Materiale

- VIM
- ELF
- Linking & Loading - Real Example

Process Format and Phases

Grep

Permette di ricercare stringhe all'interno di un file `grep [options] pattern [FILE]`

GREP Cheat Sheet

```
-c print a count of the number orf lines that match
-i ignore case
-v print out the lines that don't match the pattern
-n print out the line number before printing the matching line
-w searches for full words only, ignoring your string if it’s a part of another
word.
-r enables recursive search in the current directory

grep -vi hello *.txt
searches all txt files in the current dir for lines that do not contain any form
of the word hello eg Hello, HELLO or HeLlo

grep query file1 file2 file3 // multiple files
```

grep usa espressioni regolari

```
? The preceding item is optional and matched at most once.
* The preceding item will be matched zero or more times.
+ The preceding item will be matched one or more times.
{n} The preceding item is matched exactly n times.
{n,} The preceding item is matched n or more times.
{,m} The preceding item is matched at most m times.
{n,m} The preceding item is matched at least n times, but not more than m times.
^ start of the line
$ end of the line

grep ^..[l-z]$ hello.txt

any line in hello.txt that contains a three characters sequenze that ends with a
lowercase letter from l to z. es. [abl] [xyz]
```

VIM/VI

(è stato il primo editor di linux)

`vi / vim filename`

VIM Cheat Sheet

```
command mode, input mode, visual mode,
hjkl move the courser
^ move to begin of line
$ move to end
w start of word, b end of prev word
G end of file

cut and paste → 5dd delete 5 lines move to position and p to paste
copy and paste → yank 5yy

:w save file
:wq save and exit
:ZZ
:q! force exit
:e filename to edit another file
:x write file, then quit.
u - undo last edit.
ctrl-r redo.
/ search forward in the file.
? search backward in the file.
```

```c
#include <stdio.h>

int main(){
    char buf[80];
    int cookie;
    printf("buf: %08x cookie: %08x\n",&buf,&cookie);
    gets(buf);
    if(cookie == 0x41424344)
        printf("You Win!\n");
}
```

gcc compile without buffer overflow checks

```
gcc -m32 -z execstack -fno-stack-protector prog.c -o prog.out

install GEF for GDB
https://github.com/hugsy/gef
bash -c "$(curl -fsSL https://gef.blah.cat/sh)"
```

```
disas main // main is the function is guess
shell python -c 'print(0x80)'
run < <(python -c 'print("A"*84+"\x0f\x12\x40\x00")')
```

vim ha 9 buffer per fare cut/paste

Esecuzione di un programma

Il file compilato contiene una serie di informazioni che servono all'OS per far eseguire il programma.

Il codice sorgente `.c` viene dato in pasto al compilatore, il compilatore solitamente generava un codice oggetto `.o` che viene dato al linker. Adesso `gcc` compilatore fa entrambe `compile + linking`. L'unico formato eseguibile è quello binario.

Il compilatore fa due passate, nella prima passata, trasforma il programma in assembler (assemblatore è contenuto dentro il compilatore), assembler trasforma in un programma oggetto, l'oggetto entra dentro il linker, il linker recupera le librerie per poi generare il formato eseguibile.

Una volta che il programma è in formato eseguibile esiste un programma dell'OS che si chiama **Loader** che carica il programma.

