

# *Sistemas Operativos*

## Ejecutables - I ELF



# *Sistemas Operativos*

- ✓ Versión: Abril 2020
- ✓ Palabras Claves: Linux, Windows, PE, Ejecutable, ELF, Linking, Carga Dinámica, Proceso, Fuente



# Archivos

- ✓ Un archivo es un tipo abstracto de datos que permite la manipulación de su información a través de una interfaz conocida y bien definida
- ✓ El tipo de datos abstracto, define la forma en la que se va a agrupar la información (datos y metadatos) dentro del archivo, con el fin de que pueda ser recuperada y correctamente interpretada
- ✓ La interfaz, define el conjunto de operaciones que podrán ser llevadas a cabo sobre los archivos
- ✓ Existen distintos tipos de archivo y cada uno define el **formato** en el que los datos son representados en el mismo.



# ELF - Generalidades

- ✓ ELF : Executable and Linking Format es un formato de archivo para ejecutables, código objeto, bibliotecas compartidas y volcado de memoria
- ✓ Desarrollado inicialmente por Unix System Laboratories
- ✓ Se transformó en un standard en formato de archivo
- ✓ Es el formato binario default **Linux, Solaris 2.x, BSD**
- ✓ ELF es parte de la "System V application binary interface (ABI)", que define una interfaz del sistema operativo para operar con programas compilados ejecutables, y sus funciones, (manejo del stack, manejo del heap, llamadas al sistema)



# Comparativa de Formatos de Archivos Ejecutables

| Formato             | Sistema operativo   | Extensiones del nombre de archivo | Declaraciones explícitas de procesador | Secciones arbitrarias                          | Metadatos                                    | Firma digital   | Tabla de símbolos | 64-bit    |
|---------------------|---|-----------------------------------|--|--|--|-----------------|-------------------|-----------|
| OS/360              | Sistemas operativos de computadoras centrales OS/360 y VS/9   | ninguna                           | No                                     | No   | No   | No              | Sí                | Sí        |
| a.out               | Unix-like   | ninguna                           | No                                     | No   | No   | No              | Sí <sup>1</sup>   | Extensión |
| COFF                | Unix-like   | ninguna                           | Sí (por archivo)                       | Sí   | No   | No              | Sí                | Extensión |
| ECOFF               | Ultrix, Tru64 UNIX, IRIX  | ninguna                           | Sí (por archivo)                       | Sí   | No   | No              | Sí                | Sí        |
| XCOFF               | AIX, BeOS, Mac OS   | ninguna                           | Sí (por archivo)                       | Sí   | No   | No              | Sí <sup>2</sup>   | Sí        |
| ELF                 | Unix-like   | ninguna                           | Sí (por archivo)                       | Sí   | Sí   | Sí <sup>3</sup> | Sí <sup>4</sup>   | Sí        |
| Mach-O <sup>7</sup> | NeXTSTEP, OS X, iOS   | ninguna                           | Sí (por sección)                       | Parcial<br>(limitado, max...<br>256 secciones) | Sí   | Sí              | Sí                | Sí        |
| CMD                 | CP/M-86, MP/M-86, Concurrent CP/M-86, Personal CP/M-86, S5-DOS, Concurrent DOS, Concurrent DOS 286, FlexOS, S5-DOS/ST, S5-DOS/MT, Concurrent DOS 386, Multiuser DOS, System Manager, REAL/32, DOS Plus    | .CMD <sup>8</sup>                 | No (sólo x86)                          | Sí   | No   | No              | Extensión         | No        |
| COM (DOS)           | DOS, OS/2, Windows (excepto en ediciones de 64-bit), Concurrent CP/M-86 (sólo BDOS 3.1), Concurrent DOS, Concurrent DOS 286, FlexOS, Concurrent DOS 386, Multiuser DOS, System Manager, REAL/32, DOS Plus | .COM                              | No (sólo x86)                          | No   | Extensión<br>(Novell/Caldera<br>VERSION.EXE) | No              | No                | Extensión |
| MZ (DOS)            | DOS, OS/2, Windows (excepto en ediciones de 64-bit), Concurrent DOS 286, FlexOS, Concurrent DOS 386, Multiuser DOS, System Manager, REAL/32, DOS Plus   | .EXE                              | No (sólo x86)                          | Sí   | Extensión<br>(Novell/Caldera<br>VERSION.EXE) | No              | Extensión         | Extensión |

[https://es.wikipedia.org/wiki/Anexo:Comparaci%C3%B3n\\_de\\_formatos\\_de\\_archivos\\_ejecutables](https://es.wikipedia.org/wiki/Anexo:Comparaci%C3%B3n_de_formatos_de_archivos_ejecutables)



# Capacidades de ELF

- ✓ Soporta dynamic linking
- ✓ Soporta dynamic loading
- ✓ Facilita la creación de librerías compartidas (shared libraries)
- ✓ La representación del archivo es independiente de la plataforma



# ELF Object File

- ☑ Los object files son representaciones binarias de programas que se ejecutan directamente en el procesador.
- ☑ Los crea el ensamblador y el link editor
- ☑ Los archivos objeto participan en el linking de los programas y la ejecución
- ☑ Se excluyen los programas que requieren otras máquinas abstractas
  - ✓ Shell Scripts, Java, etc.





