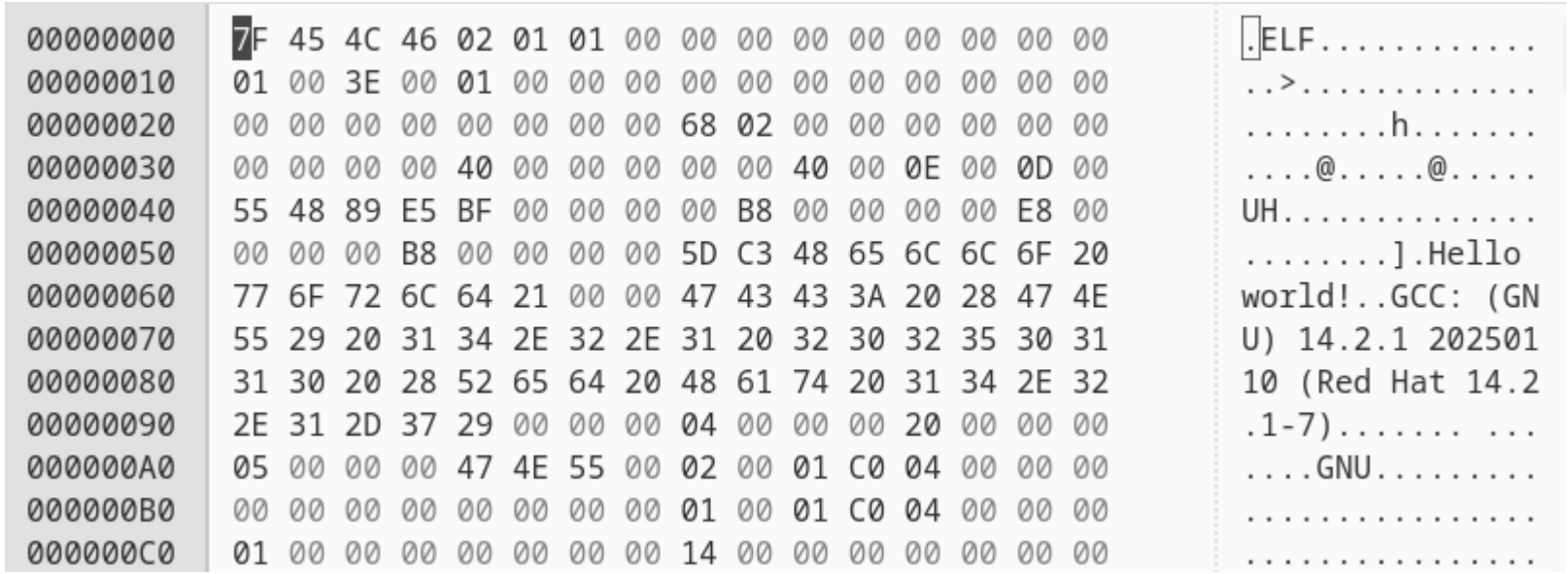
- Il codice assembly viene tradotto in LM
  - ma in un formato che non è ancora pronto per essere eseguito.
- Si può dire al gcc di fermarsi alla fase di assemblaggio con l'opzione `-c`.

Il codice oggetto

- Il file `.o` che si ottiene con `gcc -c` si chiama file oggetto.
  - Per essere più precisi, file oggetto *non eseguibile*.
- Contiene il programma in linguaggio macchina e informazioni di supporto.
- È un file binario, ovvero contiene sequenze di byte che non sono interpretabili come testo.
  - Se si prova ad aprilo con un editor di testo come Visual Studio Code, si ottiene spazzatura.
- Si può vederne il contenuto con un editor esadecimale.

