Introduzione al corso

Sicurezza delle Applicazioni e IA per la Cybersecurity – Modulo 1

Dr. Andrea Bruno, PHD · 2026/2027

Best Practice – Lezione in aula





Best Practice – Lezione teams



- Se non per porre o rispondere domande.
  - Quelle sono sempre benvolute

Chi sono

- **Tecnico informatico** al Dipartimento di Informatica UniSA: dal 2023 gestisco rete, server e laboratori
- **PhD** in Ingegneria dell'Informazione ed Informatica (UniSA), ricerca in Digital Image Forensics
- **Analista della sicurezza**, sviluppatore e project manager in un'azienda software a Praga
- **Docente a contratto** di sicurezza delle applicazioni, programmazione e sistemi distribuiti (UniSA, Federico II, Parthenope)
- **Docente di Software Security** per CyberChallenge all'Università di Salerno, da 4 anni



Perché questo corso

La sicurezza è il mio **lavoro di ogni giorno**.

Quello che vedremo viene da ciò che ho studiato e da ciò che affronto sul campo.

Indice

Introduzione al corso

Richiami di base

01 Dal codice al programma

02 Executable and Linkable Format (ELF)

03 Architettura x86

04 Gestione della memoria e calling convention

05 Debugging

Software e strumenti

06 Raccogliere informazioni dai binari

07 GDB: The GNU Project Debugger

08 GEF: GDB Enhanced Features

09 Pwntools

10 Altri strumenti: Radare2 e Ghidra

Informazioni sul corso

Orario lezioni

- Lunedì: 15:30 – 18:00, Seminario Vescovile – Aula 8
- Giovedì: 14:30 – 16:30, Teams (Codice teams: nhupn27)

Ricevimento

- Online via Teams (accordansi via mail o teams)
- A margine della lezione del lunedì

Contatti

- Mail: andrea.bruno@collaboratore.uniparthenope.it
- Sito web: https://bruand.di.unisa.it/ (eh si la concorrenza)

Modalità di esame

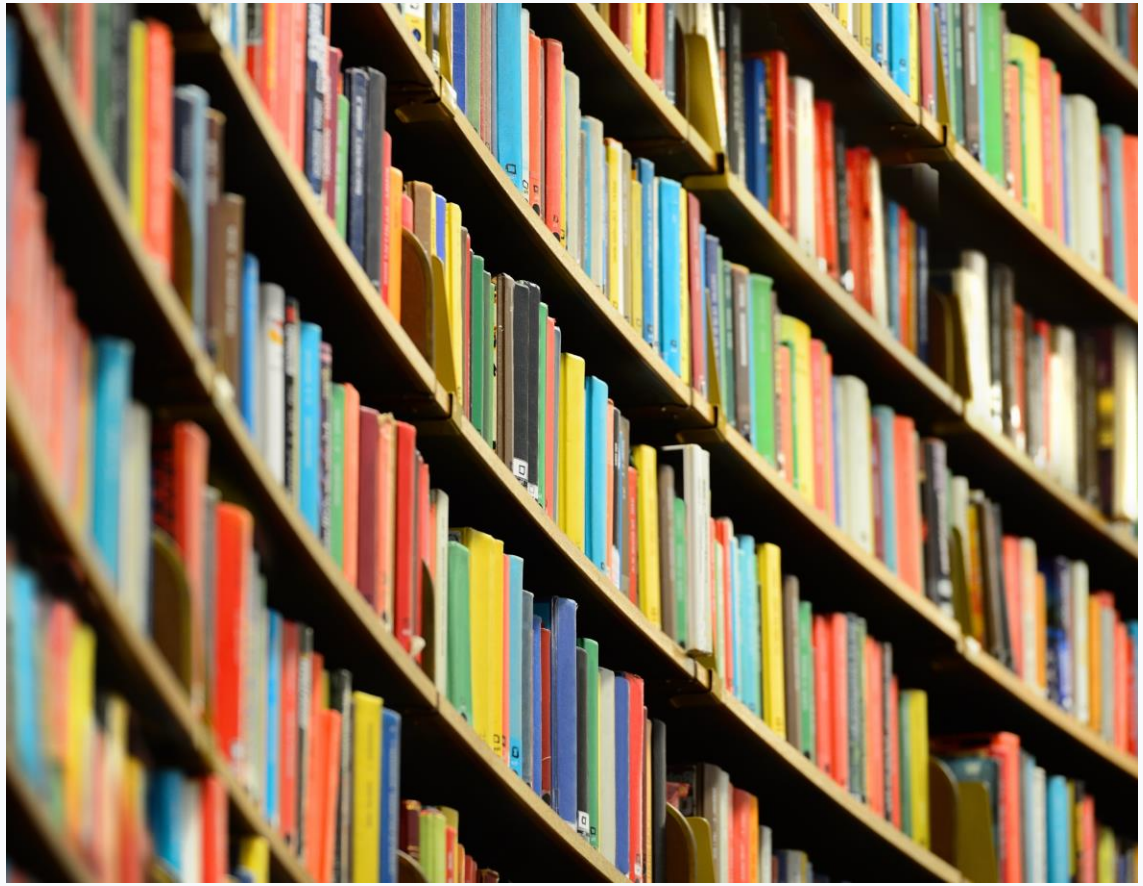
Premessa

- Il corso cambia quest'anno, le tracce con gli argomenti degli scorsi anni saranno disponibili PER I SOLI CORSISTI 2024/2025 e 2025/2026 fino a marzo, poi solo il programma nuovo

Tutti gli appelli

- Scritto e orale obbligatorio
- Orale di base, discussione degli errori dello scritto per confermare il voto dello scritto
- Orale per alzare il voto su tutto il programma
- Ammessi ad orale solo se si supera lo scritto

Materiale del corso



- Disponibile su teams
  - Attenzione, in **cartella 2026/2027**!
  - https://tinyurl.com/SicurezzaApp2627
- Sulla mia pagina personale
  - https://bruand.di.unisa.it/sicurezza-delle-applicazioni-e-intelligenza-artificiale-per-la-cybersecurity-modulo-1/
- Su GitHub
  - https://github.com/SicurezzaAppDisegimUniparthenope/SicurezzaDelleAppMod1_2027
- Libri di testo:
  - Nessun libro di testo!
  - Ci sono:
    - Le slide
    - Gli esempi
    - Eventuali appunti che creo io

vulnbox: l’ambiente per le esercitazioni

- Container Debian con gli **esempi C delle slide**
- Copre memory corruption, format string, return-to-libc e ROP, analisi statica e dinamica, contromisure (NX, stack canary, ASLR)
- Materiale specifico di questo corso, da usare in aula insieme al docente
- L’immagine è già pubblicata su GitHub Container Registry: **niente clone, niente build**

1. **Installa Docker o Podman**
   su Windows o Mac: slide 10 e 11

2. **Avvia il container**
   un pull e un run dal terminale

3. **Collegati via SSH**
   utente student, password student

Docker o Podman su Windows

Opzione A: Docker Desktop

```
# PowerShell come amministratore
> wsl --install
# riavvia il PC, poi:
> winget install -e --id Docker.DockerDesktop
# apri Docker Desktop e verifica:
> docker run hello-world
```

Richiede WSL 2 e la virtualizzazione attiva nel BIOS

Opzione B: Podman

```
# PowerShell
> winget install -e --id RedHat.Podman
# apri un nuovo terminale, poi:
> podman machine init
> podman machine start
> podman run hello-world
```

Usa anch'esso WSL 2: se manca, lo installa podman machine init

Con Podman i comandi del corso restano uguali: basta scrivere **`podman`** al posto di **`docker`**

Docker o Podman su Mac Apple Silicon

Opzione A: Docker Desktop

```
# Terminale, con Homebrew (brew.sh)
$ brew install --cask docker-desktop
# avvia Docker Desktop:
$ open -a Docker
# quando è pronto, verifica:
$ docker run hello-world
```

Richiede macOS 14 o successivo. In alternativa: docker.com, versione «Apple Chip»

Opzione B: Podman

```
# Terminale, con Homebrew (brew.sh)
$ brew install podman
# crea e avvia la VM Linux:
$ podman machine init
$ podman machine start
$ podman run hello-world
```

La VM va riavviata dopo ogni riavvio del Mac: podman machine start

Su Apple Silicon serve l'immagine **`:latest-arm64`** di vulnbox: i comandi sono nella prossima slide

Avviare vulnbox

Windows, Linux e Mac Intel (amd64): porta 2201

```
$ docker pull ghcr.io/sicurezzaappdisegimuniparthenope/vulnbox:latest
$ docker run -d --name vulnbox -p 127.0.0.1:2201:22 --cap-add=SYS_PTRACE \
    ghcr.io/sicurezzaappdisegimuniparthenope/vulnbox:latest
$ ssh -p 2201 student@localhost     # password: student
```

Mac Apple Silicon (arm64): porta 2202

```
$ docker pull ghcr.io/sicurezzaappdisegimuniparthenope/vulnbox:latest-arm64
$ docker run -d --name vulnbox-arm64 -p 127.0.0.1:2202:22 --cap-add=SYS_PTRACE \
    --security-opt seccomp=unconfined \
    ghcr.io/sicurezzaappdisegimuniparthenope/vulnbox:latest-arm64
$ ssh -p 2202 student@localhost     # password: student
```

Con `exit` si chiude solo la sessione SSH: il container resta acceso (vedi slide successiva)

vulnbox: accesso root, versione e aggiornamento

Accesso root e versione dell'immagine

```
$ ssh -p 2201 vulnbox@localhost     # password: vulnbox
$ sudo -s                           # shell di root (sudo NOPASSWD)
$ vulnbox_ver                       # versione e commit dell'immagine
  Vulnbox 202610191430 (a2c025f)
```

Aggiornare l'immagine se già scaricata

```
$ docker pull ghcr.io/sicurezzaappdisegimuniparthenope/vulnbox:latest
$ docker rm -f vulnbox                              # rimuove il container vecchio
$ docker run -d --name vulnbox -p 127.0.0.1:2201:22 --cap-add=SYS_PTRACE \
    ghcr.io/sicurezzaappdisegimuniparthenope/vulnbox:latest
# arm64: usa :latest-arm64, nome vulnbox-arm64 e porta 2202
```

L'immagine non si aggiorna da sola: docker pull scarica l'ultima release, poi va ricreato il container. Verifica con vulnbox_ver quale release è attiva. L'utente student resta senza sudo.

Chiudere vulnbox e problemi comuni

A fine sessione

```
$ docker stop vulnbox    # lo ferma
$ docker start vulnbox   # lo riaccende
$ docker ps -a           # elenca i container
$ docker rm vulnbox      # lo elimina
$ docker rmi <immagine> # libera il disco
```

Il container resta acceso in background finché non lo fermi.

Per la variante arm64 scrivi `vulnbox-arm64` al posto di `vulnbox`

Eliminare il container non cancella l'immagine scaricata

Problemi comuni

`container name "vulnbox" is already in use`

Esiste già un container con quel nome: `docker rm vulnbox` oppure un altro `--name`

`no image found … architecture "arm64"`

Sei su Apple Silicon: usa l'immagine `:latest-arm64`
Il tag :latest contiene solo l'immagine amd64

Servono gli esempi a 32 bit su Apple Silicon

Immagine amd64 in emulazione con `--platform linux/amd64` su pull e run; gdb live e strace non funzionano

Dettagli, credenziali ed elenco degli esempi: `docker/vulnbox/README.md` nel repository del corso

Programma del corso

I Parte: Sicurezza del software

- Stack e heap: layout della memoria
- Analisi statica e dinamica, anti-debug
- Memory corruption e shellcode injection
- Format-string vulnerabilities
- Contromisure: NX, stack canary, ASLR
- Return-to-libc, ROP, JOP

II Parte: Sicurezza delle applicazioni web

- Cross site request forgery (CSRF)
- Cross site scripting (XSS)
- SQL injection
- Clickjacking
- Host header validation
- Referrer policy
- Cross-origin opener policy (COOP)
- Session security
- User-uploaded content

Indice

Introduzione al corso

Richiami di base

01 **Dal codice al programma**

02 Executable and Linkable Format (ELF)

03 Architettura x86

04 Gestione della memoria e calling convention

05 Debugging

Software e strumenti

06 Raccogliere informazioni dai binari

07 GDB: The GNU Project Debugger

08 GEF: GDB Enhanced Features

09 Pwntools

10 Altri strumenti: Radare2 e Ghidra

Dal codice al programma

Richiami di base

Cosa succede tra il tasto Invio e il primo segfault

Dal codice al programma

La compilazione di un programma C è un processo in quattro fasi:

1 Preprocessing

`gcc -E hello.c`

Espande le direttive # (include, define)

→ **codice C espanso**

2 Compilazione

`gcc -S hello.c`

Traduce il codice C in assembly

→ **hello.s**

3 Assemblaggio

`gcc -c hello.c`

Traduce l'assembly in codice macchina

→ **hello.o**

4 Linking

`gcc -o hello hello.o`

Unisce i file oggetto e collega le librerie

→ **hello (eseguibile)**

Il compilatore GCC

- In questo corso useremo il compilatore gcc…
  - è il compilatore C della GNU Compiler Collection
  - è disponibile sui principali sistemi operativi
    - è il compilatore standard di tutte le distribuzioni UNIX/Linux
- La documentazione completa è disponibile qui:

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

```
$ gcc --version
$ gcc hello.c -o hello
$ ./hello
Hello, world!
```

← mostra la versione installata

← compila hello.c nell'eseguibile hello

← esegue il programma

← output del programma

Dal codice al programma: preprocessing

- Nella prima fase vengono interpretate le direttive del preprocessore (in C iniziano con ‘#’):



Dal codice al programma: compilazione

- Nella seconda fase il codice preprocessato viene tradotto in istruzioni assembly:



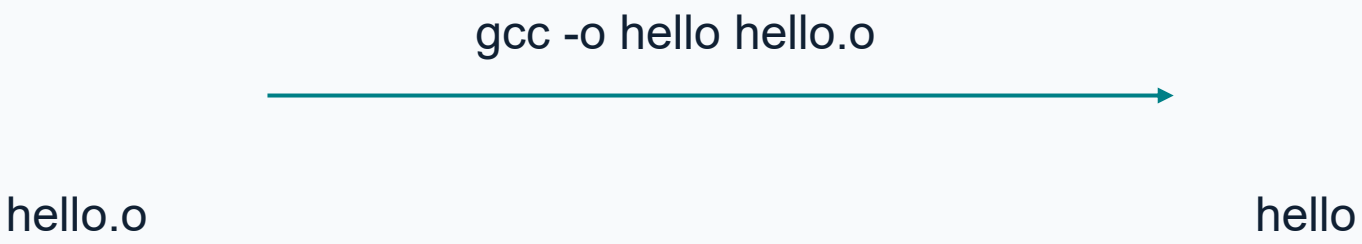
Dal codice al programma: assemblaggio

- Nella fase di assemblaggio le istruzioni assembly vengono tradotte in codice macchina (codice oggetto):