# Tipos de archivos objeto de ELF

- ☑ Existen tres tipos principales de Object Files:
  - ☑ **Executables:** Representa un programa para ser ejecutado
  - ☑ **Relocatable:** Representa código y datos que puede ser utilizado para linkearse con otros object files para formar un *executable* o un *shared object file*
  - ☑ **Shared Object:** Representa código y datos para ser utilizados en dos contextos:
    - ☑ El linkeditor puede procesarlos junto con otros relocatables y shared objects para crear un object file ← Compilación
    - ☑ El linkeditor dinámico los combina con un executable y otros shared objects para crear una imagen de proceso ← Ejecución
- ☑ Los 3 tipos son para diferentes propósitos. La estructura interna es diferente.





# Tipos - Ejecutables

- ☒ Es un formato para archivos aptos para ejecución
  - ✓ gcc -o program program.c
- ☐ Por defecto en gcc cuando no se indica ningún parámetro adicional al -o (output), lo que se genera es un archivo ejecutable



# *Tipos - Reubicables (relocatable)*

- ☑ Contiene código y datos en un formato adecuado para ser “linkeado” con otros objetos para **crear un ejecutable u objeto compartido**
- ☑ Es la “base” para crear ejecutables y Librerías
  - ✓ gcc -c program.c
- ☑ Generalmente archivos .o o .ko
- ☑ El parámetro -c indica al gcc que no ejecute al linkeditor. De este modo, la salida son object files que luego serán utilizados o linkeados desde otros programas



## *Tipos - Objetos compartidos (Shared Objects)*

- ☑ Mantiene código y datos en un formato adecuado para ser linkeado a dos contextos:
  - ✓ El linker puede crear un nuevo objeto a partir de procesar un objeto de este tipo con otros. (Compilación)
  - ✓ El dynamic linker lo combina con un ejecutable y otros objetos para crear una imagen de proceso. (Ejecución)
- ☑ Generalmente archivos .so
  - ✓ /usr/lib
  - ✓ Por ejemplo: libssl.so



# Tipos - Ejemplo

## •Ejecutables

### ✓ readelf -h /bin/mkdir

```
# readelf -h /bin/mkdir
Encabezado ELF:
Mágico:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Clase:                                ELF64
Datos:                                complemento a 2, little endian
Versión:                              1 (current)
OS/ABI:                               UNIX - System V
Versión ABI:                          0
Tipo:                                 EXEC (Fichero ejecutable)
Máquina:                              Advanced Micro Devices X86-64
Versión:                              0x1
Dirección del punto de entrada:      0x402903
Inicio de encabezados de programa:    64 (bytes en el fichero)
Inicio de encabezados de sección:     79016 (bytes en el fichero)
Opciones:                             0x0
Tamaño de este encabezado:            64 (bytes)
Tamaño de encabezados de programa:    56 (bytes)
Número de encabezados de programa:    9
Tamaño de encabezados de sección:     64 (bytes)
Número de encabezados de sección:     27
Índice de tabla de cadenas de sección de encabezado: 26
```



# Tipos - Ejemplo

## •Relocatables

✓ `readelf -h /lib/modules/3.16.0-7-amd64/kernel/sound/pci/hda/snd-hda-codec-realtek.ko`

```
# readelf -h /lib/modules/3.16.0-7-amd64/kernel/sound/pci/hda/snd-hda-codec-realtek.ko
```

Encabezado ELF:

```
Mágico: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Clase: ELF64
Datos: complemento a 2, little endian
Versión: 1 (current)
OS/ABI: UNIX - System V
Versión ABI: 0
Tipo: REL (Fichero reubicable)
Máquina: Advanced Micro Devices X86-64
Versión: 0x1
Dirección del punto de entrada: 0x0
Inicio de encabezados de programa: 0 (bytes en el fichero)
Inicio de encabezados de sección: 117224 (bytes en el fichero)
Opciones: 0x0
Tamaño de este encabezado: 64 (bytes)
Tamaño de encabezados de programa: 0 (bytes)
Número de encabezados de programa: 0
Tamaño de encabezados de sección: 64 (bytes)
Número de encabezados de sección: 30
Índice de tabla de cadenas de sección de encabezado: 27
```