Il codice oggetto visto da GHex

- File `hello.o` visto col programma *GHex* di Linux
  - Vedremo in futuro programmi specifici per questo tipo di file



```
00000000  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00  .ELF............
00000010  01 00 3E 00 01 00 00 00 00 00 00 00 00 00 00 00  ..>.............
00000020  00 00 00 00 00 00 00 00 68 02 00 00 00 00 00 00  ........h.......
00000030  00 00 00 00 40 00 00 00 00 00 40 00 0E 00 0D 00  ....@.....@.....
00000040  55 48 89 E5 BF 00 00 00 00 B8 00 00 00 00 E8 00  UH..............
00000050  00 00 00 B8 00 00 00 00 5D C3 48 65 6C 6C 6F 20  ........].Hello 
00000060  77 6F 72 6C 64 21 00 00 47 43 43 3A 20 28 47 4E  world!..GCC: (GN
00000070  55 29 20 31 34 2E 32 2E 31 20 32 30 32 35 30 31  U) 14.2.1 202501
00000080  31 30 20 28 52 65 64 20 48 61 74 20 31 34 2E 32  10 (Red Hat 14.2
00000090  2E 31 2D 37 29 00 00 00 04 00 00 00 20 00 00 00  .1-7)....... ...
000000A0  05 00 00 00 47 4E 55 00 02 00 01 C0 04 00 00 00  ....GNU.........
000000B0  00 00 00 00 00 00 00 00 01 00 01 C0 04 00 00 00  ................
000000C0  01 00 00 00 00 00 00 00 14 00 00 00 00 00 00 00  ................
```

Fase di collegamento

- Codici oggetti multipli vengono combinati tra di loro e con le librerie di sistema (`.dll` in Windows, `.so` in Linux) in un unico file eseguibile.
  - Ad esempio, la funzione `printf` usata nel programma `hello.c` si trova nella libreria C di Linux

(Nome convenzionale del file eseguibile generato da gcc)

Collegamento statico e dinamico

- Due approcci sono possibili per la fase di collegamento
  - Collegamento dinamico: il file eseguibile dipende da librerie dinamiche (file oggetto condivisi) e funziona solo in loro presenza.
    - Il codice di `printf` non viene copiato nel file eseguibile.

    È la soluzione di default
  - Collegamento statico: il file eseguibile generato è auto-contenuto e non dipende da nessuna libreria esterna.
    - Il codice di `printf` viene copiato nel file eseguibile

    Si può attivare con l'opzione `-static` di gcc

Collegamento dinamico


```
$ gcc hello.c -o hello
$ ls -l hello
-rwxr-xr-x. 1 amato amato 16616 13 apr 10.46 hello
$ ldd hello
        linux-vdso.so.1 (0x00007f9e14284000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f9e1406e000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f9e14286000)
$
```

- Il file è relativamente piccolo, solo 16.616 byte
- Il comando `ldd` mostra le librerie dinamiche utilizzate:
  - `/lib64/ld-linux-x86-64.so`: è il loader (vedi dopo)
  - `linux-vdso.so.1`: accelera alcune chiamate al sistema operativo
  - `/lib64/libc.so.6`: è la libreria C, e contiene il codice di `printf`
  - I nomi di queste librerie possono cambiare da una versione di Linux ad un altra. Quelli che trovate qui sono per Fedora Linux.

Collegamento statico


```
$ gcc hello.c -static -o hello-static
$ ls -l hello-static
-rwxr-xr-x. 1 amato amato 799096 13 apr 11.20 hello-static
$ ldd hello-static
        not a dynamic executable
$
```

- Il file è molto più grande !
  - contiene una copia di varie funzioni della libreria C.
- Per funzionare non ha bisogno che nel sistema sia presente alcuna libreria:
  - Il comando `ldd` risponde che non si tratta di un eseguibile con collegamento dinamico.