Dal codice al programma: linking

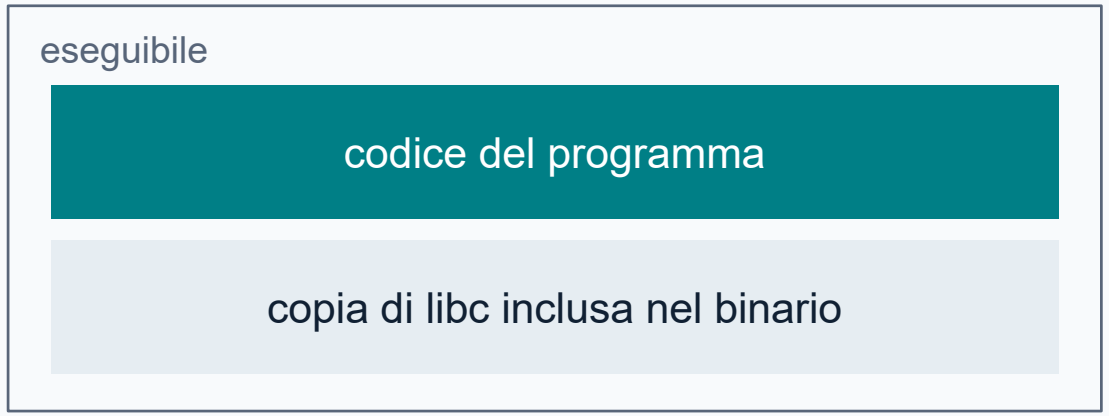
- Nell’ultima fase uno o più file oggetto vengono combinati in un unico eseguibile
- Nel file generato vengono aggiunti i riferimenti (link) alle librerie usate



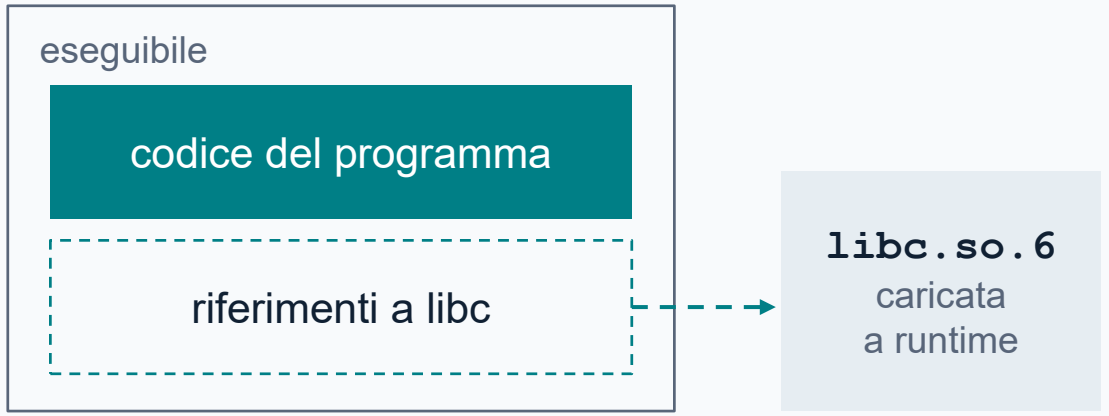
Linking statico e dinamico

- Nella fase di linking si possono seguire due approcci:
  - Linking statico
    - I binari sono autosufficienti e non dipendono da librerie esterne
  - Linking dinamico
    - I binari usano librerie di sistema caricate quando servono
    - Servono meccanismi per rilocare dinamicamente il codice

Linking statico



Linking dinamico



Indice

Introduzione al corso

Richiami di base

01 Dal codice al programma

02 **Executable and Linkable Format (ELF)**

03 Architettura x86

04 Gestione della memoria e calling convention

05 Debugging

Software e strumenti

06 Raccogliere informazioni dai binari

07 GDB: The GNU Project Debugger

08 GEF: GDB Enhanced Features

09 Pwntools

10 Altri strumenti: Radare2 e Ghidra

Executable and Linkable Format (ELF)

Richiami di base

Non è un elfo, ma di magia ne fa parecchia

Executable and Linkable Format (ELF)

- L'Executable and Linkable Format (ELF) è un formato di file comune per i file oggetto
- Esistono tre tipi di file oggetto:
  - File rilocabili: contengono codice e dati da collegare con altri file oggetto per creare un eseguibile o un oggetto condiviso
  - File eseguibili: contengono un programma pronto per l'esecuzione
  - File oggetto condivisi, che possono essere:
    - collegati con altri file rilocabili e condivisi per ottenere un nuovo file oggetto
    - usati dal linker dinamico, insieme ad altri eseguibili e file oggetto, per creare l'immagine di un processo

`main.o`

File rilocabile

prodotto da gcc -c main.c

`programma`

File eseguibile

prodotto dal linker, pronto per l'esecuzione

`libc.so.6`

Oggetto condiviso

usato dal linker dinamico a runtime

Struttura di un file ELF

- Ogni file ELF è strutturato così:
  - un ELF header che descrive il contenuto del file
  - una Program header table con le informazioni per creare l'immagine del processo
  - una sequenza di sezioni con quanto serve al linking (istruzioni, dati, tabella dei simboli, informazioni di rilocazione…)
  - una Section header table che descrive le sezioni precedenti



ELF: le sezioni principali

- `.text`: contiene le istruzioni eseguibili del programma
- `.bss`: contiene i dati non inizializzati che fanno parte dell'immagine in memoria del programma
- `.data`, `.data1`: contengono i dati inizializzati che fanno parte dell'immagine in memoria del programma
- `.rodata`, `.rodata1`: come `.data` e `.data1`, ma per dati in sola lettura
- `.symtab`: contiene la tabella dei simboli del programma
- `.dynamic`: fornisce le informazioni per il linking

Disposizione tipica delle sezioni in un eseguibile ELF



La sezione tratteggiata (.bss) non occupa spazio nel file: viene azzerata in memoria

Indice

Introduzione al corso

Richiami di base

01 Dal codice al programma

02 Executable and Linkable Format (ELF)

03 Architettura x86

04 Gestione della memoria e calling convention

05 Debugging

Software e strumenti

06 Raccogliere informazioni dai binari

07 GDB: The GNU Project Debugger

08 GEF: GDB Enhanced Features

09 Pwntools

10 Altri strumenti: Radare2 e Ghidra

Architettura x86

Richiami di base

Quasi cinquant'anni di retrocompatibilità in un solo processore

Il set di istruzioni x86

- Vediamo un sottoinsieme piccolo ma utile delle istruzioni e delle direttive dell'assembly x86
- Una descrizione completa è disponibile qui:
  - Intel x86 Instruction Set Reference
    - http://www.felixcloutier.com/x86/
  - Manuali Intel per sviluppatori
    - http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

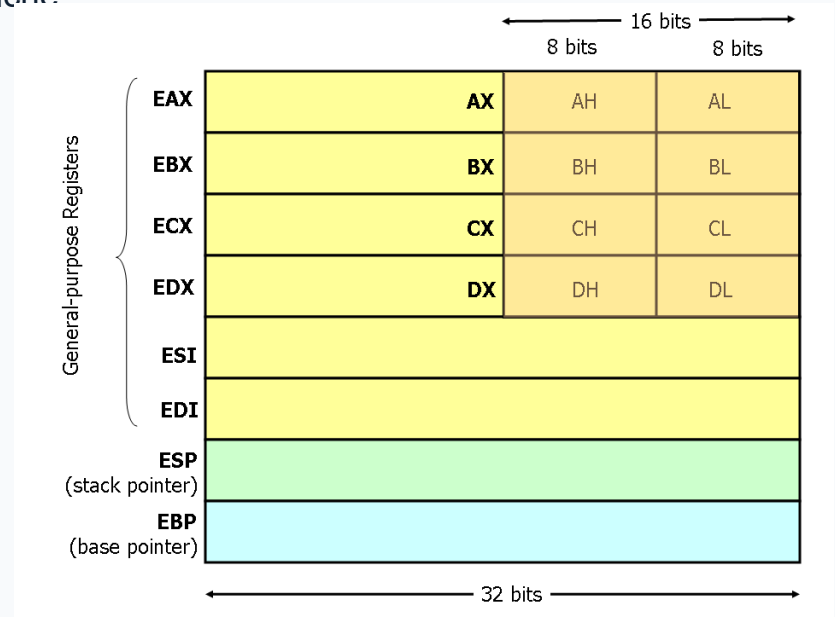
```
mov   eax,   [ebp+8]
```

istruzione **destinazione (op1)** **sorgente (op2)**

Sintassi Intel: prima la destinazione, poi la sorgente. Qui EAX riceve il valore che si trova all'indirizzo EBP+8

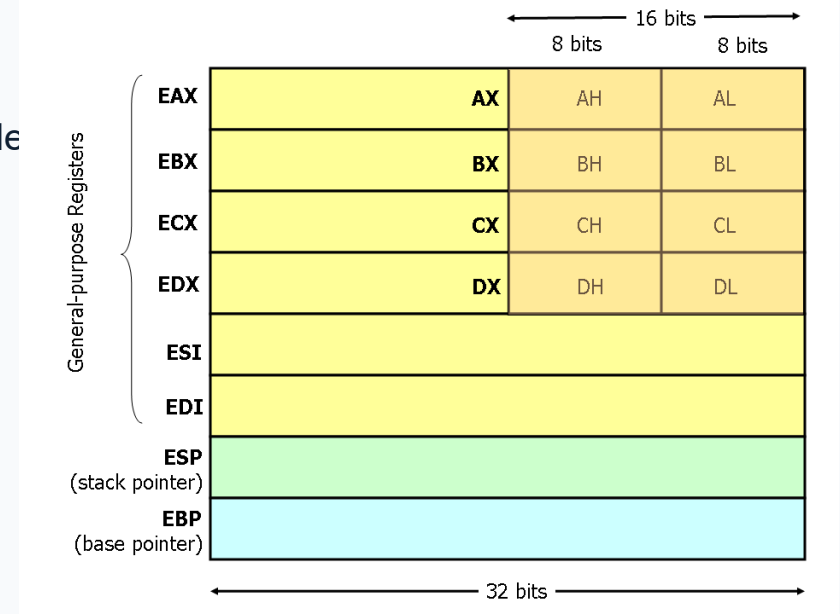
Registri x86-32

- I processori x86-32 hanno otto registri general purpose a 32 bit
- I nomi dei registri sono in gran parte storici…
  - EAX era detto accumulatore perché usato da molte operazioni aritmetiche
  - ECX era detto contatore perché conteneva l'indice dei cicli
- Due registri hanno usi speciali:
  - lo stack pointer (ESP)
  - il base pointer (EBP)



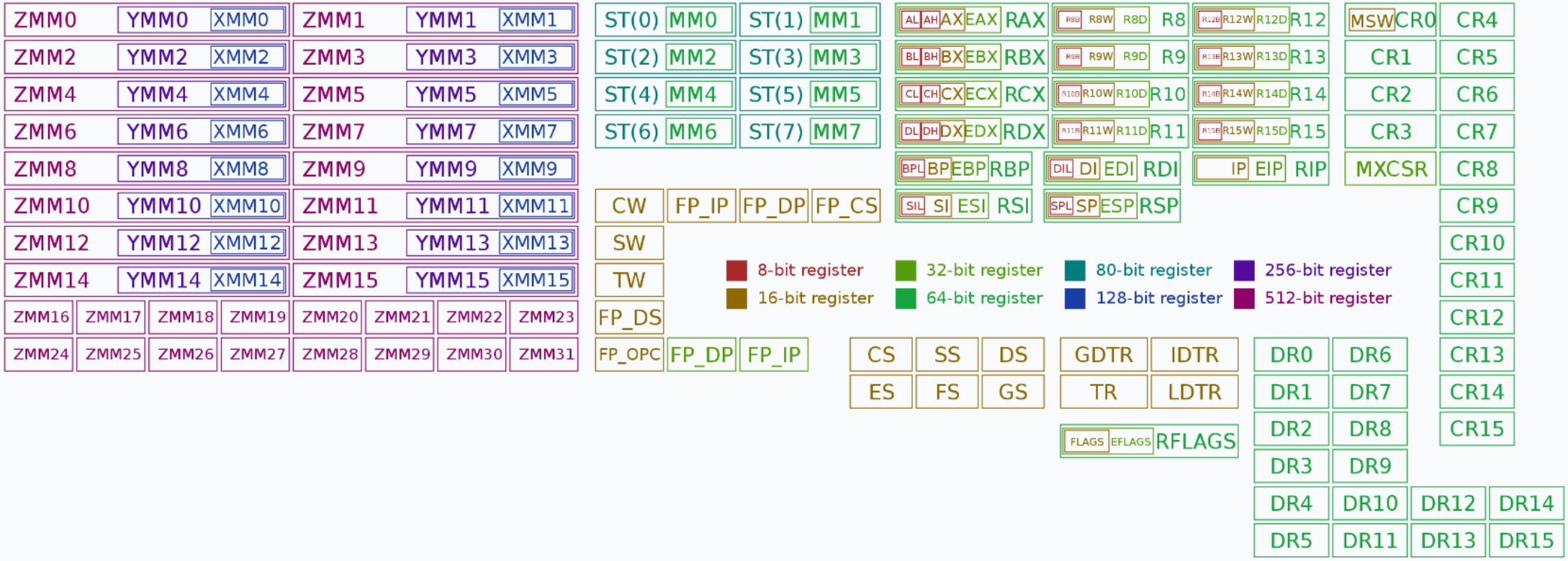
Registri x86-32

- Per i registri EAX, EBX, ECX ed EDX si possono usare delle sottoparti
- Ad esempio:
  - AX indica i 2 byte meno significativi di EAX
  - AL indica il byte meno significativo di AX
  - AH indica il byte più significativo di AX
- Questi sotto-registri sono un retaggio delle vecchie versioni a 16 bit de



Registri x86-64

- L’architettura x86-64 mette a disposizione più registri:



Istruzioni x86

- Istruzioni di spostamento dati
  - `mov op1,op2`: copia in op1 il valore di op2
  - `push op1`: mette op1 in cima allo stack
  - `pop op1`: toglie l'elemento in cima allo stack e lo copia in op1
  - `lea op1,op2`: carica in op1 l'indirizzo calcolato da op2

Sintassi Intel: il primo operando è la destinazione, il secondo la sorgente. Nella sintassi AT&T (default di gcc e objdump) l'ordine è invertito.