# Tipos - Ejemplo

## •Shared Object (DYN)

✓ `readelf -h /usr/lib/x86_64-linux-gnu/sasl2/libcrammd5.so`

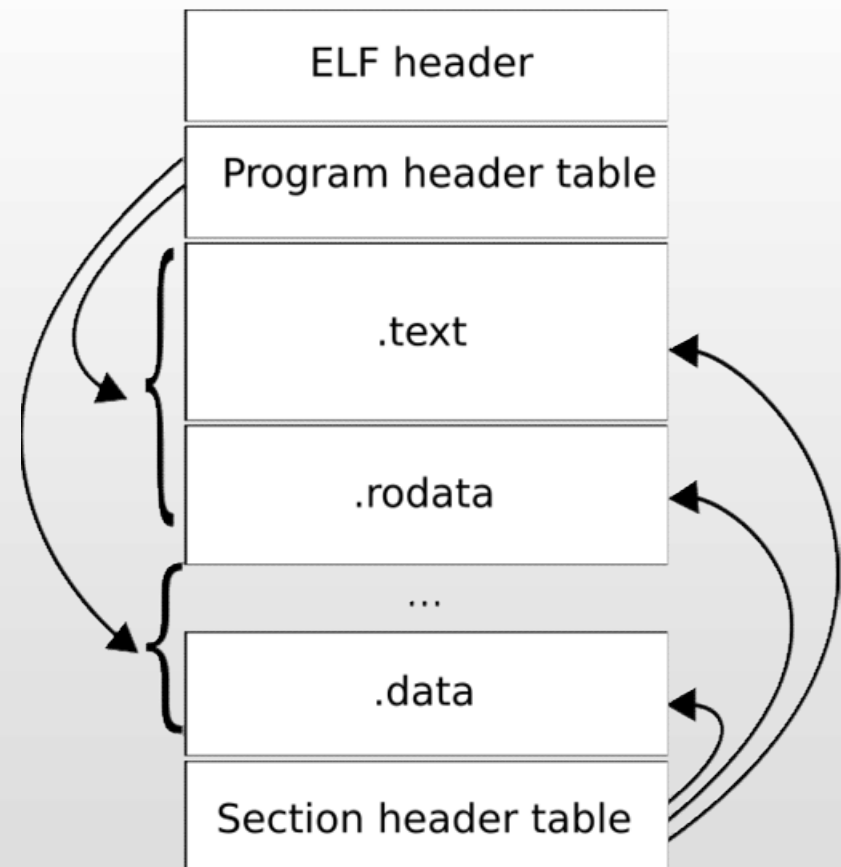
```
# readelf -h /usr/lib/x86_64-linux-gnu/sasl2/libcrammd5.so
Encabezado ELF:
  Mágico:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Clase:                                ELF64
  Datos:                                complemento a 2, little endian
  Versión:                              1 (current)
  OS/ABI:                               UNIX - System V
  Versión ABI:                          0
  Tipo:                                DYN (Fichero objeto compartido)
  Máquina:                             Advanced Micro Devices X86-64
  Versión:                              0x1
  Dirección del punto de entrada:       0x1250
  Inicio de encabezados de programa:     64 (bytes en el fichero)
  Inicio de encabezados de sección:      21056 (bytes en el fichero)
  Opciones:                             0x0
  Tamaño de este encabezado:             64 (bytes)
  Tamaño de encabezados de programa:     56 (bytes)
  Número de encabezados de programa:      7
  Tamaño de encabezados de sección:      64 (bytes)
  Número de encabezados de sección:       25
  Índice de tabla de cadenas de sección de encabezado: 24
```



# Secciones del ELF

Cada archivo ELF está formado por un Header y sus datos:

1. Header
2. Program Header Table
3. Section Header Table
4. Información
  1. ELF sections
  2. ELF segments

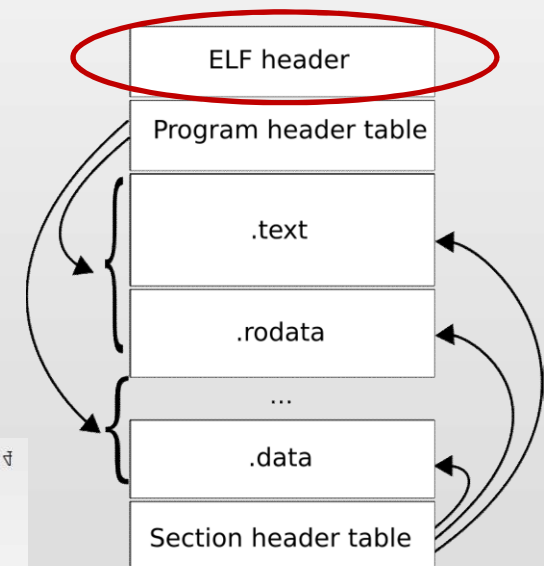




# ELF header

- ✓ Siempre reside al comienzo del archivo
- ✓ Tiene un "road map" que describe la organización del archivo
- ✓ Es único y tiene un lugar fijo dentro del formato
- ✓ Define si se utilizan direcciones de 32 o 64 bits
- ✓ Dentro de sus campos se encuentran:
  - ✓ Magic Number
  - ✓ Endianness (big o Little)
  - ✓ Plataforma para la que fue compilado
  - ✓ Arquitectura para la que fue compilado
  - ✓ Etc...

```
Machine:      Advanced Micro Devices X86-64
Type:         EXEC (Executable file)
Version ABI:  0
OS/ABI:       UNIX - System V
Version:      1 (current)
Data:         complemento a 2, little endian
```