Il caricatore (loader)

- La parte del sistema operativo che si occupa di caricare un programma in memoria ed eseguirla.
- Per gli eseguibili statici:
  - Molto semplice, carichi il file in memoria e via !
- Per gli eseguibili dinamici:
  - Bisogna modificare il file eseguibile durante il caricamento:
    - gli indirizzi delle funzioni provenienti dalla librerie dinamiche (come printf) sono fittizi;
    - vanno riempiti con gli indirizzi reali.
  - In Linux su Intel a 64bit il lavoro è svolto da un programma specifico
    - `/lib64/ld-linux-x86-64.so.2`

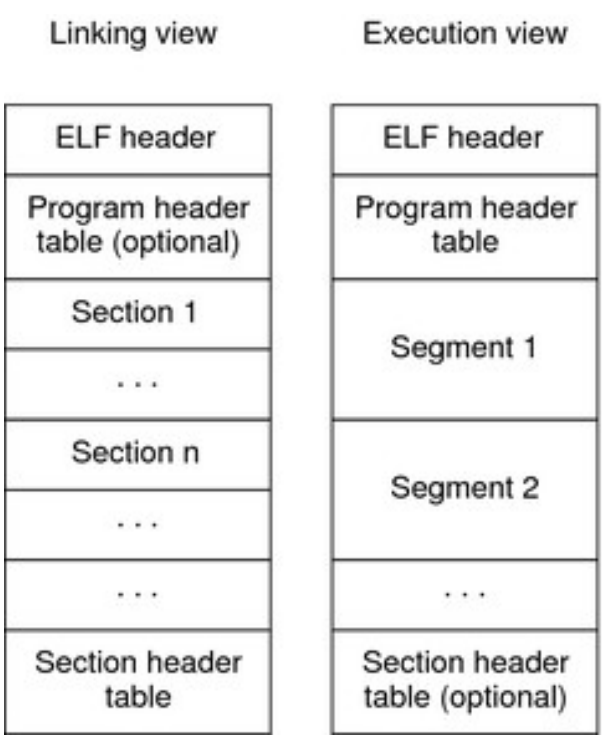
Il codice oggetto

Il formato ELF

- In Linux il formato standard per i file oggetto è il formato ELF
  - ELF: Executable and Linkable Format (https://wiki.osdev.org/ELF)
- Ci sono tre tipi di file oggetto:
  - rilocabili (relocatable): contengono codice e dati che possono essere collegati con altri file rilocabili per creare nuovi file ELF (esempio, il file `hello.o`)
  - eseguibili (executable): contengono programmi pronti ad essere eseguiti (esempio, i file `hello` ed `hello-static`)
    - possono dipendere da librerie dinamiche oppure no
  - condivisi (shared): come i rilocabili, ma possono svolgere il ruolo di librerie dinamiche (esempio, il file /lib64/libc.so.6)

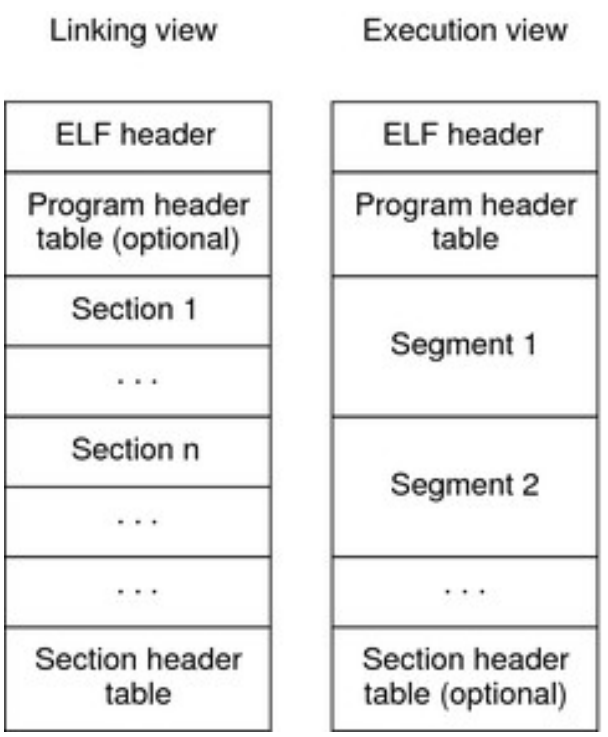
Struttura di un file ELF

- Un file ELF può essere usato in due contesti:
  - esecuzione di un programma;
  - collegamento con altro codice oggetto.
- Per svolgere questo doppio compito, il file ELF può essere visto contemporaneamente in due modi distinti:
  - come un insieme di sezioni;
  - come un insieme di segmenti.