Esempio: una sequenza di istruzioni

```
mov eax, 5
```
eax = 5

copia un valore nel registro

```
push eax
```
[esp] = 5

esp scende di 4, poi scrive

```
pop ebx
```
ebx = 5

legge da [esp], poi esp sale di 4

```
lea ecx, [ebx+8]
```
ecx = 13

calcola l'indirizzo, senza leggere la memoria

Istruzioni x86

- Istruzioni aritmetiche e logiche
  - `add op1,op2`: salva in op1 il risultato di op1+op2
  - `sub op1,op2`: salva in op1 il risultato di op1-op2
  - `and op1,op2`
  - `or op1,op2`
  - `xor op1,op2`
  - ...

`and`, **`or`**, **`xor`** lavorano bit a bit: eseguono l'operazione logica sugli operandi e salvano il risultato nel primo operando

Esempio su 8 bit: `al = 1100 1010  bl = 1010 0110`

`and al, bl`

`al = 1000 0010`

1 dove entrambi i bit sono 1

`or al, bl`

`al = 1110 1110`

1 dove almeno un bit è 1

`xor al, bl`

`al = 0110 1100`

1 dove i bit sono diversi

Istruzioni x86

- Istruzioni di controllo del flusso
  - `jmp op`: salta sempre all'indirizzo indicato da op
  - `cmp op1,op2`: confronta op1 e op2 e aggiorna i flag di stato
  - `j<condizione> op`: salta a op solo se la condizione sui flag è vera

> **`cmp`** calcola op1 − op2 senza salvare il risultato: aggiorna solo i flag (ZF, SF, CF, OF) che i salti condizionati leggono.

Esempio: salti condizionati dopo cmp eax, ebx

`je / jz`

`eax == ebx`

salta se ZF = 1

`jne / jnz`

`eax != ebx`

salta se ZF = 0

`jg / jl`

`eax > / < ebx`

confronto con segno

`ja / jb`

`eax > / < ebx`

confronto senza segno

Indice

Introduzione al corso

Richiami di base

01 Dal codice al programma

02 Executable and Linkable Format (ELF)

03 Architettura x86

04 **Gestione della memoria e calling convention**

05 Debugging

Software e strumenti

06 Raccogliere informazioni dai binari

07 GDB: The GNU Project Debugger

08 GEF: GDB Enhanced Features

09 Pwntools

10 Altri strumenti: Radare2 e Ghidra

Gestione della memoria e calling convention

Richiami di base

Chi passa gli argomenti, e chi pulisce lo stack dopo la festa

Gestione della memoria

- Molti linguaggi moderni permettono di usare i tipi di dato senza sapere come sono rappresentati
- Allo stesso modo, spesso il programmatore ignora dove i dati vengono memorizzati (allocati)…
  - il compilatore prende delle decisioni, ma a runtime l'allocazione è controllata da sistema operativo e CPU
- In alcuni linguaggi, come il C, la memoria può essere gestita dal programmatore:
  - la memoria può essere allocata e deallocata dinamicamente
  - si può ottenere l'indirizzo di memoria delle variabili (puntatori)
- Se x è una variabile, `&x` indica il puntatore a x, cioè l'indirizzo di memoria in cui x è memorizzata

Allocazione della memoria

- Consideriamo questo semplice programma C:

```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);
}
```

Assumiamo che:

- *int occupa 4 byte*
- *char occupa 1 byte*
- *short occupa 2 byte*
- *long occupa 8 byte*

Allocazione della memoria

- Consideriamo questo semplice programma C:

```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);
}
```

Dichiarazioni delle variabili

Assumiamo che:

- *int occupa 4 byte*
- *char occupa 1 byte*
- *short occupa 2 byte*
- *long occupa 8 byte*

Allocazione della memoria

- Consideriamo questo semplice programma C:

```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);
}
```

Indirizzi delle variabili

Assumiamo che:

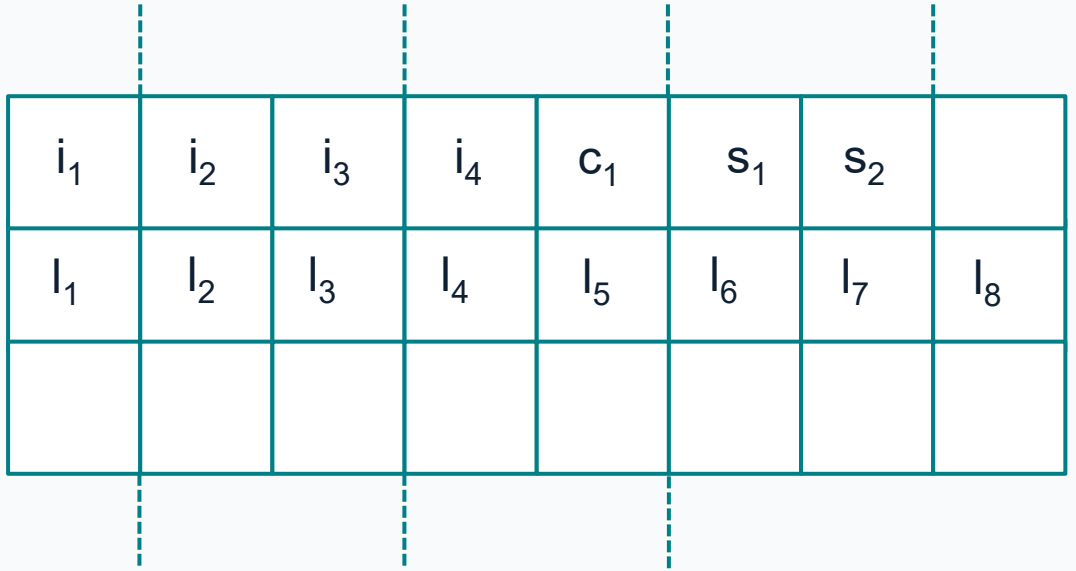
- *int occupa 4 byte*
- *char occupa 1 byte*
- *short occupa 2 byte*
- *long occupa 8 byte*

Allocazione della memoria

- La memoria si rappresenta di solito come una sequenza di byte:



In un’architettura a n·8 bit, i byte sono raggruppati a gruppi di n:



Allocazione della memoria

```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);
}
```

Eseguiamo il programma per osservare come i dati vengono allocati in memoria

Allocazione della memoria

- gcc può usare politiche di allocazione diverse:

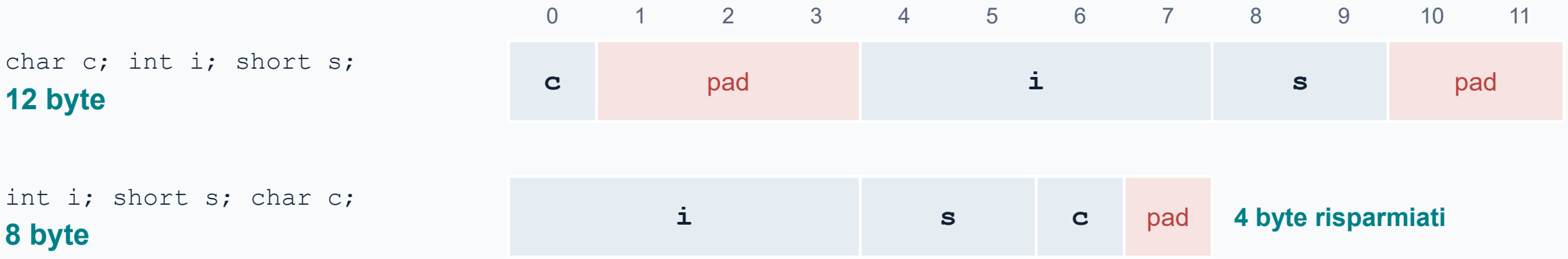
```
[CC> gcc -o alignment alignment.c
[CC> ./alignment
i is allocated at 0x7ffee117e7cc
c is allocated at 0x7ffee117e7cb
s is allocated at 0x7ffee117e7c8
l is allocated at 0x7ffee117e7c0
[CC>
```

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

Allineamento dei dati

- Il compilatore può aggiungere padding o cambiare l'ordine dei dati in memoria
- Va trovato un compromesso tra velocità e uso della memoria
- I compilatori C offrono molte ottimizzazioni opzionali

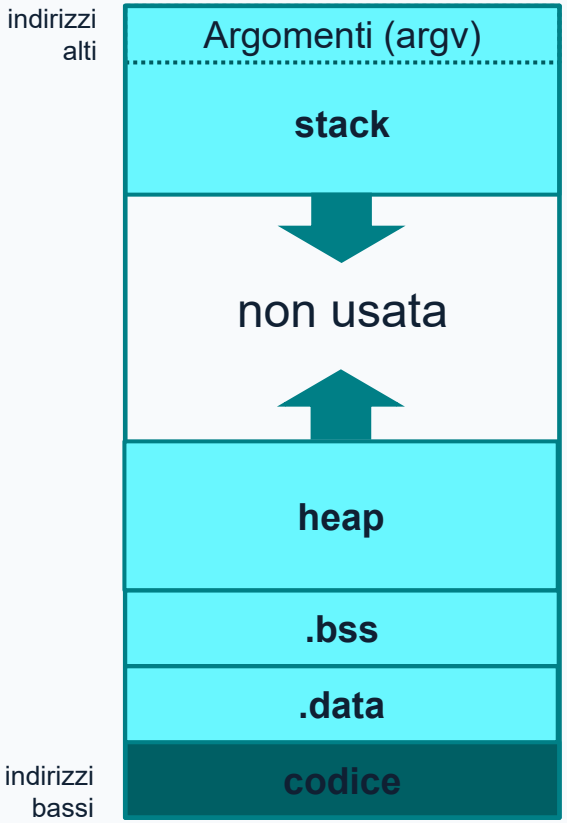
Esempio: la stessa struct con i campi in ordine diverso



Ipotesi: int allineato a 4 byte, short a 2, char a 1 (x86, gcc). Il padding finale allinea la struct al suo campo più grande.

Segmenti di memoria

- A ogni processo (un programma in esecuzione) viene assegnata memoria per dati e codice.
- Questa memoria è divisa in segmenti:
  - stack: per le variabili locali
  - heap: per la memoria dinamica
  - segmento dati:
    - variabili globali non inizializzate (.bss)
    - variabili globali inizializzate (.data)
  - segmento di codice

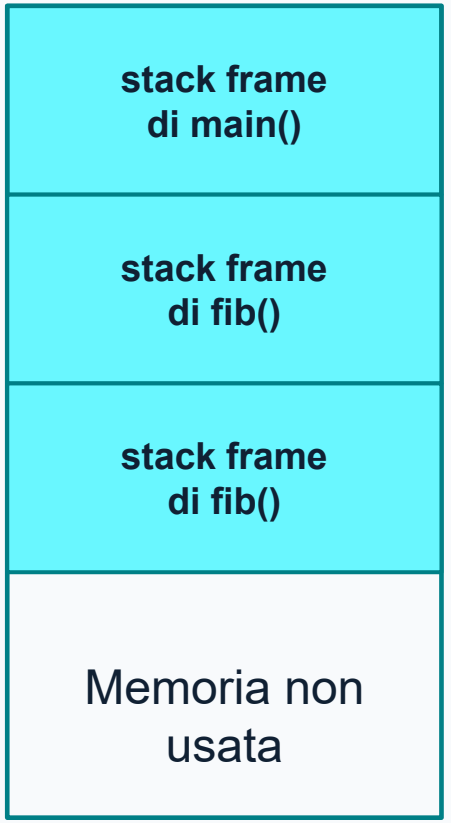


Lo stack

- Lo stack è una sequenza di stack frame (o record di attivazione), uno per ogni chiamata di funzione:
  - allocato alla chiamata
  - deallocato al termine