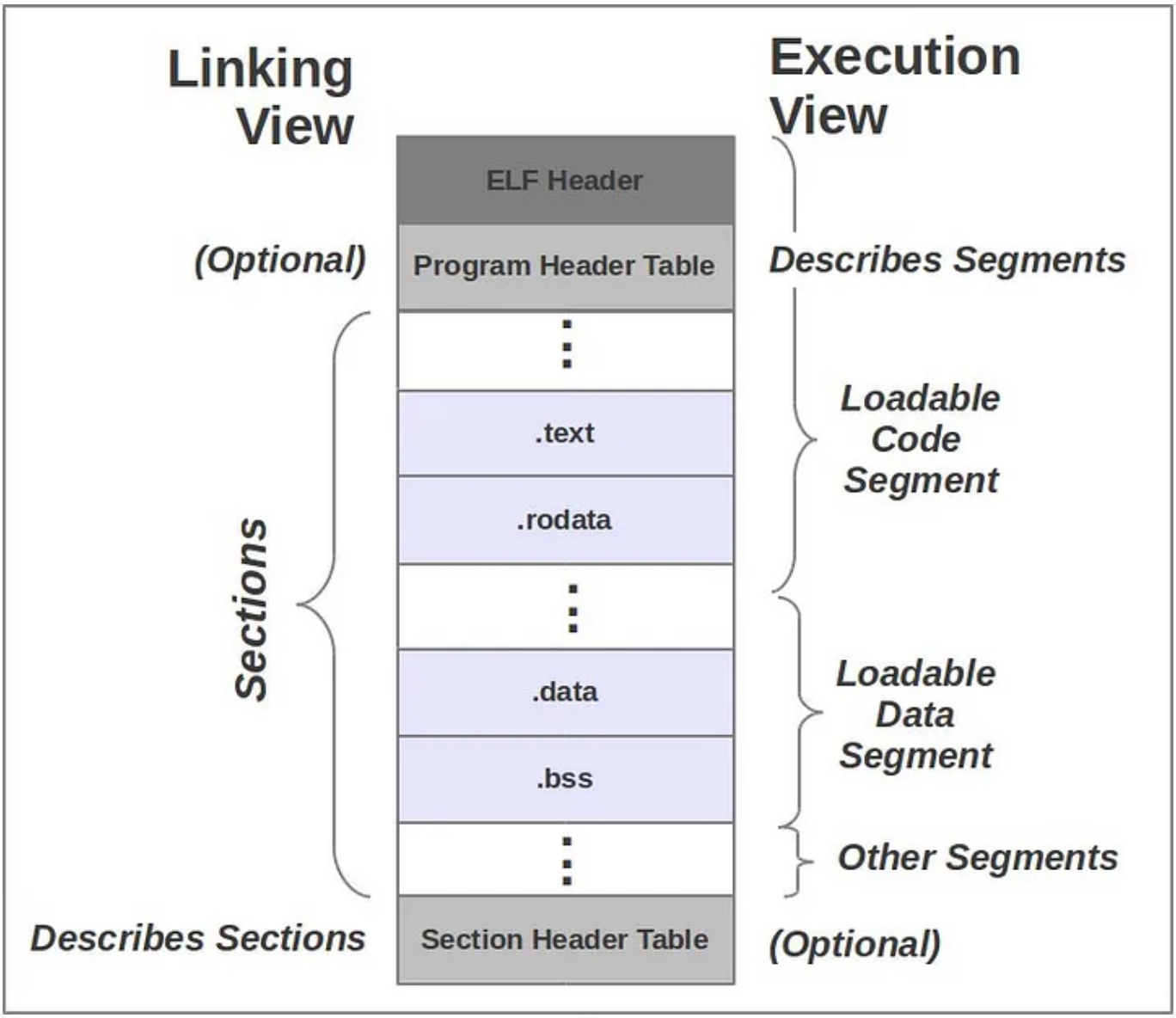
Il **loader di UNIX** è `ld`

Executable and Linkable Format - ELF

Le informazioni di un file eseguibile/oggetto creato dal compilatore è formattato in un formato standard ben definito, nel caso di UNIX è il formato **Executable and Linkable Format ELF**. Nel mondo Windows si chiama **Portable Executable PE**. (ELF è usato anche da PlayStation)

Componenti di ELF

I componenti di un file ELF sono tre: Header, Section, Segments.




```
Program header table
.text // codice del programma tradotto in codice binario
.rodata // initialized read only data
...
.data // initialized data
.bss // uninitialized data
.plt // PLT (Procedure Linkage Table) IAT equivalent
.symtab // global symbol table
...
Section header table
```