# ELF header (cont).

- ☑ Identificación del archivo, tipo, arquitectura, punto de entrada (donde comienza a ejecutarse, dir. virtual), donde comienzan las Program y Header tables (desplazamiento dentro el archivo) y cantidad de entradas de la misma, etc.

```
#define EI_NIDENT      16

typedef struct {
    unsigned char    e_ident[EI_NIDENT];
    Elf32_Half       e_type;
    Elf32_Half       e_machine;
    Elf32_Word       e_version;
    Elf32_Addr       e_entry;
    Elf32_Off        e_phoff;
    Elf32_Off        e_shoff;
    Elf32_Word       e_flags;
    Elf32_Half       e_ehsize;
    Elf32_Half       e_phentsize;
    Elf32_Half       e_phnum;
    Elf32_Half       e_shentsize;
    Elf32_Half       e_shnum;
    Elf32_Half       e_shstrndx;
} Elf32_Ehdr;
```



# Programa ejemplo

```
/* test.c */
#include <stdio.h>
int global_data = 4;
int global_data_2;
int main(int argc, char **argv) {
    int local_data = 3;
    printf("Hello World\n");
    printf("global_data = %d\n", global_data);
    printf("global_data_2 = %d\n", global_data_2);
    printf("local_data = %d\n", local_data);
    return (0);
}
```

```
gcc -o test test.c
```

```
gcc -o test-st --static test.c
```



# ELF header (cont).

☑ Ejemplo:

✓ readelf -h test

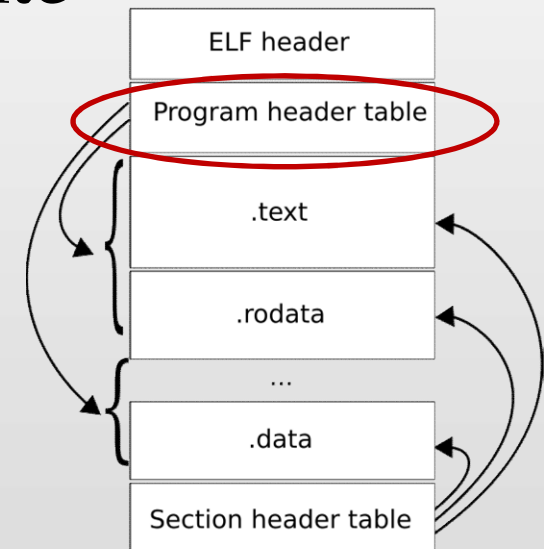
## ELF Header:

```
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: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x8048340
Start of program headers: 52 (bytes into file)
Start of section headers: 5900 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 8
Size of section headers: 40 (bytes)
Number of section headers: 36
Section header string table index: 33
```



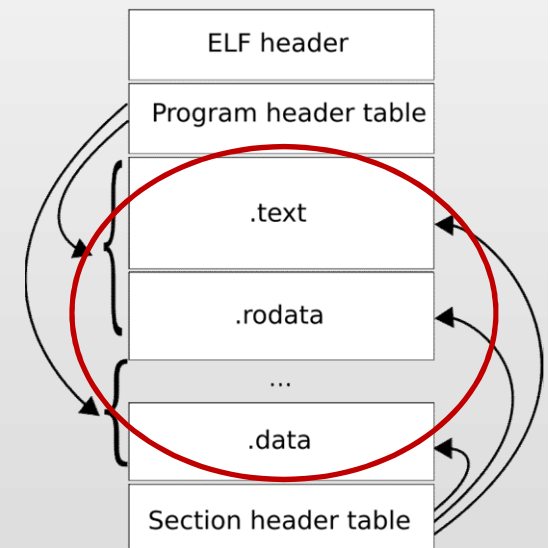
# Program header table

- ✓ Es una tabla que indica como se debe crear la imagen del proceso.
- ✓ En el File Header, se indica donde está el comienzo de esta tabla.
- ✓ Es opcional y no siempre está presente
- ✓ Los ejecutables la deben tener siempre
- ✓ Los relocatables no la necesitan



# Sections y Segments

- ✓ Ambos son divisiones Lógicas dentro de archivo que agrupan la información
  - ✓ Datos, código, etc.
- ✓ Las secciones son utilizadas durante el linking
  - ✓ Indican como construir el programa
- ✓ Los segmentos son utilizados durante la ejecución
  - ✓ Indican como ejecutar el programa