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

int fib(int n) {
  int f1;
  int f2;
  if (n<=2) {
      return 1;
  } else {
      f1 = fib( n: n-1);
      f2 = fib( n: n-2);
      return f1+f2;
  }
}
```



Lo stack

- La struttura esatta dello stack dipende da architettura, sistema operativo e compilatore.
- Per semplicità, in questo corso ci concentriamo sulle architetture x86 (32 e 64 bit) e sul compilatore gcc su Linux
- Approfondimenti:
  - https://docs.microsoft.com/en-us/cpp/cpp/argument-passing-and-naming-conventions
  - http://refspecs.linuxbase.org/

Le scelte del corso

Architettura

x86 · x86-64

32 e 64 bit: registri e dimensione delle parole

Sistema operativo

Linux

convenzioni di chiamata System V

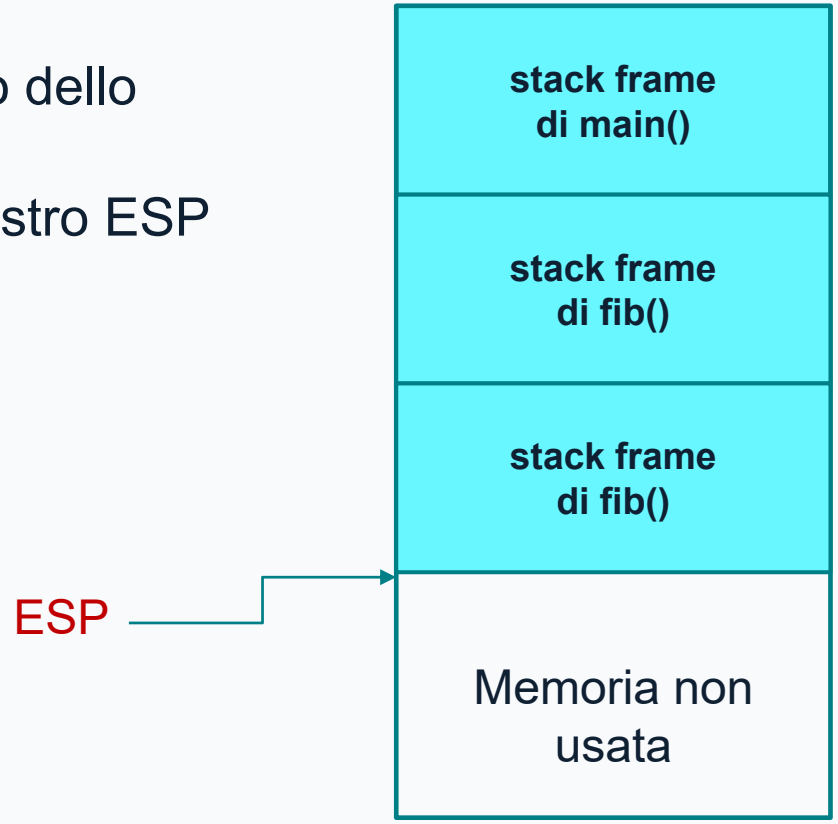
Compilatore

gcc

layout dei frame e ottimizzazioni

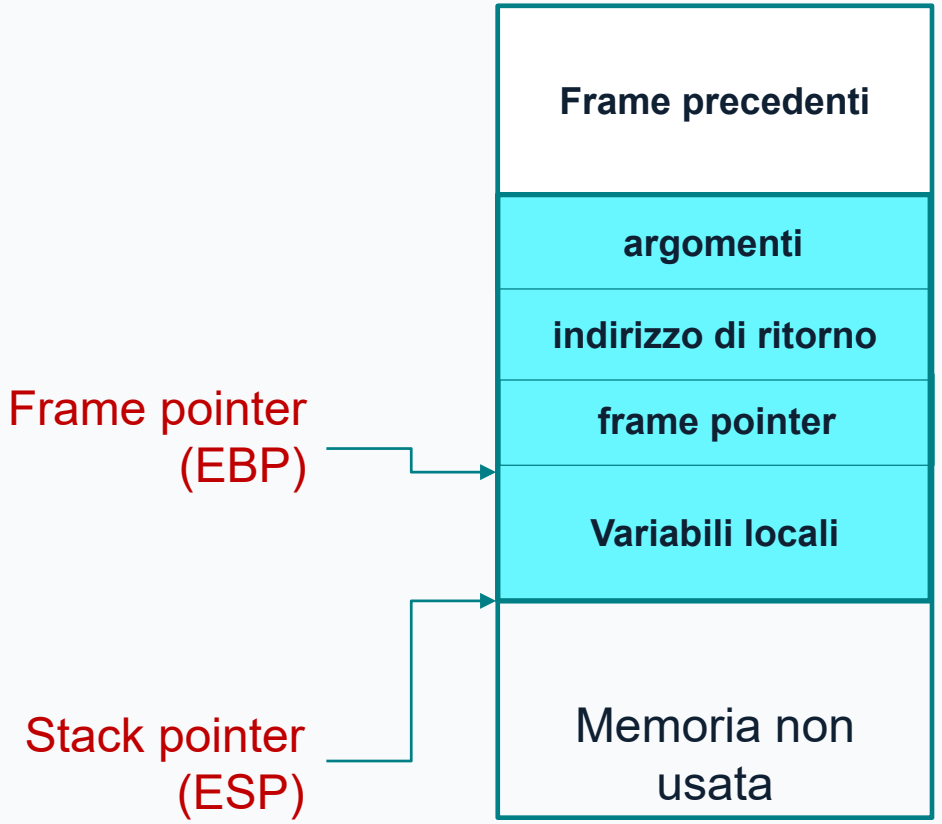
Lo stack

- Di solito lo stack cresce verso il basso
- Lo stack pointer (SP) punta all'ultimo elemento dello stack
- Su x86 lo stack pointer è memorizzato nel registro ESP (Extended Stack Pointer)



Stack frame (x86)

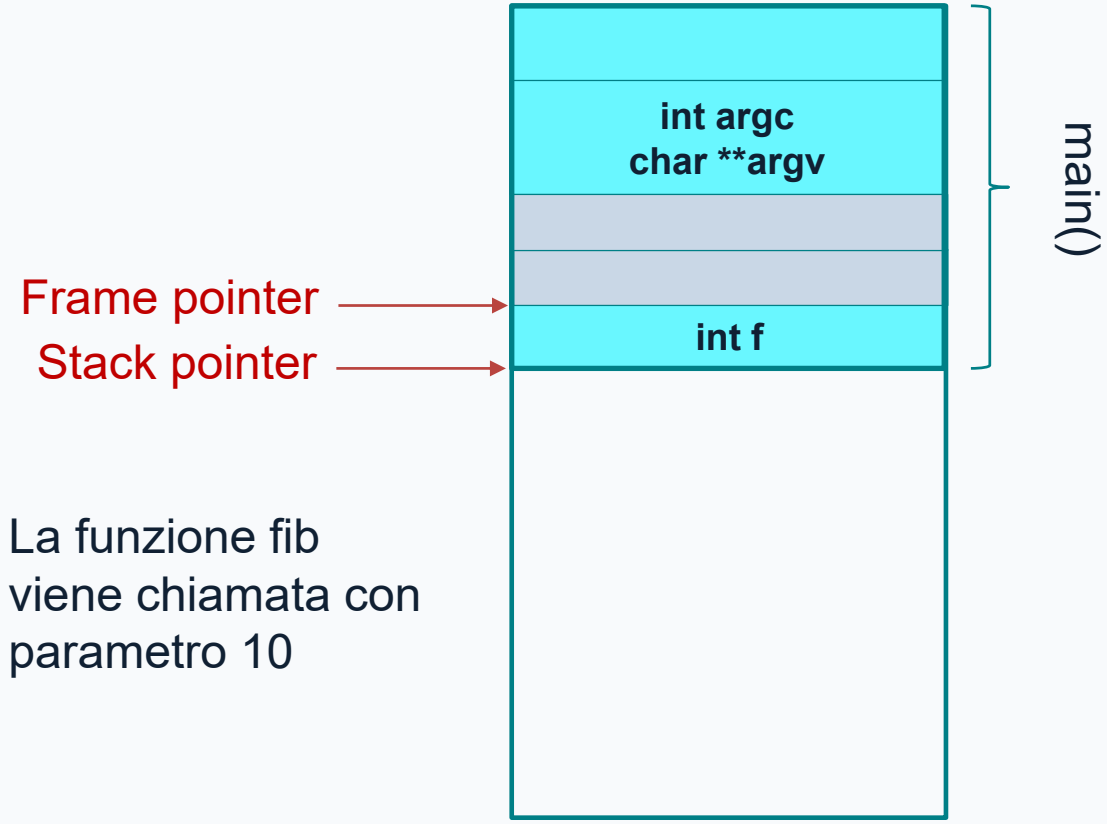
- Su x86 ogni stack frame contiene:
  - gli argomenti della funzione
  - le variabili locali
  - le copie dei registri da ripristinare:
    - l'indirizzo di ritorno
    - il frame pointer precedente
- Il frame pointer, detto Extended Base Pointer (EBP), è il punto di riferimento per le variabili locali



Stack frame: esempio

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

int fib(int n) {
  int f1;
  int f2;
  if (n<=2) {
      return 1;
  } else {
      f1 = fib( n: n-1);
      f2 = fib( n: n-2);
      return f1+f2;
  }
}
```

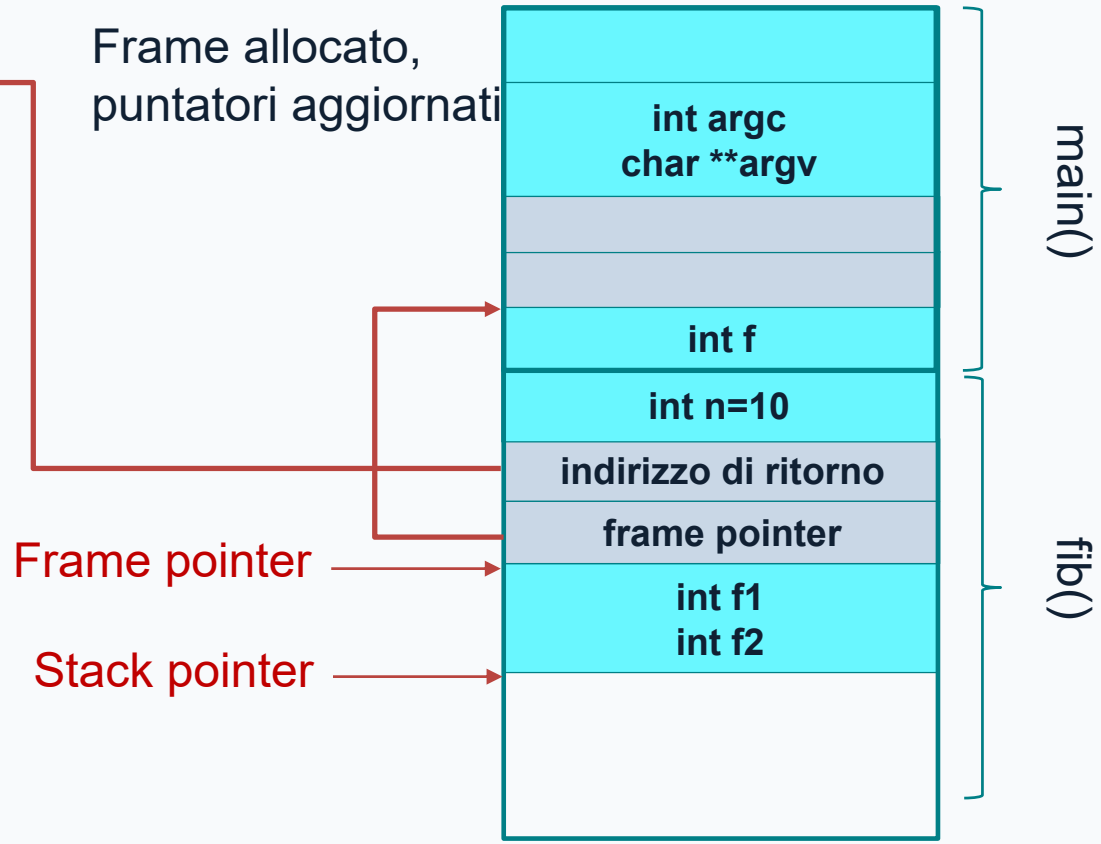


La funzione fib viene chiamata con parametro 10

Stack frame: esempio

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

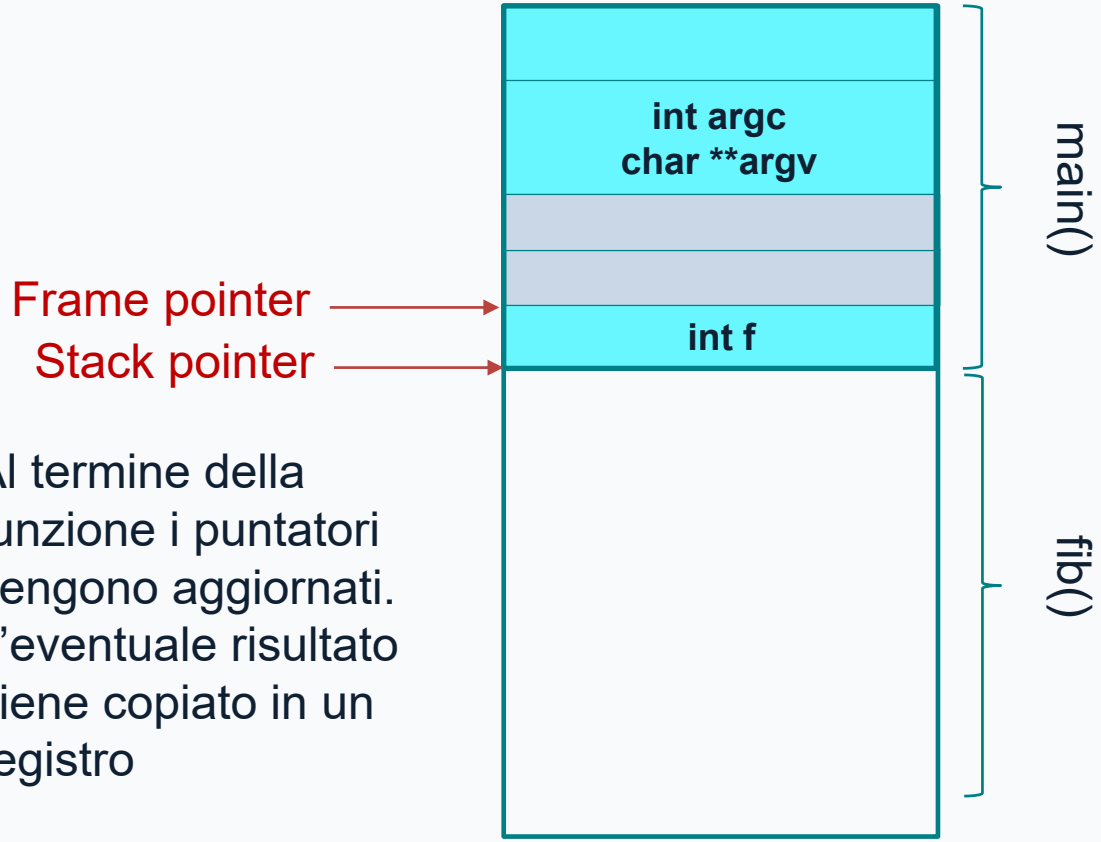
int fib(int n) {
  int f1;
  int f2;
  if (n<=2) {
      return 1;
  } else {
      f1 = fib( n: n-1);
      f2 = fib( n: n-2);
      return f1+f2;
  }
}
```



Stack frame: esempio

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

int fib(int n) {
  int f1;
  int f2;
  if (n<=2) {
      return 1;
  } else {
      f1 = fib( n: n-1);
      f2 = fib( n: n-2);
      return f1+f2;
  }
}
```



Al termine della funzione i puntatori vengono aggiornati. L'eventuale risultato viene copiato in un registro

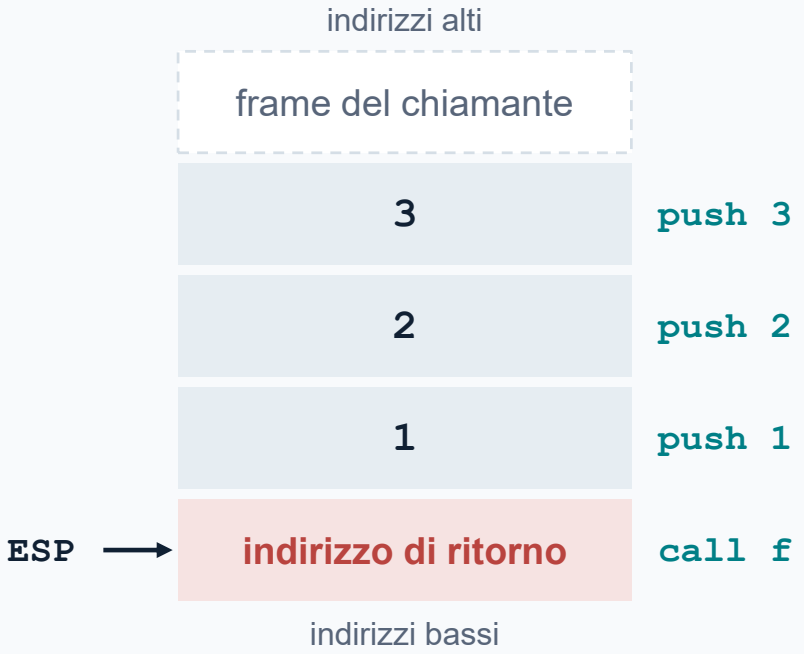
Stack frame (x86-64)