ELF Header

`readelf -h filename` # mostra ELF Header di un file ELF

```
// ELF Header
magic_number: un numero che rappresenta il tipo di file
```

```
e_entry: entry point dell'applicazione l'indirizzo di memoria della prima
istruzione eseguibile del programma;
e_phoff: file offset of the program header table;
e_shoff: offset of the section header table;
e_flag: processor-specific flags associated with the file;
e_ehsize: ELF header size;
e_phentsize: program  header entry size in program header table;
e_phnum: number of program header;
e_shentsize: section header entry size in section header table;
e_shnum: number of section headers;
e_shstrndx: index in section header table denoting section dedicated to hold
section names.
```

```
┌──(kali㉿kali)-[~/lab-sp/20231012]
└─$ readelf -h prog.bin
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x1080
  Start of program headers:          52 (bytes into file)
  Start of section headers:          13816 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         11
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

Sections

Le sezioni hanno tutte le informazioni che servono per costruire l'eseguibile, viene usato nella fase successiva del linking. Il numero di sezioni dipende dalla grandezza del programma, dalle librerie che il programma usa, ecc. Quindi, non sono presenti tutte le sezioni del programma nel file ELF. Le sezioni presenti nel maggior parte dei casi sono `.text`, `.rodata` e `.data`.

Section Header Table è una struttura dati che contiene elenco delle sezioni e loro indirizzi e altre informazioni che servono per costruire segmenti.

`readelf -S <filename>` # per vedere la section table del file

```
// Section Header
sh_name : name of the section
sh_type : section type categoria della sezione
sh_flag : 1-bit flags descrive attributi
sh_addr : indirizzo del primo byte della sezione se la sezione è nella memoria del
```

```
processo.
sh_offset : offset dal primo byte del file al primo byte della sezione
sh_size : section size in bytes
sh_link : section header table index link
sh_info : informazioni extra sulla sezione
```

```
┌──(kali㉿kali)-[~/lab-sp/20231012]
└─$ readelf -S prog.bin
There are 30 section headers, starting at offset 0×35f8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        00000194 000194 000013 00   A  0   0  1
  [ 2] .note.gnu.bu[ ... ] NOTE          000001a8 0001a8 000024 00   A  0   0  4
  [ 3] .note.ABI-tag     NOTE            000001cc 0001cc 000020 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        000001ec 0001ec 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          0000020c 00020c 0000a0 10   A  6   1  4
  [ 6] .dynstr           STRTAB          000002ac 0002ac 0000b2 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          0000035e 00035e 000014 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         00000374 000374 000040 00   A  6   1  4
  [ 9] .rel.dyn          REL             000003b4 0003b4 000040 08   A  5   0  4
  [10] .rel.plt          REL             000003f4 0003f4 000020 08  AI  5  23  4
  [11] .init             PROGBITS        00001000 001000 000020 00  AX  0   0  4
  [12] .plt              PROGBITS        00001020 001020 000050 04  AX  0   0 16
  [13] .plt.got          PROGBITS        00001070 001070 000008 08  AX  0   0  8
  [14] .text             PROGBITS        00001080 001080 00019a 00  AX  0   0 16
  [15] .fini             PROGBITS        0000121c 00121c 000014 00  AX  0   0  4
  [16] .rodata           PROGBITS        00002000 002000 000029 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        0000202c 00202c 00002c 00   A  0   0  4
  [18] .eh_frame         PROGBITS        00002058 002058 0000b4 00   A  0   0  4
  [19] .init_array       INIT_ARRAY      00003ee8 002ee8 000004 04  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      00003eec 002eec 000004 04  WA  0   0  4
  [21] .dynamic          DYNAMIC         00003ef0 002ef0 0000f0 08  WA  6   0  4
  [22] .got              PROGBITS        00003fe0 002fe0 000014 04  WA  0   0  4
  [23] .got.plt          PROGBITS        00003ff4 002ff4 00001c 04  WA  0   0  4
  [24] .data             PROGBITS        00004010 003010 000008 00  WA  0   0  4
  [25] .bss              NOBITS          00004018 003018 000004 00  WA  0   0  1
  [26] .comment          PROGBITS        00000000 003018 00001e 01  MS  0   0  1
  [27] .symtab           SYMTAB          00000000 003038 000290 10     28  18  4
  [28] .strtab           STRTAB          00000000 0032c8 00022b 00      0   0  1
  [29] .shstrtab         STRTAB          00000000 0034f3 000105 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)
```

Segments

Sezioni servono per costruire l'eseguibile, **i segmenti compongono l'eseguibile**. Vengono chiamati **Program Header**, sono composti dai pezzi di codice e dei dati, preparati per essere caricati nella memoria.