# Sections y Segments (cont.)

Linking View

|                                    |
|------------------------------------|
| ELF header                         |
| Program header table<br>(optional) |
| section 1                          |
| ...                                |
| section n                          |
| ...                                |
| ...                                |
| Section header table               |

Execution View

|                                    |
|------------------------------------|
| ELF header                         |
| Program header table               |
| Segment 1                          |
| Segment 2                          |
| ...                                |
| ...                                |
| Section header table<br>(optional) |





# Sections y Segments (cont.)

## ☑ Section header table:

- ✓ Contiene información que describe las secciones del archivo y permite localizarlas
- ✓ Cada sección tiene una entrada en la tabla, con información sobre la sección, su nombre, tamaño, la tabla de símbolos de la sección, la tabla de strings, etc.

## ☑ Los archivos que se usan durante el linking (relocatables principalmente) deben tener section header table; los object files pueden o no tenerla.



# Sections (cont.)

- ☑ Section Header Table es un array de estructuras:
  - ✓ Nombre, tipo, comienzo en el archivo, tamaño, flags (se puede leer, escribir, ejecutar, etc)
  - ✓ El campo `sh_type` categoriza el contenido y la semántica de la sección. Por ejemplo `sh_type = SHT_SYMTAB` define que el elemento es una Symbol Table

```
typedef struct {  
    Elf32_Word    sh_name;  
    Elf32_Word    sh_type;  
    Elf32_Word    sh_flags;  
    Elf32_Addr    sh_addr;  
    Elf32_Off     sh_offset;  
    Elf32_Word    sh_size;  
    Elf32_Word    sh_link;  
    Elf32_Word    sh_info;  
    Elf32_Word    sh_addralign;  
    Elf32_Word    sh_entsize;  
} Elf32_Shdr;
```



# Sections

- ✓ Son una colección de información de tipo similar.
- ✓ Cada sección representa una porción del archivo
- ✓ Contienen un nombre y un tipo
- ✓ Ejemplos de secciones especiales (principalmente utilizadas por el Sistema Operativo):
  - ✓ **.text:** Instrucciones ejecutables
  - ✓ **.data:** variables inicializadas por el usuario
  - ✓ **.bss:** Datos no inicializados
  - ✓ **.comment:** Información sobre control de versiones
  - ✓ **.strtab:** Strings que representan los nombres asociados en la Symbol Table



# Sections (cont.)

☑ Ejemplo: Viendo la Section Header Table

✓ readelf -S test

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 | 08048134 | 000134 | 000013 | 00 | A   | 0  | 0   | 1  |
| [ 2] | .note.ABI-tag  | NOTE     | 08048148 | 000148 | 000020 | 00 | A   | 0  | 0   | 4  |
| [ 3] | .hash          | HASH     | 08048168 | 000168 | 00002c | 04 | A   | 5  | 0   | 4  |
| [ 4] | .gnu.hash      | GNU_HASH | 08048194 | 000194 | 000020 | 04 | A   | 5  | 0   | 4  |
| [ 5] | .dynsym        | DYNSYM   | 080481b4 | 0001b4 | 000060 | 10 | A   | 6  | 1   | 4  |
| [ 6] | .dynstr        | STRTAB   | 08048214 | 000214 | 000051 | 00 | A   | 0  | 0   | 1  |
| [ 7] | .gnu.version   | VERSYM   | 08048266 | 000266 | 00000c | 02 | A   | 5  | 0   | 2  |
| [ 8] | .gnu.version_r | VERNEED  | 08048274 | 000274 | 000020 | 00 | A   | 6  | 1   | 4  |
| [ 9] | .rel.dyn       | REL      | 08048294 | 000294 | 000008 | 08 | A   | 5  | 0   | 4  |
| [10] | .rel.plt       | REL      | 0804829c | 00029c | 000020 | 08 | A   | 5  | 12  | 4  |
| [11] | .init          | PROGBITS | 080482bc | 0002bc | 000030 | 00 | AX  | 0  | 0   | 4  |
| [12] | .plt           | PROGBITS | 080482ec | 0002ec | 000050 | 04 | AX  | 0  | 0   | 4  |
| [13] | .text          | PROGBITS | 08048340 | 000340 | 0001cc | 00 | AX  | 0  | 0   | 16 |

...