- Nella versione a 64 bit di x86:
  - i registri sono estesi a 64 bit
  - si aggiungono 8 nuovi registri
- non tutti gli argomenti stanno sullo stack:
  - i primi 6 sono passati nei registri
  - gli altri sono messi sullo stack (come su x86)
- i puntatori EBP ed ESP si chiamano RBP e RSP
- sotto RSP c'è una red zone di 128 byte
  - si può usare per altre variabili locali e non viene modificata da gestori di interrupt, eccezioni e segnali

C declaration: cdecl

- cdecl (C declaration) è una calling convention usata da molti compilatori per x86
  - Una calling convention descrive come una funzione riceve i parametri dal chiamante e come restituisce i risultati
- In cdecl gli argomenti passano sullo stack, mentre i valori restituiti passano nei registri
- Gli argomenti vengono inseriti sullo stack da destra verso sinistra
- Il chiamante pulisce lo stack quando la funzione chiamata termina

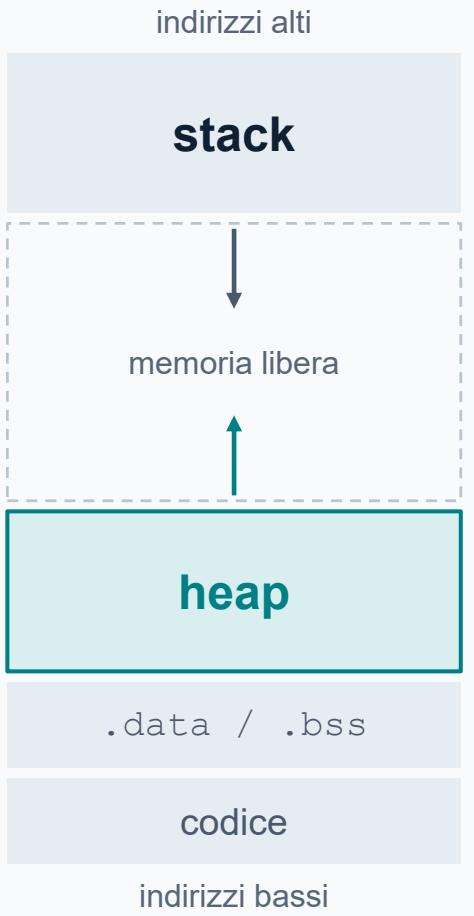
Chiamata f(1, 2, 3) con cdecl



Al termine: valore restituito in eax
poi il chiamante esegue `add esp, 12`

Lo heap

- Allocare e deallocare memoria sullo stack è molto veloce
  - Però questa memoria non si può usare dopo il termine della funzione
- Lo heap contiene i dati allocati dinamicamente che sopravvivono alle chiamate di funzione:
  - quest'area è responsabilità del programmatore



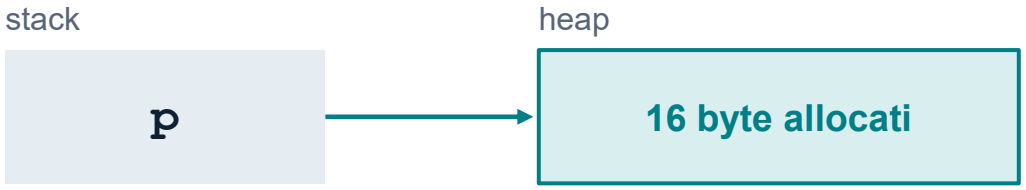
Variabili locali
allocazione veloce, liberate al termine della funzione

Dati dinamici
malloc / free: restano finché il programmatore non li libera

Funzioni di gestione della memoria

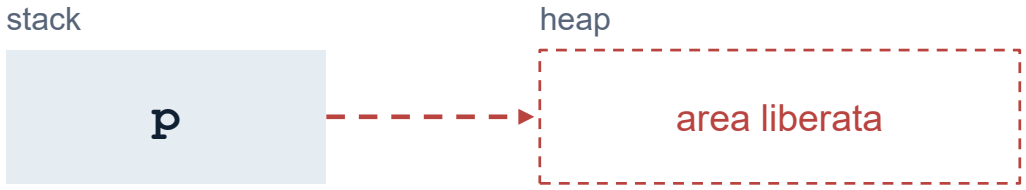
- Le funzioni C di base per gestire la memoria sono:
  - `malloc(int)`: dato un intero n, alloca un'area contigua di n byte e restituisce un puntatore all'area
  - `free(void*)`: dealloca la memoria associata a un puntatore

```
int *p = malloc(16);
```



malloc restituisce l'indirizzo dell'area; p lo conserva nello stack

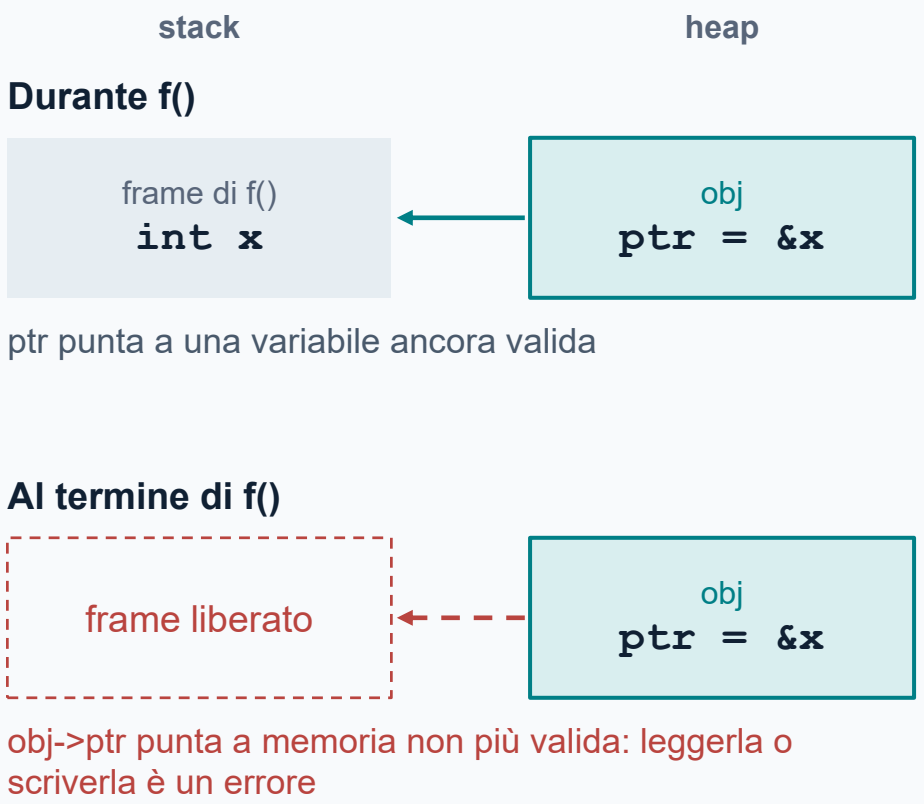
```
free(p);
```



p contiene ancora lo stesso indirizzo: è un **dangling pointer**
Dopo free conviene scrivere `p = NULL;`

Puntatori a stack e heap

- Bisogna sempre conservare un riferimento alle aree allocate nello heap
  - Altrimenti non possiamo più usarle!
- Questi riferimenti (puntatori) possono stare nello stack o nello heap
- Potremmo memorizzare nello heap puntatori allo stack
  - È pericoloso: la memoria puntata potrebbe essere già stata rilasciata!



Indice

Introduzione al corso

Richiami di base

01 Dal codice al programma

02 Executable and Linkable Format (ELF)

03 Architettura x86

04 Gestione della memoria e calling convention

05 **Debugging**

Software e strumenti

06 Raccogliere informazioni dai binari

07 GDB: The GNU Project Debugger

08 GEF: GDB Enhanced Features

09 Pwntools

10 Altri strumenti: Radare2 e Ghidra

Debugging

Richiami di base

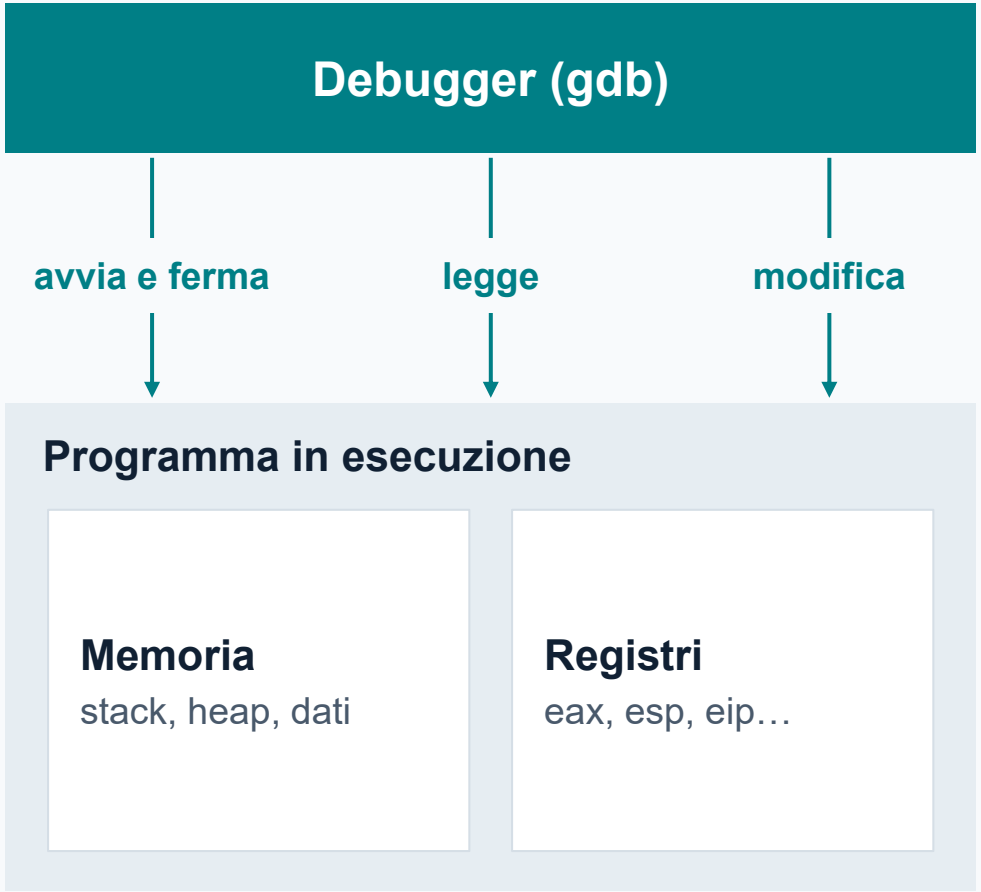
Se fare debugging è togliere i bug, programmare è metterli

Debugging

- Il debugging è il processo con cui si trovano e correggono i bug (difetti o problemi che impediscono il corretto funzionamento) in programmi, software o sistemi.
- Il debugging può basarsi su tecniche e metodologie diverse, tra cui:
  - debugging interattivo
  - analisi del flusso di controllo
  - test di unità e di integrazione
  - analisi dei log e dell'output
  - dump della memoria
  - profiling
- Molti linguaggi e ambienti di sviluppo offrono programmi di supporto al debugging, detti debugger

Debugging

- Un debugger è uno strumento software che permette di:
  - eseguire il programma in condizioni controllate
  - seguire le operazioni del programma mentre è in esecuzione
  - monitorare i cambiamenti delle risorse
  - visualizzare il contenuto della memoria
  - modificare il contenuto di memoria e registri



Debugging

- I compilatori possono emettere dati aggiuntivi (opzionali) che il debugger usa per dare informazioni più ricche
- Con gcc si usa il parametro `-g`
- Le informazioni di debug sono salvate in sezioni specifiche del file ELF:
  - `.debug`: informazioni per il debugging simbolico
  - `.line`: informazioni sui numeri di riga

```
$ gcc -g -o hello hello.c
$ readelf -S hello | grep debug
  [27] .debug_aranges  …
  [28] .debug_info     …
  [29] .debug_abbrev   …
  [30] .debug_line     …
  [31] .debug_str      …
```

Con gcc attuale (formato DWARF) le sezioni si chiamano .debug_*: .debug_line contiene i numeri di riga

Esempio

```
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
```

Indice

Introduzione al corso

Richiami di base

01 Dal codice al programma

02 Executable and Linkable Format (ELF)

03 Architettura x86

04 Gestione della memoria e calling convention

05 Debugging

Software e strumenti

06 Raccogliere informazioni dai binari

07 GDB: The GNU Project Debugger

08 GEF: GDB Enhanced Features

09 Pwntools

10 Altri strumenti: Radare2 e Ghidra

Raccogliere informazioni dai binari

Software e strumenti

L'arte di far parlare un file che non vuole collaborare

Raccogliere informazioni dai file binari

Dato un file binario possiamo…

- verificare se il file è eseguibile
- scoprire per quale architettura è stato compilato
- raccogliere simboli e stringhe usati dal programma
- verificare se c'è un processo in esecuzione associato al binario
- leggere lo SHA del file e verificare se è associato a malware noto
- identificare i nomi delle funzioni e le librerie usate

Raccogliere informazioni dai file binari

- Per estrarre informazioni da un file ELF sono disponibili diversi strumenti, tra cui:
  - `strings`
  - `objdump`
  - `readelf`

`strings`
sequenze di testo leggibile nel file

`objdump`
header, sezioni e disassembly

`readelf`
header ELF, segmenti e tabella dei simboli

strings

- È un semplice strumento da terminale che raccoglie tutte le stringhe presenti in un file binario
- Per ogni file, `strings` stampa le sequenze di caratteri stampabili lunghe almeno 4 caratteri e seguite da un carattere non stampabile
- Le stringhe raccolte possono rivelare segreti e dati usati dal programma

Byte del file (esadecimale)

7F	45	4C	46	02	01	01	00
.	E	L	F	.	.	.	.