Struttura di un file ELF

- Una intestazione iniziale (ELF header)
  - contiene informazioni sulla struttura del file ELF.
- Il file ELF è diviso in sezioni
  - Manipolate dalla fase di collegamento.
  - Elenco presente nella "*Section header table*".
- Le sezioni sono raggruppate in segmenti
  - Manipolati dal caricatore (loader).
  - Elenco presente nella "*Program header table*".
  - Solo per i file eseguibili.



Sezioni rilevanti di un file ELF

- .text: istruzioni del programma
- .data: dati inizializzati
- .bss: spazio usato dal programma per dati non inizializzati
- .rodata: simile a .data, ma per dati in sola lettura
- .symtab, .strtab: tabella dei simboli definiti nel programma
- .dynsym, .dynstr: tabella della funzioni da librerie dinamiche
- .dynamic, .rela.dyn, .rela.plt, .got, .got.plt: informazioni per il loader

Analizzare il contenuto del file ELF

- Strumenti su riga di comando per analizzare un file ELF:
  - `strings`
  - `objdump`
  - `readelf`
  - `nm`
- Strumenti grafici per analizzare un file ELF:
  - Cutter (https://github.com/rizinorg/cutter)
  - Elfparser-ng (https://github.com/mentebinaria/elfparser-ng)

strings

- Semplice strumento per visualizzare tutte le stringhe presenti in un file
- `strings` visualizza
  - sequenze di caratteri stampabili
  - lunghe almeno 4 caratteri
  - seguite da caratteri non stampabili
- Le stringhe potrebbero contenere valori segreti.

```
$ strings hello
PTE1
/lib64/ld-linux-x86-64.so.2
__libc_start_main
printf
libc.so.6
GLIBC_2.2.5
GLIBC_2.34
__gmon_start__
Hello world!
;*3$"
GCC: (GNU) 14.2.1 20250110 (Red Hat 14.2.1-7)
AV:4p1269
```

strings

Alcune opzioni di `strings`

- `-d` (o `--data`): visualizza solo le stringhe presenti nella sezioni dati di un file ELF
- `-n <num>` (o `--bytes=<num>`): visualizza sequenze di caratteri stampabili lunghe almeno <num> (il default è 4)
- `-e <encoding>` o (`--encoding=<encoding>`): imposta la codifica utilizzata per le stringhe.
  - per default, solo i caratteri ASCII standard (codici 0–127) vengono considerati stampabili;
  - per stringhe che contengono anche caratteri accentati, usare `-eS`.

Ambiente per l'esecuzione di codice a 32bit

- Alcune challenge sono per CPU Intel a 32bit
  - La vostra CPU è sicuramente a 64bit
  - Può eseguire anche codice a 32bit, ma è necessario installare dei pacchetti addizionali
- Su distribuzioni Debian e derivate (Ubuntu, Kali, etc…), installare il pacchetto `libc6-i386` con il comando
  - `apt install libc6-i386`

Esercizi

Svolgere la challenge

SS_01 – The safe

readelf

Visualizzare l'intestazione (`-h` / `--file-header`)

- `Type`
  - `EXEC` (eseguibile)
  - `REL` (rilocabile)
  - `DYN` (condiviso)
- `Machine`
  - Tipo CPU
  - 32 / 64 bit

```
$ readelf -h hello
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:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401040
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14568 (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:         32
  Section header string table index: 31
```

readelf

Visualizzare le sezioni (`-S` / `--sections`)