# Sections (cont.)

✓ Ejemplo: Viendo el contenido de la Section .text  
(instrucciones ejecutables)

✓ `objdump -d -j .text test`

```
...
080483f4 <main>:
80483f4:  8d 4c 24 04      lea    0x4(%esp),%ecx
80483f8:  83 e4 f0         and    $0xffffffff0,%esp
80483fb:  ff 71 fc         pushl  -0x4(%ecx)
80483fe:  55              push   %ebp
80483ff:  89 e5            mov    %esp,%ebp
8048401:  51              push   %ecx
8048402:  83 ec 24         sub    $0x24,%esp
8048405:  c7 45 f8 03 00 00 00 movl   $0x3,-0x8(%ebp)
804840c:  c7 04 24 30 85 04 08 movl   $0x8048530,(%esp)
8048413:  e8 14 ff ff ff   call   804832c <puts@plt>
```

...

**.text:** Instrucciones  
ejecutables



# Sections (cont.)

✓ Ejemplo: Viendo el contenido de la Section .data  
(datos inicializados)

✓ `objdump -d -j .data test`

```
...  
0804a018 <global_data>:  
  804a018:  04 00 00 00  
...·
```

**.data:** variables  
inicializadas por el  
usuario

```
/* test.c */  
#include <stdio.h>  
int global_data = 4;  
int global_data_2;  
int main(int argc, char **argv) {  
  int local_data = 3;  
  printf("Hello World\n");  
  printf("global_data = %d\n", global_data);  
  printf("global_data_2 = %d\n", global_data_2);  
  printf("local_data = %d\n", local_data);  
  return (0);  
}
```

✓ En un momento veremos como sabemos que la dirección `0x804a018` corresponde a la variable “global\_data”



# Sections (cont.)

✓ Ejemplo: Viendo el contenido de la Section .bss  
(datos NO inicializados)

✓ `objdump -d -j .bss test`

...  
0804a024 <global\_data\_2>:  
804a024: 00 00 00 00

...

**.bss:** Datos no  
inicializados

```
/* test.c */  
#include <stdio.h>  
int global_data = 4;  
int global_data_2;  
int main(int argc, char **argv) {  
    int local_data = 3;  
    printf("Hello World\n");  
    printf("global_data = %d\n", global_data);  
    printf("global_data_2 = %d\n", global_data_2);  
    printf("local_data = %d\n", local_data);  
    return (0);  
}
```

✓ En un momento veremos como sabemos que la dirección **0x804a024** corresponde a la variable “global\_data\_2”





# Segments

- ✓ Los Segments tienen significado para el Kernel, ya que se utilizan y resuelven en tiempo de ejecución
- ✓ El Kernel los utiliza para ejecutar un programa
- ✓ Un Segment puede corresponderse con una o varias sections
- ✓ El Kernel mantiene estructuras VMA (Virtual Memory Area) para cada proceso:
  - ✓ Se corresponden con un segmento
  - ✓ Cada VMA se corresponde con un conjunto de páginas
- ✓ Los segmentos determinan las propiedades que tendrán las paginas en la memoria (read-only, write, execute)



# Segments (cont.)

☑ Program Header Table es un array de estructuras que indica como se debe crear la imagen del proceso :

✓ Tipo, desplazamiento de comienzo dentro del archivo, dirección virtual donde se encontrara, tamaño, flags (read, write, exec)

```
typedef struct {  
    Elf32_Word    p_type;  
    Elf32_Off     p_offset;  
    Elf32_Addr    p_vaddr;  
    Elf32_Addr    p_paddr;  
    Elf32_Word    p_filesz;  
    Elf32_Word    p_memsz;  
    Elf32_Word    p_flags;  
    Elf32_Word    p_align;  
} Elf32_Phdr;
```



# Segments (cont.)

## ☑ Ejemplo: Viendo la Program Header Table

✓ readelf -l test

...

There are 8 program headers, starting at offset 52

Program Headers:

| Type      | Offset   | VirtAddr   | PhysAddr   | FileSiz | MemSiz  | Flg | Align  |
|-----------|----------|------------|------------|---------|---------|-----|--------|
| PHDR      | 0x000034 | 0x08048034 | 0x08048034 | 0x00100 | 0x00100 | R E | 0x4    |
| INTERP    | 0x000134 | 0x08048134 | 0x08048134 | 0x00013 | 0x00013 | R   | 0x1    |
| LOAD      | 0x000000 | 0x08048000 | 0x08048000 | 0x00578 | 0x00578 | R E | 0x1000 |
| LOAD      | 0x000f0c | 0x08049f0c | 0x08049f0c | 0x00110 | 0x0011c | RW  | 0x1000 |
| DYNAMIC   | 0x000f20 | 0x08049f20 | 0x08049f20 | 0x000d0 | 0x000d0 | RW  | 0x4    |
| NOTE      | 0x000148 | 0x08048148 | 0x08048148 | 0x00020 | 0x00020 | R   | 0x4    |
| GNU_STACK | 0x000000 | 0x00000000 | 0x00000000 | 0x00000 | 0x00000 | RW  | 0x4    |
| GNU_RELRO | 0x000f0c | 0x08049f0c | 0x08049f0c | 0x000f4 | 0x000f4 | R   | 0x1    |