“ELF”: solo 3 caratteri, scartata

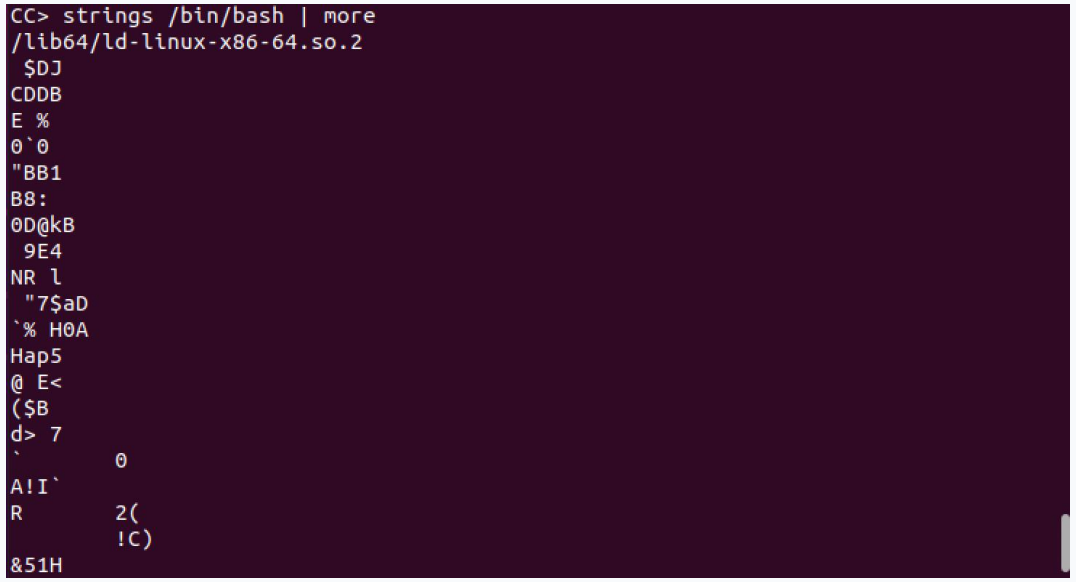
00	70	61	73	73	77	6F	72	64	00
.	p	a	s	s	w	o	r	d	.

“password”: 8 caratteri, stampata

```
$ strings ./prog
password
…
```

strings: un esempio

- Su Linux possiamo usare `strings` per raccogliere dati da `/bin/bash`, la shell Bash:



strings: alcune opzioni

- Le opzioni principali sono:
  - `-d` (o `--data`): raccoglie le stringhe solo dalle sezioni dati
  - `-n <num>` (o `--bytes=<num>`): stampa solo le stringhe di almeno <num> caratteri (default 4)
  - `-h` (o `--help`): stampa l'help del programma

```
# solo le stringhe delle sezioni dati
CC> strings -d /bin/bash

# solo stringhe di almeno 8 caratteri
CC> strings -n 8 /bin/bash

# help del programma
CC> strings --help
```

objdump

- …mostra informazioni su uno o più file oggetto, ad esempio:
  - le informazioni dell'header generale di ciascun file oggetto:

```
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

- le informazioni dagli header delle sezioni del file oggetto (opzione `-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  000000000001c9a8  000000000001c9a8  0001c9a8  2**1
```

objdump

- il contenuto di sezioni specifiche:

```
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

- Lo strumento si può usare anche per disassemblare i binari:
- L'elenco completo delle funzionalità si ottiene con l'opzione `-H` (o `--help`)

```
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>
```

readelf

- Le informazioni principali mostrate da `readelf` sono:
  - l'header del file (opzione `-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

- le informazioni degli header dei segmenti (opzione `-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

- le voci della tabella dei simboli del 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)
```

Indice

Introduzione al corso

Richiami di base

01 Dal codice al programma

02 Executable and Linkable Format (ELF)

03 Architettura x86

04 Gestione della memoria e calling convention

05 Debugging

Software e strumenti

06 Raccogliere informazioni dai binari

07 GDB: The GNU Project Debugger

08 GEF: GDB Enhanced Features

09 Pwntools

10 Altri strumenti: Radare2 e Ghidra

GDB: The GNU Project Debugger

Software e strumenti

Il vecchio saggio della riga di comando

GDB: The GNU Project Debugger

- GDB è uno strumento di debugging che permette di…
  - avviare il programma specificando tutto ciò che ne influenza il comportamento
  - fermare il programma al verificarsi di condizioni specifiche
  - esaminare cosa è successo quando il programma si è fermato
  - modificare memoria o registri mentre il programma è in esecuzione

1	**Avvia** `run`
2	**Ferma** `break main`
3	**Esamina** `print x, info registers`
4	**Modifica** `set var x = 0`

GDB: The GNU Project Debugger

- GDB supporta molti linguaggi:
  - Assembly (x86, ARM, MIPS…)
  - C
  - C++
  - Rust
  - …

Assembly	**C**
C++	**Rust**
Go	**Fortran**
Ada	**D**

In evidenza: i linguaggi che useremo nel corso

GDB: The GNU Project Debugger

- GDB è uno strumento da terminale e si avvia con il comando `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

- Si può avviare gdb passando il programma da analizzare

gdb <program>

- si può passare come secondo argomento l'ID di un processo in esecuzione per analizzarlo:

gdb <program> <pid>

- Oppure, in modo equivalente:

gdb -p <pid>

Nuovo processo

```
$ gdb ./prog
```

→ **processo** avviato con run

Processo già in esecuzione (attach)

```
$ gdb -p 1234
```

→ **PID 1234** già attivo

> Per l'attach serve il permesso di ptrace: su Ubuntu usa `sudo` oppure imposta `ptrace_scope=0`

GDB: avvio

All’avvio gdb esegue questi passi principali:

- configura l’interprete dei comandi secondo la riga di comando
  - gdb può essere eseguito in diverse modalità:
    - batch: esegue una lista di comandi e attende la terminazione
    - quiet: non stampa i messaggi introduttivi e di copyright
- carica i file di configurazione
  - definiscono il comportamento di gdb all’avvio
  - possono essere globali di sistema o locali alla directory corrente
  - i dettagli sono nella documentazione di gdb
- carica i simboli del programma da analizzare
- attende i comandi dell’utente

GDB: comandi

- Un comando gdb è una singola riga di input che contiene:
  - il nome del comando
  - una sequenza di parametri
- Il comando `run` avvia un programma 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: comandi

- Il comando `help` mostra l'elenco dei comandi disponibili:

```
(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: comandi

- I comandi `set args` e `show args` impostano e mostrano gli argomenti del programma:

```
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: comandi

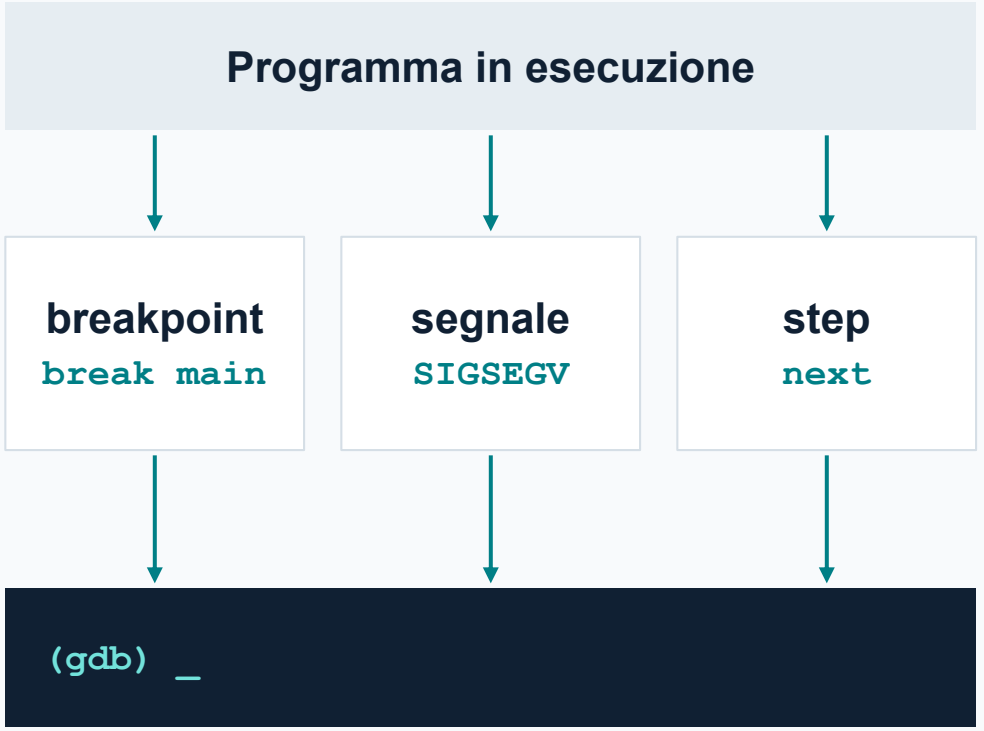
- L'ambiente è l'insieme delle variabili d'ambiente con informazioni come nome utente, home directory, tipo di terminale o percorso di ricerca dei programmi.
- In gdb le variabili d'ambiente si leggono e modificano con questi comandi:
  - `path <directory>`: aggiunge la directory al path
  - `show paths`: mostra il contenuto del path
  - `show environment [<var>]`: mostra l'ambiente (o la variabile indicata)
  - `set environment <var> <value>`: imposta il valore della variabile
  - `unset environment <var>`: elimina la variabile dall'ambiente

```
(gdb) show environment HOME
HOME = /home/studente
(gdb) set environment LANG C
(gdb) unset environment LD_PRELOAD
(gdb) path /opt/tools/bin
Executable and object file path: …
```

> Le modifiche valgono dal prossimo **`run`**

GDB: fermare e riprendere

- Il motivo principale per usare un debugger è poter fermare il programma prima che termini, per controllarne lo stato e, se ci sono problemi, capirne la causa.
- In gdb un programma può fermarsi per:
  - un breakpoint
  - un segnale
  - il completamento di uno step



Il programma resta sospeso: ne ispezioni lo stato finché non dai continue, step o next

GDB: breakpoint

- Un breakpoint ferma il programma ogni volta che si raggiunge un certo punto
  - si possono aggiungere condizioni per controllare più finemente quando fermarsi
- Un watchpoint è un breakpoint speciale che ferma il programma quando cambia il valore di un'espressione
- Un catchpoint è un altro breakpoint speciale che ferma il programma quando si verifica un certo tipo di evento

Breakpoint
si ferma quando si raggiunge un punto del codice

```
break fib if n == 2
```

Watchpoint
si ferma quando cambia il valore di un'espressione

```
watch counter
```

Catchpoint
si ferma quando si verifica un evento

```
catch syscall write
```

GDB: breakpoint

- I breakpoint si impostano con il comando `break` (abbreviato `b`):
  - `break <location>`: imposta un breakpoint nel punto indicato
  - `break`: imposta un breakpoint sull'istruzione successiva
  - `break [location] if <cond>`: imposta un breakpoint con la condizione indicata
- La posizione di un breakpoint può essere:
  - un numero di riga
  - il nome di una funzione
  - un indirizzo

```
(gdb) break 12  # riga
(gdb) break fib  # funzione
(gdb) break *0x401136  # indirizzo
(gdb) break fib if n==5  # condizione
(gdb) info breakpoints
(gdb) delete 1
```

info breakpoints elenca i breakpoint con il loro numero; delete <n> ne rimuove uno

GDB: esempio

- Consideriamo questo semplice programma C:

```
#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)
```

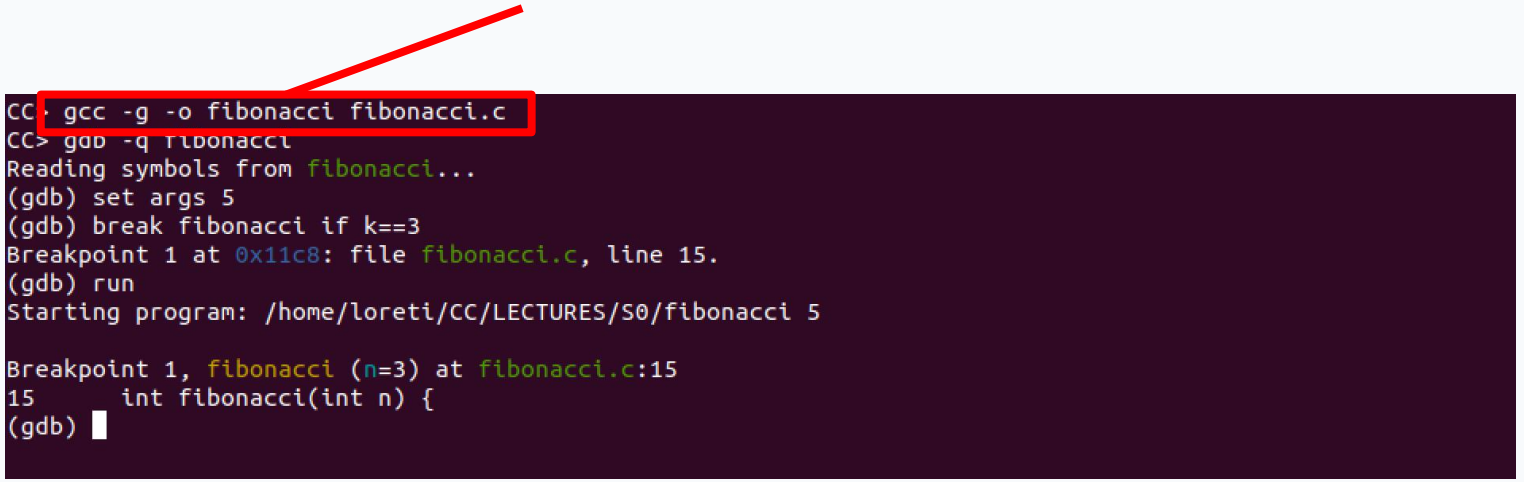
GDB: esempio

```
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)
```