`-W` per formato largo più leggibile

```
$ readelf -W --sections hello
There are 32 section headers, starting at offset 0x38e8:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.property NOTE            0000000000400318 000318 000040 00   A  0   0  8
  [ 2] .note.gnu.build-id NOTE            0000000000400358 000358 000024 00   A  0   0  4
  [ 3] .note.ABI-tag     NOTE             000000000040037c 00037c 000020 00   A  0   0  4
  [ 4] .init             PROGBITS         0000000000401000 001000 00001b 00  AX  0   0  4
  [ 5] .plt              PROGBITS         0000000000401020 001020 000020 10  AX  0   0 16
  [ 6] .text             PROGBITS         0000000000401040 001040 000100 00  AX  0   0 16
  [ 7] .fini             PROGBITS         0000000000401140 001140 00000d 00  AX  0   0  4
  [ 8] .interp           PROGBITS         0000000000402000 002000 00001c 00   A  0   0  1
```

- `Name`: nome della sezione
- `Type`: tipo della sezione (`NULL`: sezione vuota, `PROGBITS`: programmi e dati, etc...)
- `Address`: indirizzo in memoria dove viene caricata la sezione (per i file eseguibili)
- `Off`: indirizzo della sezione a partire dall'inizio del file (offset)
- `Size`: dimensione della sezione
- Flags: informazioni sulla sezione (`A`: occupa spazio in memoria, `W`: è scrivibile, `X`: contiene istruzioni, etc...)

readelf

Visualizzare la tabella dei simboli (`-s` / `--symbols`)


```
$ readelf -W --syms hello

Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.34 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (3)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

Symbol table '.symtab' contains 34 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crt1.o
```

- Si tratta del contenuto delle sezioni
  - `.dynsym`: simboli usati per il caricamento dinamico
    - Le funzioni `printf` e `__libc_start_main` devono essere fornite dall'esterno
  - `.symtab`: lista completa dei simboli

readelf

Visualizzare la tabella dei simboli (`-s` / `--symbols`)


```
    26: 0000000000402170     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    27: 0000000000404010     0 NOTYPE  GLOBAL DEFAULT   25 _end
    28: 0000000000401070     5 FUNC    GLOBAL HIDDEN     6 _dl_relocate_static_pie
    29: 0000000000401040    38 FUNC    GLOBAL DEFAULT    6 _start
    30: 000000000040400c     0 NOTYPE  GLOBAL DEFAULT   25 __bss_start
    31: 0000000000401126    26 FUNC    GLOBAL DEFAULT    6 main
    32: 0000000000404010     0 OBJECT  GLOBAL HIDDEN    24 __TMC_END__
    33: 0000000000401000     0 FUNC    GLOBAL HIDDEN     4 _init
```

- Nella sezione `.symtab`
  - Compare la funzione `main` all'indirizzo `x401126`
  - Il programma non parte in realtà dalla funzione `main` ma da `_start`
    - Confrontate l'indirizzo di `_start` con l'entry-point del programma nell'intestazione del file ELF

readelf

Visualizzare il contenuto di una sezione

- Opzione `-x <nome_sezione>` / `--hex-dump=<nome_sezione>`

```
$ readelf -x .rodata hello

Hex dump of section '.rodata':
  0x00402170 01000200 00000000 00000000 00000000 ................
  0x00402180 48656c6c 6f20776f 726c6421 00       Hello world!.

$ readelf -x .text hello

Hex dump of section '.text':
  0x00401040 f30f1efa 31ed4989 d15e4889 e24883e4 ....1.I..^H..H..
  0x00401050 f0505445 31c031c9 48c7c726 114000ff .PTE1.1.H..&.@..
  0x00401060 15732f00 00f4662e 0f1f8400 00000000 .s/...f.........
  0x00401070 f30f1efa c3662e0f 1f840000 00000090 .....f..........
  0x00401080 b8104040 00483d10 40400074 13b80000 ..@@.H=.@@.t....
  0x00401090 00004885 c07409bf 10404000 ffe06690 ..H..t...@@...f.
  0x004010a0 c366662e 0f1f8400 00000000 0f1f4000 .ff...........@.
```