# Segments (cont.)

## ☑ Ejemplo: Viendo la Program Header Table (cont.)

Section to Segment mapping:

Segment Sections...

```
00
01      .interp
02      .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini
.rodata .eh_frame
03      .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
04      .dynamic
05      .note.ABI-tag
06
07      .ctors .dtors .jcr .dynamic .got
```



# Segments (cont.)

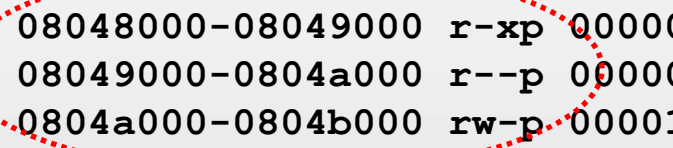
## ☑ Ejemplo: Viendo los Segments en la memoria virtual

- ✓ `gdb test` // para que el programa no se cierre
- ✓ `(gdb) b main` // agregamos un breack point
- ✓ `(gbd) r` // lo ejecutamos
- ✓ Breakpoint 1, 0x08048402 in main ()
- ✓ `ps -eax` // Para ver el PID
- ✓ `cat /proc/<PID>/maps`



# Segments (cont.)

✓ Ejemplo: Viendo los Segments en la memoria virtual (cont.)



```
08048000-08049000 r-xp 00000000 08:01 3096762 /home/user/elf/test
08049000-0804a000 r--p 00000000 08:01 3096762 /home/user/elf/test
0804a000-0804b000 rw-p 00001000 08:01 3096762 /home/user/elf/test
b7db5000-b7db6000 rw-p b7db5000 00:00 0
b7db6000-b7f0e000 r-xp 00000000 08:01 3269238 /lib/tls/i686/cmov/libc-
2.8.90.so
b7f0e000-b7f10000 r--p 00158000 08:01 3269238 /lib/tls/i686/cmov/libc-
2.8.90.so
b7f10000-b7f11000 rw-p 0015a000 08:01 3269238 /lib/tls/i686/cmov/libc-
2.8.90.so
```

.....



# Sections – Symbol Table

- ✓ Es una lista de todos los símbolos (funciones, variables, etc.) que se definen o referencian dentro del archivo
- ✓ Se encuentra definida en el header y sus valores en la section con el tipo específico
- ✓ Cada entrada en la tabla es una estructura:
  - ✓ Nombre, dirección, tamaño, visibilidad (local, global), etc.

```
typedef struct {  
    Elf32_Word      st_name;  
    Elf32_Addr      st_value;  
    Elf32_Word      st_size;  
    unsigned char    st_info;  
    unsigned char    st_other;  
    Elf32_Half      st_shndx;  
} Elf32_Sym;
```





# Tabla de Símbolos (cont.)

☑ Se definen en 2 secciones:

- ✓ “.dynsym”: Tabla de Símbolos para el linking dinámico
- ✓ “.symtab”: Tabla de Símbolos del archivo

☑ Hay una dirección asociada con el símbolo y una indicación sobre el tipo de símbolo.

☑ El tipo de símbolo es luego utilizado para acceder a la tabla de strings



# Tabla de Símbolos (cont.)

☑ Ejemplo: Viendo la tabla de símbolos

✓ readelf -s test

Symbol table '.dynsym' contains 6 entries:

| Num:  | Value    | Size | Type | Bind   | Vis     | Ndx | Name                 |
|-------|----------|------|------|--------|---------|-----|----------------------|
| ..... |          |      |      |        |         |     |                      |
| 3:    | 00000000 | 0    | FUNC | GLOBAL | DEFAULT | UND | printf@GLIBC_2.0 (2) |

Symbol table '.symtab' contains 77 entries:

| Num:  | Value    | Size | Type   | Bind   | Vis     | Ndx | Name                   |
|-------|----------|------|--------|--------|---------|-----|------------------------|
| ..... |          |      |        |        |         |     |                        |
| 57:   | 0804a024 | 4    | OBJECT | GLOBAL | DEFAULT | 24  | global_data_2          |
| ..... |          |      |        |        |         |     |                        |
| 73:   | 0804a018 | 4    | OBJECT | GLOBAL | DEFAULT | 23  | global_data            |
| 74:   | 080484da | 0    | FUNC   | GLOBAL | HIDDEN  | 13  | __i686.get_pc_thunk.bx |
| 75:   | 080483f4 | 111  | FUNC   | GLOBAL | DEFAULT | 13  | main                   |
| 76:   | 080482bc | 0    | FUNC   | GLOBAL | DEFAULT | 11  | _init                  |

Nos permite relacionar la dirección de memoria con el valor del programa (símbolo)