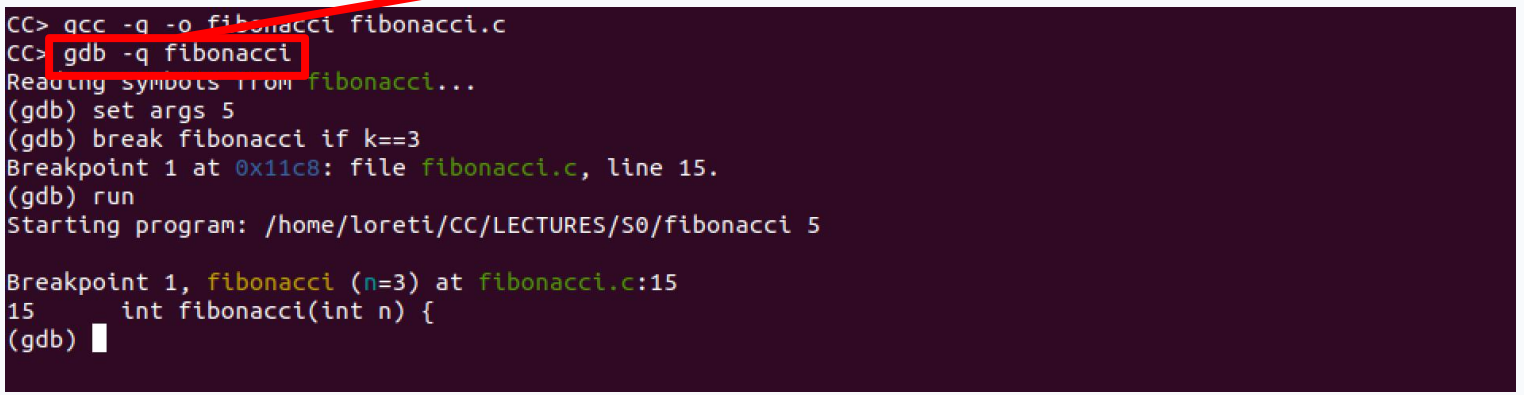
GDB: esempio

Lo compiliamo per il debugging



GDB: esempio

Avviamo gdb



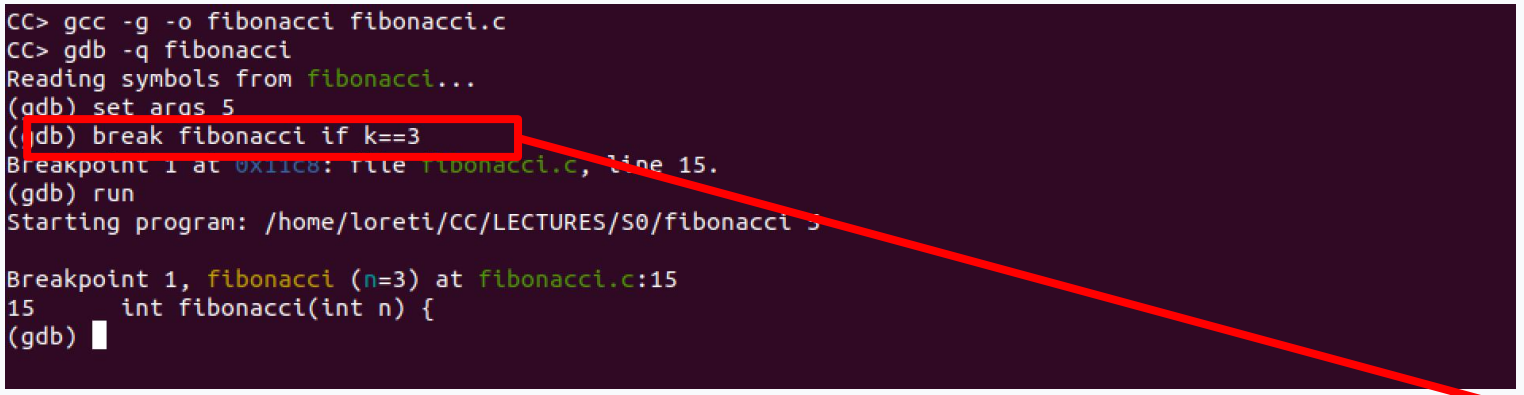
GDB: esempio

Impostiamo gli argomenti del programma

```
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: esempio



Aggiungiamo un breakpoint condizionale

GDB: esempio

```
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)
```

Eseguiamo il programma

GDB: esempio

```
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)
```

L’esecuzione si ferma quando si raggiunge il breakpoint con la condizione indicata

GDB: continue e stepping

- Quando il programma si ferma a un breakpoint, l'esecuzione si può riprendere in due modi:
  - continue: riprende l'esecuzione finché il programma non termina normalmente
  - stepping: esegue un solo "passo" del programma
    - un passo può essere un'istruzione del codice sorgente o un'istruzione assembly

```
1   int main() {
2 →   int x = f(1);
3     int y = f(2);
4     x = x + y;
5 ●   printf("%d", x);
6   }
```

→ **siamo qui**

next: riga 3

● **breakpoint**

continue: riga 5

step	esegue una riga, entra nelle funzioni
next	esegue una riga, scavalca le funzioni
stepi / nexti	una sola istruzione assembly
continue	fino al prossimo breakpoint

GDB: esempio

```
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: esempio

```
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)
```

Viene eseguito un singolo passo e gdb stampa l'istruzione successiva

GDB: esempio

```
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)
```

Possiamo fare un altro passo…

GDB: esempio

```
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)
```

…oppure continuare finché il programma termina o si raggiunge un altro breakpoint.

GDB: ispezionare lo stack

- Quando il programma si è fermato, la prima cosa da sapere è dove si è fermato e come ci è arrivato
- Il primo elemento da considerare è il contenuto dello stack
- gdb ha comandi per esaminare lo stack e leggere qualsiasi stack frame memorizzato
- In uno stack frame si trovano:
  - il punto della chiamata nel programma
  - gli argomenti della chiamata
  - le variabili locali della funzione chiamata

`#2  main ()`
variabili locali: x

`#1  fib (n=2)`
chiamata da main, riga 14

corrente
`#0  fib (n=1)`
chiamata da fib, riga 7

bt (backtrace) stampa la catena dal frame corrente fino a main

GDB: ispezionare lo stack

- Per leggere il contenuto dello stack si usano questi comandi:
  - `frame [<selection>]`
    - stampa una breve descrizione dello stack frame selezionato
  - `info frame [<selection>]`
    - stampa una descrizione dettagliata dello stack frame selezionato
- Per la (lunga) lista di opzioni si veda la documentazione di gdb

```
(gdb) bt
#0  fib (n=1) at fib.c:5
#1  0x… in fib (n=2) at fib.c:7
#2  0x… in main () at fib.c:14
(gdb) frame 1
#1  0x… in fib (n=2) at fib.c:7
(gdb) info frame
Stack level 1, frame at 0x7ffe…
  saved rip = 0x5555…
```

frame <n> seleziona un frame; info frame ne mostra indirizzi, registri salvati e argomenti

GDB: esempio

```
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)
```

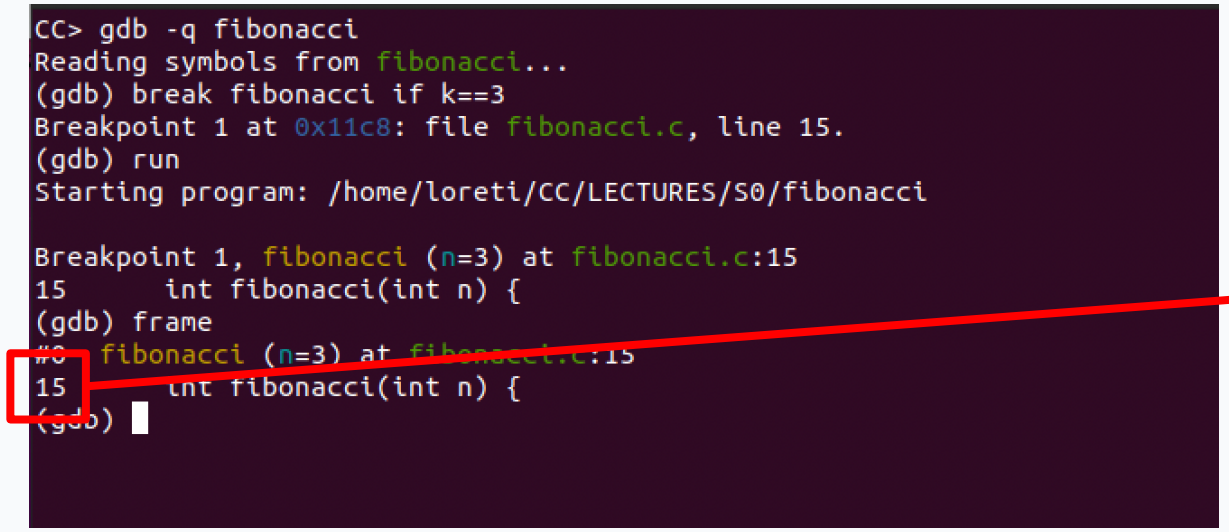
GDB: esempio

```
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)
```

Il comando `frame` mostra una breve descrizione del frame corrente

GDB: esempio


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
15 int fibonacci(int n) {
Numero del frame corrente

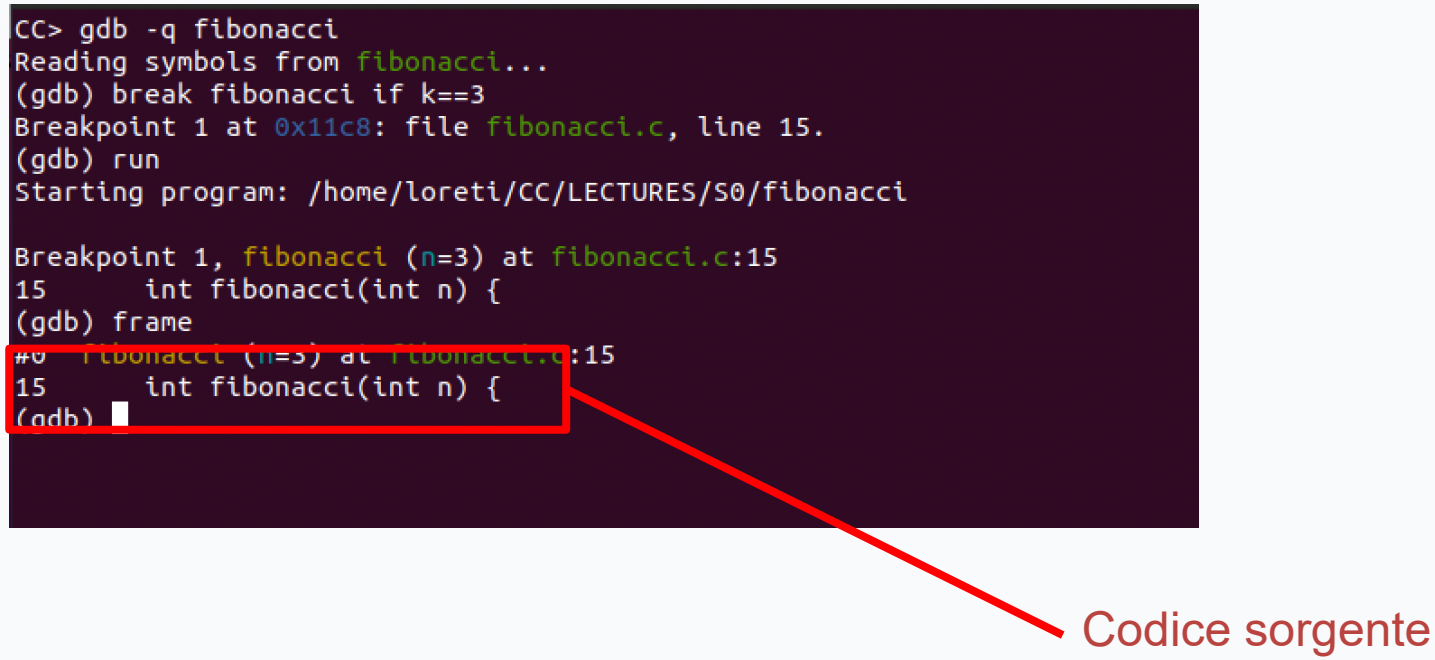
GDB: esempio

```
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)
```

Nome della funzione, argomenti e riga di codice

GDB: esempio



Codice sorgente

GDB: esempio

- Il comando `info frame` mostra informazioni dettagliate sullo stack frame corrente:

```
(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: disassemblare

- Il comando `disas` disassembla una funzione

```
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: esaminare i dati

- Per esaminare i dati del programma si usa di solito il comando `print` (abbreviato `p`) o il sinonimo `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)
```

Indice

Introduzione al corso

Richiami di base

01 Dal codice al programma

02 Executable and Linkable Format (ELF)

03 Architettura x86

04 Gestione della memoria e calling convention

05 Debugging

Software e strumenti

06 Raccogliere informazioni dai binari

07 GDB: The GNU Project Debugger

08 GEF: GDB Enhanced Features

09 Pwntools

10 Altri strumenti: Radare2 e Ghidra

GEF: GDB Enhanced Features

Software e strumenti

Lo stesso GDB, ma con gli occhiali nuovi

GEF: GDB Enhanced Features

- GEF è un insieme di comandi che estende GDB con funzioni aggiuntive per l'analisi dinamica e lo sviluppo di exploit.
- GEF si basa sulle API Python di GDB
- Funzioni principali di GEF:
  - vista hexdump integrata
  - dereferenziazione automatica di dati e registri
  - analisi dello heap
  - visualizzazione delle informazioni ELF
- La documentazione completa di GEF è disponibile qui:

https://gef.readthedocs.io/en/master/

GEF: GDB Enhanced Features

- GEF è stato progettato con questi vincoli:
  - installazione semplice e veloce
  - nessuna dipendenza esterna
    - alcune estensioni aggiungono altre funzionalità
  - compatibile con Python 2 e Python 3
  - indipendente dall'architettura
  - estendibile
  - ben documentato

Installazione in un comando

```
$ bash -c "$(curl -fsSL \
    https://gef.blah.cat/sh)"
$ gdb -q ./prog
gef➤
```

> Le versioni attuali di GEF richiedono Python 3: il supporto a Python 2 è stato rimosso

GEF: GDB Enhanced Features

- Stampa leggibile dei registri (con dereferenziazione automatica)…