Esercizi

Svolgere la challenge

Challenge SS_03 – dissection

Dall’eseguibile al sorgente

Dal file eseguibile all'assembly

- Spesso abbiamo a disposizione solo il file oggetto.
  - Il codice del programma si trova di solito nella sezione `.text`
  - Con `readelf` possiamo leggere il contenuto della sezione in esadecimale, ma non è per nulla comprensibile
- Ci viene in aiuto `objdump`
  - Simile a `readelf`, ma è in grado di visualizzare il contenuto della sezione `.text` in assembly invece che in LM
  - Un programma che trasforma un codice in LM in codice assembly si chiama disassemblatore.

objdump

Disassemblare il programma (`-d` / `--disassemble`)


```
$ objdump --disassembler-color=on -d hello

hello:     file format elf64-x86-64


Disassembly of section .init:

0000000000401000 <_init>:
  401000:       f3 0f 1e fa             endbr64
  401004:       48 83 ec 08             sub    $0x8,%rsp
  401008:       48 8b 05 d1 2f 00 00    mov    0x2fd1(%rip),%rax        # 403fe0 <__gmon_start__@Base>
  40100f:       48 85 c0                test   %rax,%rax
  401012:       74 02                   je     401016 <_init+0x16>
  401014:       ff d0                   call   *%rax
  401016:       48 83 c4 08             add    $0x8,%rsp
  40101a:       c3                      ret

Disassembly of section .plt:

0000000000401020 <printf@plt-0x10>:
  401020:       ff 35 ca 2f 00 00       push   0x2fca(%rip)        # 403ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
  401026:       ff 25 cc 2f 00 00       jmp    *0x2fcc(%rip)        # 403ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
  40102c:       0f 1f 40 00             nopl   0x0(%rax)
```

`--disassebler-color=on`
visualizza l'output a colori

objdump

Disassemblare una singola funzione (`--disassemble=<func>`)


```
$ objdump --disassembler-color=on --disassemble=main hello

hello:     file format elf64-x86-64


Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .text:

0000000000401126 <main>:
  401126:       55                      push   %rbp
  401127:       48 89 e5                mov    %rsp,%rbp
  40112a:       bf 80 21 40 00          mov    $0x402180,%edi
  40112f:       b8 00 00 00 00          mov    $0x0,%eax
  401134:       e8 f7 fe ff ff          call   401030 <printf@plt>
  401139:       b8 00 00 00 00          mov    $0x0,%eax
  40113e:       5d                      pop    %rbp
  40113f:       c3                      ret

Disassembly of section .fini:
```