```
// Program Header
p_type : kind of segment
p_offset : offset from the beginning of the file to the first byte of the segment
```

```
p_vaddr : virtual address of first byte of the segment in memory
p_paddr : physical address
p_filesz : the number of bytes in the file image of the segment
p_memsz : the number of bytes in the memory image of the segment
p_flags : flags relevant to the segment

// Type of Segments
pt_null : seg informativi
pt_load : sono quelli che vengono caricati in memoria dal loader
pt_interp : specifica il punto e la grandezza di un path name da invocare come
interprete
```

`readelf -l filename` # per vedere segmenti.

```
┌──(kali㉿kali)-[~/lab-sp/20231012]
└─$ readelf -l prog.bin

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1080
There are 11 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00160 0x00160 R   0x4
  INTERP         0x000194 0x00000194 0x00000194 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x00414 0x00414 R   0x1000
  LOAD           0x001000 0x00001000 0x00001000 0x00230 0x00230 R E 0x1000
  LOAD           0x002000 0x00002000 0x00002000 0x0010c 0x0010c R   0x1000
  LOAD           0x002ee8 0x00003ee8 0x00003ee8 0x00130 0x00134 RW  0x1000
  DYNAMIC        0x002ef0 0x00003ef0 0x00003ef0 0x000f0 0x000f0 RW  0x4
  NOTE           0x0001a8 0x000001a8 0x000001a8 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x00202c 0x0000202c 0x0000202c 0x0002c 0x0002c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x002ee8 0x00003ee8 0x00003ee8 0x00118 0x00118 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr
   03     .init .plt .plt.got .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .got.plt .data .bss
   06     .dynamic
   07     .note.gnu.build-id .note.ABI-tag
   08     .eh_frame_hdr
   09
   10     .init_array .fini_array .dynamic .got
```

`objdump -S` per leggere ELF header

`objdump -d` **disassembly** mostra il codice assembler del file ELF.

`objcopy` to copy ELF sections.

Il `type` del Header dice il tipo di file, può `REL` oppure `DYN`.