```
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

- Informazioni sui chunk dello heap…

```
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

- Informazioni sulle protezioni di sicurezza…

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

GEF: GDB Enhanced Features

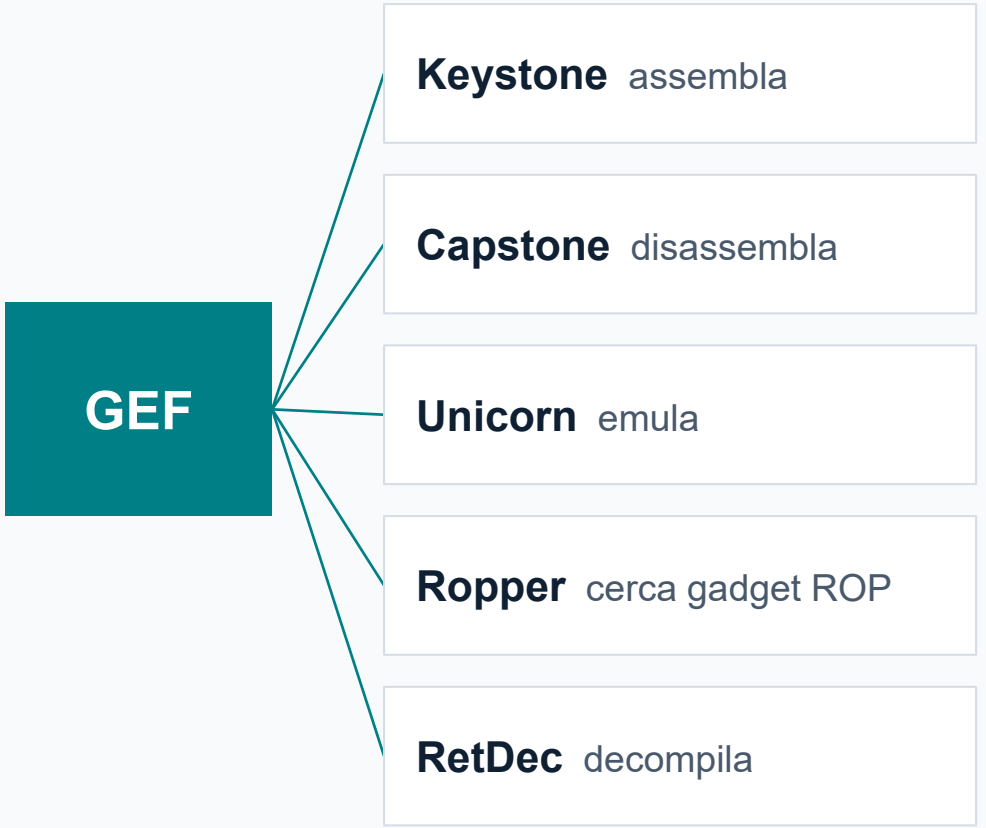
- Vista hexdump di un intervallo di memoria…

```
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

- Estensioni di GDB:
  - `keystone-assemble`: motore di assemblaggio
  - `capstone-disassemble`: motore di disassemblaggio
  - `unicorn-emulate`: motore di emulazione
  - `ropper`: generatore di gadget ROP
  - `retdec`: decompilatore

(dettagli nella documentazione di GEF)



Indice

Introduzione al corso

Richiami di base

01 Dal codice al programma

02 Executable and Linkable Format (ELF)

03 Architettura x86

04 Gestione della memoria e calling convention

05 Debugging

Software e strumenti

06 Raccogliere informazioni dai binari

07 GDB: The GNU Project Debugger

08 GEF: GDB Enhanced Features

09 Pwntools

10 Altri strumenti: Radare2 e Ghidra

Pwntools

Software e strumenti

Scrivere exploit in Python, perché a mano è roba da cavernicoli

Pwntools…

- …è un framework per CTF e una libreria per lo sviluppo di exploit:
  - scritto in Python
  - pensato per la prototipazione rapida
  - progettato per rendere la scrittura di exploit il più semplice possibile

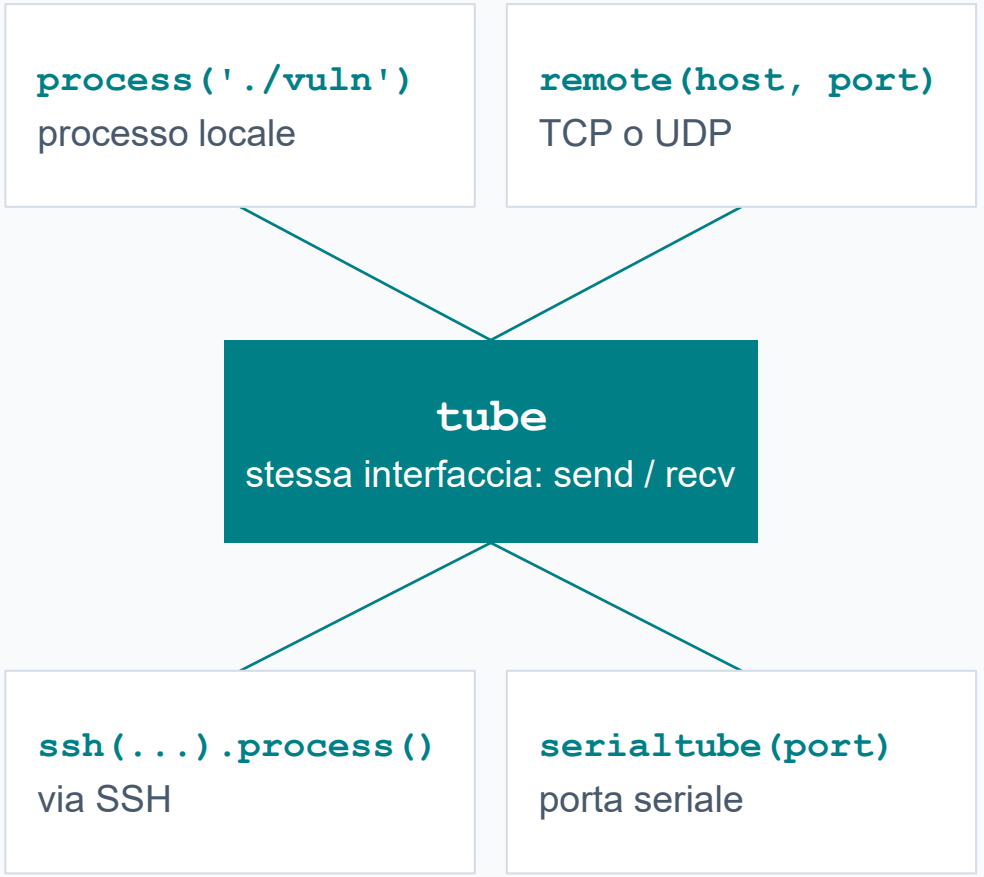
Uno scheletro di exploit

```
from pwn import *
# avvia il binario
p = process('./vuln')
# aspetta il prompt
p.recvuntil(b'> ')
# invia il payload
p.sendline(b'A'*40 + p64(0x401196))
# passa il controllo all'utente
p.interactive()
```

Indirizzo e lunghezza del payload sono solo un esempio

Pwntools: tube

- I tube sono wrapper di I/O che supportano i principali tipi di connessione:
  - processi locali (pipe)
  - connessioni remote TCP o UDP
  - processi in esecuzione su un server remoto via SSH
  - I/O su porta seriale



Pwntools: tube

- Con un tube si può:
  - inviare dati
    - `send(data)`: invia dati
    - `sendline(line)`: invia dati seguiti da un a capo
  - ricevere dati
    - `recv(n)`: riceve il numero di byte indicato
    - `recvline()`: riceve dati fino a un a capo
    - `recvuntil(delim)`: riceve dati fino al delimitatore
    - `recvregex(pattern)`: riceve dati finché non è soddisfatta la regex
    - `recvrepeat(timeout)`: continua a ricevere fino al timeout
    - `clean()`: scarta tutti i dati nel buffer
  - manipolare interi
    - `pack(int)`: invia un intero impacchettato della dimensione di una word
    - `unpack()`: riceve e spacchetta un intero della dimensione di una word

Pwntools: processi

- Un tube può interagire con un processo, creato passando:
  - il nome del binario da eseguire
  - la lista di argomenti e l'ambiente

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

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

Pwntools: esempio

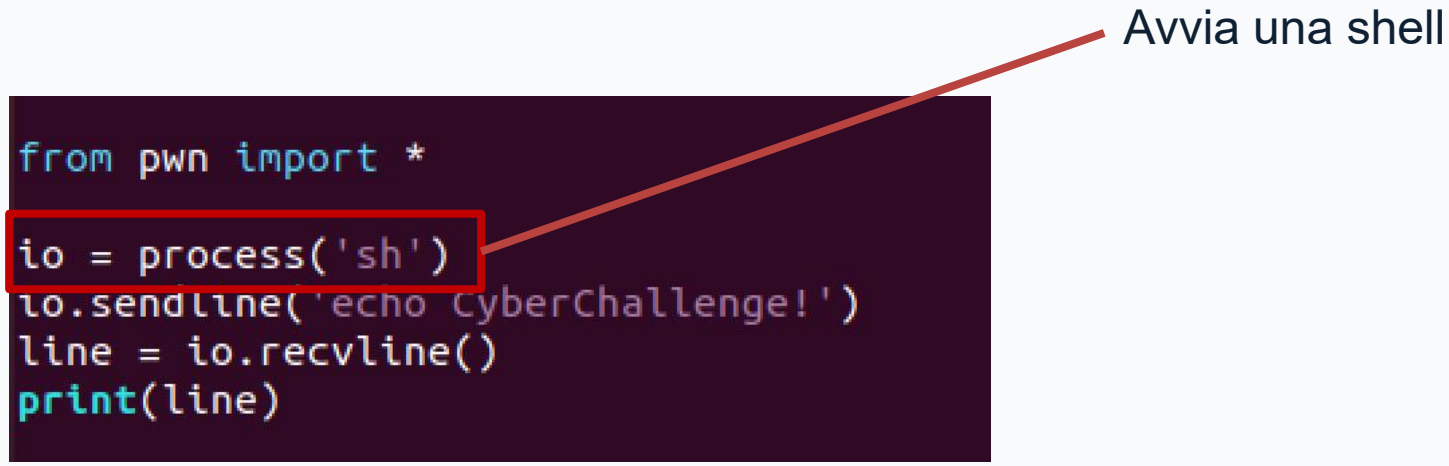
- Consideriamo questo script Python:

```
from pwn import *

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

Pwntools: esempio

- Consideriamo questo script Python:



Pwntools: esempio

- Consideriamo questo script Python:

```
from pwn import *

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

Invia un comando

Pwntools: esempio

- Consideriamo questo script Python:

```
from pwn import *

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

Riceve il risultato

Pwntools: rete

- Con pwntools si aprono facilmente connessioni di rete o si attendono connessioni in ingresso:
  - `remote` apre una connessione verso un server remoto
  - `listen` attende una connessione in ingresso
- si possono usare sia TCP sia UDP

```
# server
l = listen(1337)
c = l.wait_for_connection()
```




```
# client
r = remote('localhost', 1337)
r.sendline(b'ciao')
```

Dopo la connessione, c e r sono tube: si usano con send e recv

Pwntools: esempio

- Consideriamo questo script Python:

```
from pwn import *

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

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

Pwntools: esempio

- Consideriamo questo script Python:

```
from pwn import *

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

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

Apre una connessione

Pwntools: esempio

- Consideriamo questo script Python:

```
from pwn import *

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

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

Invia una richiesta

Pwntools: esempio

- Consideriamo questo script Python:

```
from pwn import *

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

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

Riceve la risposta

Pwntools: SSH

- Pwntools supporta le sessioni SSH:
- Data una sessione, si possono eseguire processi come visto nelle slide precedenti:

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

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

Pwntools: altre funzionalità

- Un modulo di utilità con funzioni per:
  - impacchettare e spacchettare interi
  - I/O su file
  - hashing e codifica
  - generazione di pattern
  - manipolazione di file ELF
  - assemblare e disassemblare

```
>>> p32(0xdeadbeef)
b'\xef\xbe\xad\xde'
>>> cyclic(16)
b'aaaabaaacaaadaaa'
>>> cyclic_find(b'daaa')
12
>>> enhex(b'AB')
'4142'
>>> asm('nop')
b'\x90'
```

cyclic genera un pattern senza ripetizioni: cyclic_find trova l'offset del pezzo che ha sovrascritto un registro

Indice

Introduzione al corso

Richiami di base

01 Dal codice al programma

02 Executable and Linkable Format (ELF)

03 Architettura x86

04 Gestione della memoria e calling convention

05 Debugging

Software e strumenti

06 Raccogliere informazioni dai binari

07 GDB: The GNU Project Debugger

08 GEF: GDB Enhanced Features

09 Pwntools

10 Altri strumenti: Radare2 e Ghidra

Altri strumenti: Radare2 e Ghidra

Software e strumenti

Uno nato dalla community, l'altro regalato dalla NSA

Radare2

- Radare2 è una toolchain che semplifica attività come:
  - reverse engineering del software
  - exploiting
  - debugging
- Radare2 è open source ed è disponibile qui:

https://github.com/radareorg/radare2

Una sessione tipica

```
$ r2 -A ./prog
[0x00401040]> afl   # funzioni
[0x00401040]> s main   # vai a main
[0x00401136]> pdf   # disassembla
[0x00401136]> V   # vista visuale
```

> -A analizza il binario all'avvio. I comandi sono sigle: a = analizza, s = seek, p = print; con ? dopo un comando (es. `pd?`) ottieni l'aiuto

Radare2

- Il framework comprende diversi strumenti da riga di comando, tra cui:
  - `radare2`: un potente editor esadecimale e debugger
  - `rabin2`: estrae informazioni dai binari eseguibili
  - `rasm2`: assembler e disassembler da riga di comando
  - `rahash2`: uno strumento di hashing a blocchi

```
r2 ./prog
```
apre il binario in radare2

```
rabin2 -I ./prog
```
architettura, protezioni, formato

```
rasm2 -a x86 -b 64 nop
```
stampa i byte: 90

```
rahash2 -a sha256 ./prog
```
calcola l'hash del file

Ghidra

- Ghidra è uno strumento di Software Reverse Engineering (SRE) sviluppato dalla National Security Agency (NSA)
- Ghidra è multipiattaforma e include:
  - un disassembler e decompiler interattivo
  - un insieme di strumenti per l'analisi del codice
- Lo strumento è open source ed è disponibile qui:
  - https://ghidra-sre.org
- Ghidra sarà approfondito nei prossimi moduli

Binario

```
8b 45 fc 83 c0 01 89 45 fc
```

Disassembler

```
mov eax, [rbp-4]
add eax, 1
mov [rbp-4], eax
```

Decompiler

```
x = x + 1;
```

Ghidra mostra disassembly e pseudo-C affiancati