Esercizi

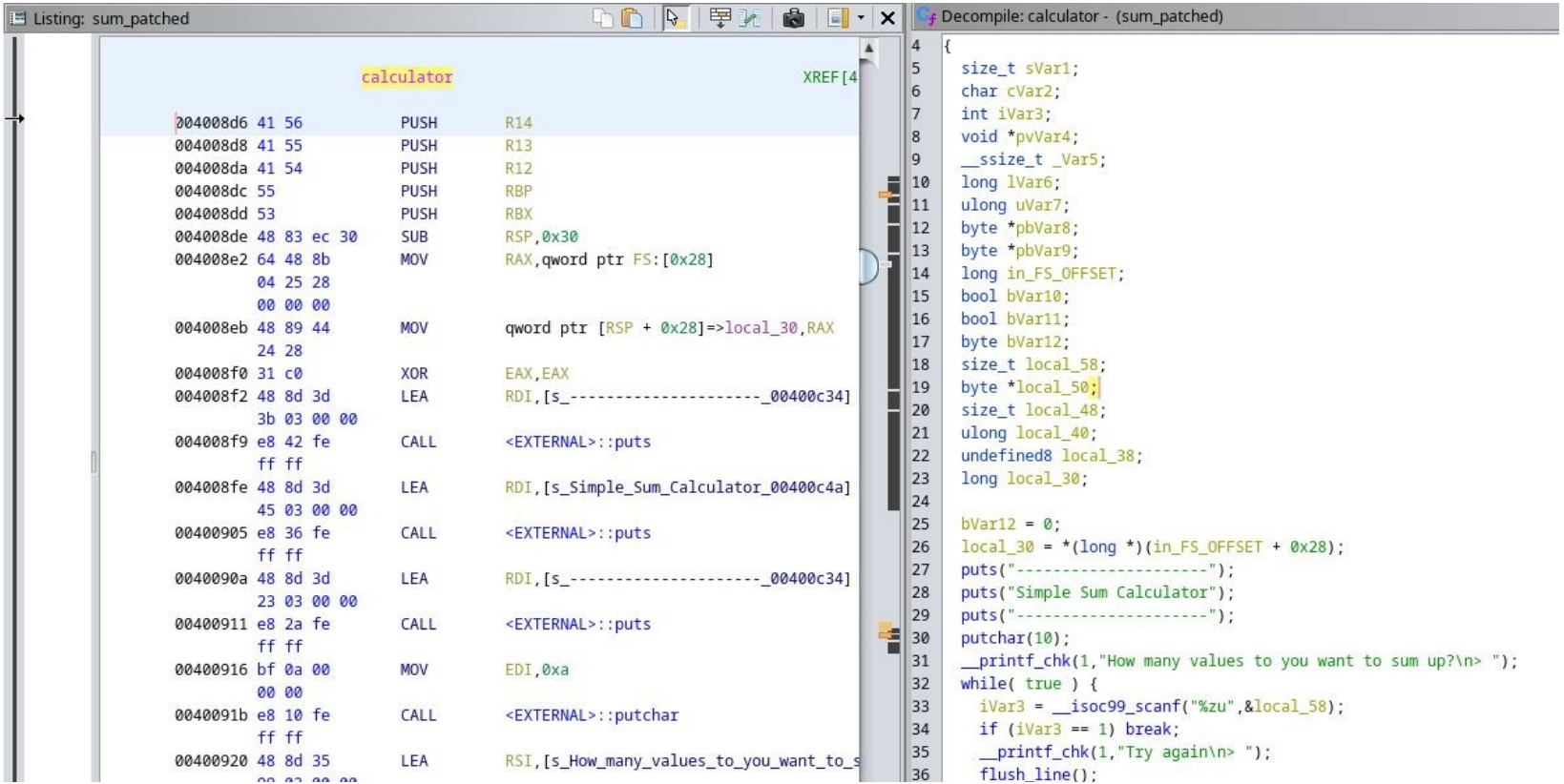
Svolgere la challenge

Challenge SS_02 – acrostic

Dal file eseguibile al C

- Il linguaggio assembly è comunque di difficile lettura.
- Vorremmo riuscire a tornare indietro dal codice oggetto ad un programma ad alto livello scritto in C.
  - Ci serve un decompilatore.
- L'operazione non è comunque del tutto automatizzabile e serve un po' di intervento manuale per ottenere un codice C leggibile.