- `gcc -c` crea un file (file oggetto) di tipo **REL - Relocatable File** (rilocabile) (un file che si può spostare). Nel formato REL il compilatore come prima cosa fa, traduce il programma assumendo che il programma sia l'unico programma all'interno della memoria del calcolatore. Come se partisse dall'indirizzo zero. Assegna al programma degli indirizzi partendo da zero.

  Nella fase generazione del codice oggetto vengono aggiunti le funzioni di libreria. A questo il programma non parte più da zero ma da un altro indirizzo. Perché il compilatore aggiunge le funzioni che servono per il programma il alto (es. la funzione printf di C).

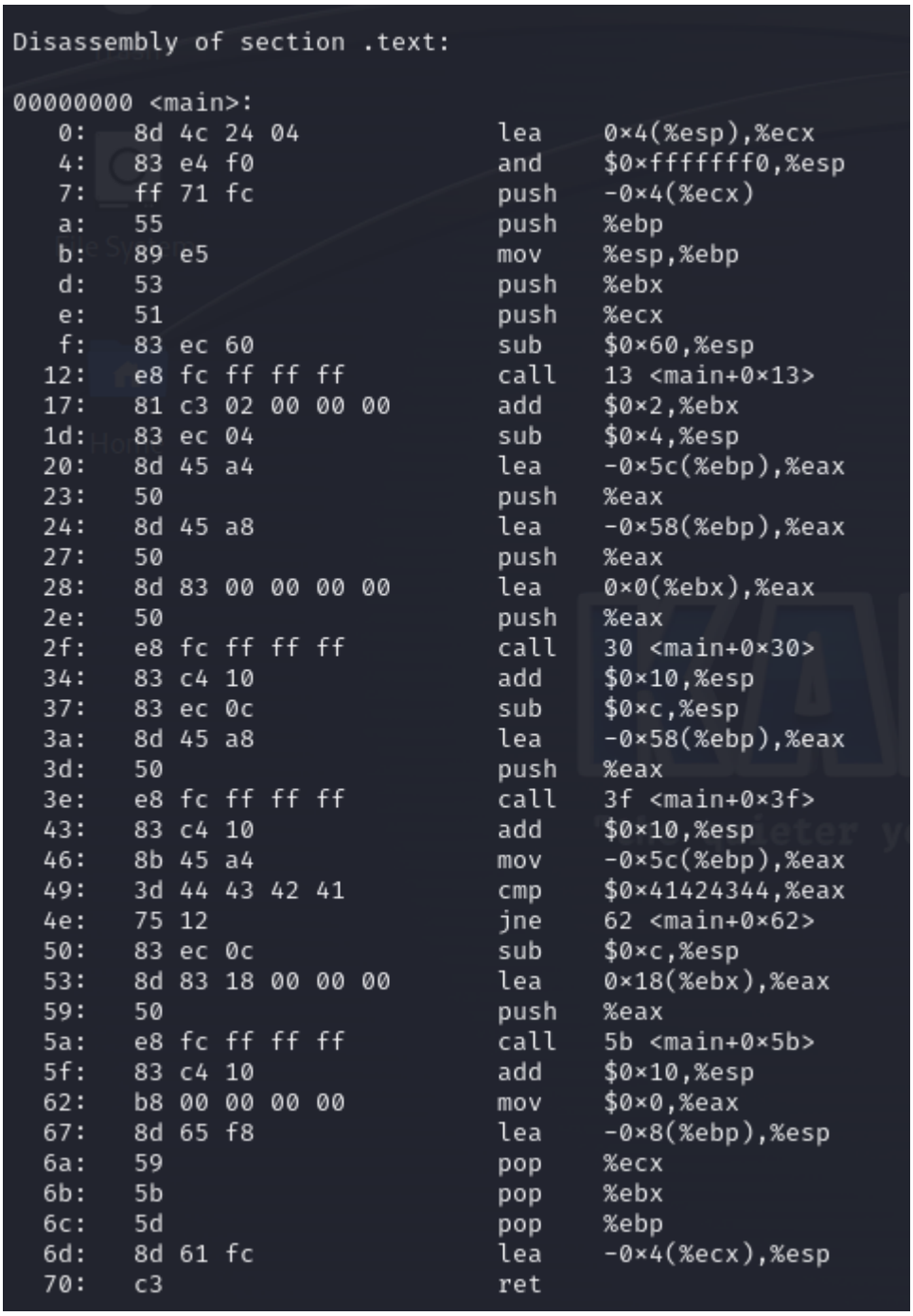
  Per questo il codice è disposto in maniera rilocabile, per essere spostato quando le funzioni saranno aggiunte.

  Vantaggio: gli si può modificare gli indirizzi.

- `gcc -o` crea un file di tipo eseguibile (ma linkato dinamicamente, con le librerie shared) **DYN - Position-Independent Executable file**.

```
┌──(kali㉿kali)-[~/lab-sp/20231012]
└─$ diff obin.txt oobj.txt
8c8
<   Type:                              DYN (Position-Independent Executable file)
---
>   Type:                              REL (Relocatable file)
11,13c11,13
<   Entry point address:               0×1080
<   Start of program headers:          52 (bytes into file)
<   Start of section headers:          13816 (bytes into file)
---
>   Entry point address:               0×0
>   Start of program headers:          0 (bytes into file)
>   Start of section headers:          796 (bytes into file)
16,17c16,17
<   Size of program headers:           32 (bytes)
<   Number of program headers:         11
---
>   Size of program headers:           0 (bytes)
>   Number of program headers:         0
19,20c19,20
<   Number of section headers:         30
<   Section header string table index: 29
---
>   Number of section headers:         15
>   Section header string table index: 14
```

Object file disassembly `objdump -d filename`:



```
Disassembly of section .text:

00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                push   -0x4(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   53                      push   %ebx
   e:   51                      push   %ecx
   f:   83 ec 60                sub    $0x60,%esp
  12:   e8 fc ff ff ff          call   13 <main+0x13>
  17:   81 c3 02 00 00 00       add    $0x2,%ebx
  1d:   83 ec 04                sub    $0x4,%esp
  20:   8d 45 a4                lea    -0x5c(%ebp),%eax
  23:   50                      push   %eax
  24:   8d 45 a8                lea    -0x58(%ebp),%eax
  27:   50                      push   %eax
  28:   8d 83 00 00 00 00       lea    0x0(%ebx),%eax
  2e:   50                      push   %eax
  2f:   e8 fc ff ff ff          call   30 <main+0x30>
  34:   83 c4 10                add    $0x10,%esp
  37:   83 ec 0c                sub    $0xc,%esp
  3a:   8d 45 a8                lea    -0x58(%ebp),%eax
  3d:   50                      push   %eax
  3e:   e8 fc ff ff ff          call   3f <main+0x3f>
  43:   83 c4 10                add    $0x10,%esp
  46:   8b 45 a4                mov    -0x5c(%ebp),%eax
  49:   3d 44 43 42 41          cmp    $0x41424344,%eax
  4e:   75 12                   jne    62 <main+0x62>
  50:   83 ec 0c                sub    $0xc,%esp
  53:   8d 83 18 00 00 00       lea    0x18(%ebx),%eax
  59:   50                      push   %eax
  5a:   e8 fc ff ff ff          call   5b <main+0x5b>
  5f:   83 c4 10                add    $0x10,%esp
  62:   b8 00 00 00 00          mov    $0x0,%eax
  67:   8d 65 f8                lea    -0x8(%ebp),%esp
  6a:   59                      pop    %ecx
  6b:   5b                      pop    %ebx
  6c:   5d                      pop    %ebp
  6d:   8d 61 fc                lea    -0x4(%ecx),%esp
  70:   c3                      ret
```

Come nasce un processo dal shell

Prima cosa che fa la shell è visualizzare la shell → legge il comando → fa `fork()` (dopo la fork abbiamo due shell) → Il figlio esegue `exec(getcommand(cmdline))` con exec metto il codice del comando → e controlla se `fg` o `bg` (un programma che non è in interazione con utente).

Loading a Process

L'OS legge argomento dell'`exec` che è un **path** (path a un file binario/eseguibile) → va **cercare quel file** → **carica in memoria** → va **verificare che sia un ELF** → (come fa?) lo verifica dal header leggendo il **magic number** → legge header per acquisire informazioni → header gli dice indirizzi dei segmenti (dove si trova il codice, i dati, le librerie, ecc.).

magic number ogni file ha il suo magic number. Serve per riconoscere il tipo di file.

Se il file inizia con `#!` sh-bang allora shell chiama l'interprete riferimento e gli passa il file.

Differenza tra un file dinamicamente linkato / staticamente linkato

File **dinamicamente linkato** : linka a runtime le cose che sono condivise. Risparmio spazio, es. perché se uso la stessa libraria in diversi processi mi basta tenere in memoria solo copia della libreria. Ma è lento perché devo andare a cercare la parte che mi serve nella memoria centrale. inoltre, serve un programmino che intercetta le richieste.

File **staticamente linkato** : è autonomo, contiene tutto quello che serve per essere eseguito, è auto consistente.

`readelf -a nameOfExec` # per capire l'interprete.

Linux Process

un processo è costituito: da una parte di testo binario, librerie, heap, stack, eventuale zone di memoria che programma usa, kernel code.

A questo punto il processo è generato. (quello che fa exec da shell).