Decompilatore in azione



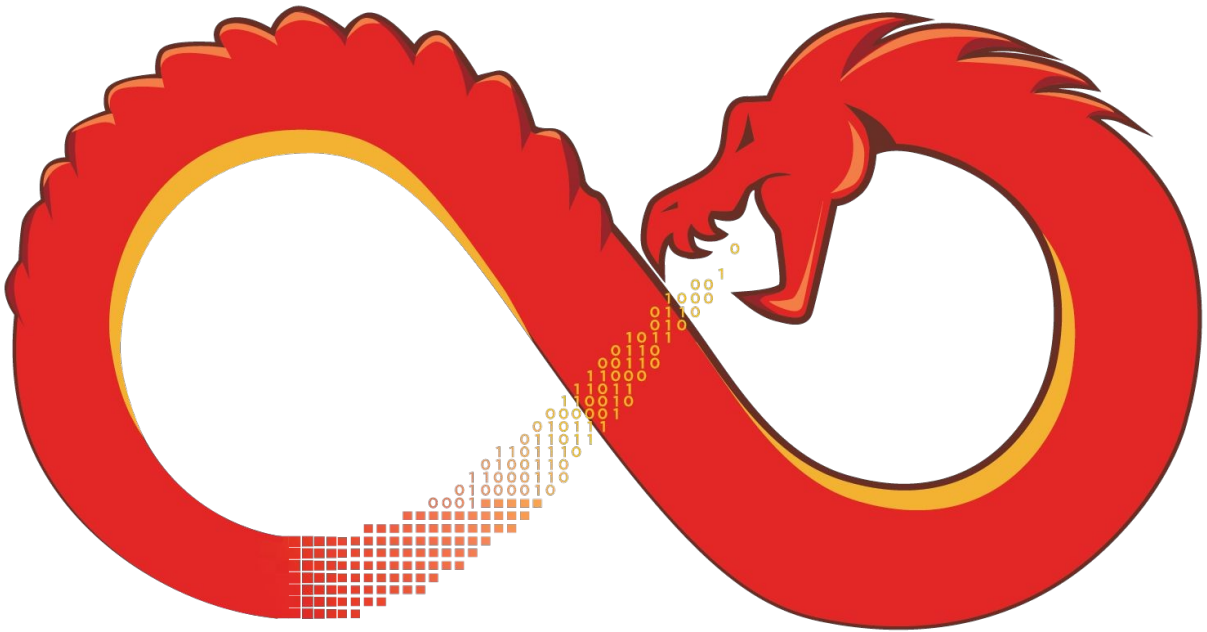
Decompilatori

Ghidra: uno strumento per il reverse engineering sviluppato dalla NSA (National Security Agency). È open source e ampiamente utilizzato.
https://ghidra-sre.org/

IDA Pro: disassemblatore e decompilatore sviluppato da Hex Rays. La versione gratuita è limitata alle architetture x86 e x86-64. Le versioni a pagamento sono costose.
https://hex-rays.com/ida-pro

Binary Ninja: come *IDA Pro*, si tratta di software a pagamento dotato di una versione free limitata.
https://binary.ninja/

La nostra scelta



Installazione di Ghidra

- Scaricare Ghidra dal sito GitHub
- Unzippare il file ottenuto
- Lanciare Ghidra dal file `ghidraRun` (Linux / Mac) o `ghidraRun.bat` (Windows)
  - Su Linux, meglio lanciarlo da un terminale
- Le ultime versioni di Ghidra richiedono per funzionare Java 21 o superiore.
- Su distribuzioni Debian e derivate (Ubuntu, Kali, etc...) dare da `root` i seguenti comandi:
  - `apt install openjdk-21-jdk`

Dimostrazione uso di Ghidra

questa parte non è coperta dalle slide per la difficoltà di tradurre in forma scritta le operazioni svolte su un software con interfaccia grafica

Demo

Decompilazione del file `hello`

File senza tabella dei simboli

- È possibile rimuovere la tabella dei simboli da un file eseguibile (sezioni `.symtab` e `.strtab`) con il comando `strip`.

```
$ strip mystery -o mystery-stripped
$ ls -l mystery mystery-stripped
-rwxr-xr-x. 1 amato amato 16680 Apr 14 11:58 mystery
-rwxr-xr-x. 1 amato amato 14952 Apr 14 12:22 mystery-stripped
$
```

- La decompilazione (e, nelle prossime lezioni, il debugging) di programmi senza tabelle dei simboli è più complesso.

Dimostrazione uso di Ghidra

Demo

Decompilazione del file `mystery-stripped`

Esercitazione uso di Ghidra

Esercizio

Cosa fa il programma `mychallenge-stripped` ?

Gianluca Amato
Università di Chieti-Pescara

Software Security 02

Dal sorgente al codice eseguibile (e ritorno)

FINE

https://cybersecnatlab.it