# Tabla de Strings

- ☑ Es una lista de los strings (cadenas de caracteres) que se utilizan dentro del archivo
- ☑ Se definen en 2 secciones:
  - ✓ “.dynstr”: Tabla de Símbolos para el linking dinámico
  - ✓ “.strtab”: Tabla de Símbolos del archivo
- ☑ Se almacenan los nombres de las sections, de los símbolos, etc
  - ✓ Las estructuras que vimos de sections o símbolos, en el campo del nombre contiene como valor el índice dentro de esta tabla



# Tabla de Strings (cont.)

## ☑ Ejemplo

| Index | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 | +8 | +9 |
|-------|----|----|----|----|----|----|----|----|----|----|
| 0     | \0 | n  | a  | m  | e  | .  | \0 | V  | a  | r  |
| 10    | i  | a  | b  | l  | e  | \0 | a  | b  | l  | e  |
| 20    | \0 | \0 | x  | x  | \0 |    |    |    |    |    |

| Index | String             |
|-------|--------------------|
| 0     | <i>none</i>        |
| 1     | name.              |
| 7     | Variable           |
| 11    | able               |
| 16    | able               |
| 24    | <i>null string</i> |



# Relocations

- ✓ Es el proceso de conectar referencias simbólicas con sus definiciones.
- ✓ Cuando un programa llama a una función, la llamada a instrucción asociada debe transferir el control a la correspondiente dirección de memoria destino en tiempo de ejecución
- ✓ Los llamados a funciones externas (librerías) necesitan ser resueltos durante la ejecución por lo que se necesita información de su ubicación
  - En nuestro ejemplo llamados a **printf()**
- ✓ Esta información es necesaria durante el proceso linking dinámico
- ✓ La información en los archivos relocatables, permite a los ejecutables y shared objects, mantener información correcta para las imágenes de proceso



# Relocations (cont.)

- ☑ Se almacena en las sections “.rel#name” y “.rela#name”
  - ✓ donde #name es el nombre de la section donde se debe aplicar el cambio

```
typedef struct {  
    Elf32_Addr    r_offset;  
    Elf32_Word    r_info;  
} Elf32_Rel;
```

```
typedef struct {  
    Elf32_Addr    r_offset;  
    Elf32_Word    r_info;  
    Elf32_Sword    r_addend;  
} Elf32_Rela;
```



# Relocations (cont.)

☑ Ejemplo: Viendo las relocations

✓ readelf -r test

Relocation section '.rel.dyn' at offset 0x294 contains 1 entries:

| Offset   | Info     | Type           | Sym.Value | Sym. Name      |
|----------|----------|----------------|-----------|----------------|
| 08049ff0 | 00000106 | R_386_GLOB_DAT | 00000000  | __gmon_start__ |

Relocation section '.rel.plt' at offset 0x29c contains 4 entries:

| Offset   | Info     | Type            | Sym.Value | Sym. Name         |
|----------|----------|-----------------|-----------|-------------------|
| 0804a000 | 00000107 | R_386_JUMP_SLOT | 00000000  | __gmon_start__    |
| 0804a004 | 00000207 | R_386_JUMP_SLOT | 00000000  | __libc_start_main |
| 0804a008 | 00000307 | R_386_JUMP_SLOT | 00000000  | printf            |
| 0804a00c | 00000407 | R_386_JUMP_SLOT | 00000000  | puts              |





## Como se resuelven los llamados a rutinas externas

- ✓ La llamada a `printf()` se compila a:

```
8048421: c7 04 24 3c 85 04 08    movl    $0x804831c, (%esp)
8048428: e8 ef fe ff ff          call    <printf@plt>
```

- ✓ La dirección `0x804831c` se encuentra con una sección `".plt"` (Procedure Linkage Table) que contiene lo siguiente:

0804831c <printf@plt>:

```
804831c: ff 25 08 a0 04 08    jmp     *0x804a008
8048322: 68 10 00 00 00      push    $0x10
8048327: e9 c0 ff ff ff      jmp     80482ec <_init+0x30>
```

Relocation

- ✓ La dirección `0x804a008` se encuentra con una sección `".got"` (Global Offset Table) que contiene lo siguiente:

0x804a008

Jump

08048322

....



## Como se resuelven los llamados a rutinas externas

- ✓ Las siguientes instrucciones en la sección “.plt” indican un llamado al Dinamic Linker:

0804831c <printf@plt>:

|          |                   |      |                      |
|----------|-------------------|------|----------------------|
| 804831c: | ff 25 08 a0 04 08 | jmp  | *0x804a008           |
| 8048322: | 68 10 00 00 00    | push | \$0x10               |
| 8048327: | e9 c0 ff ff ff    | jmp  | 80482ec <_init+0x30> |

- ✓ Cuando el Dinamic Linker se ejecuta, actualiza la entrada en la “.got” colocando la dirección donde se cargo la rutina printf()
  - ✓ Así la próxima vez que se la invoca printf(), la dirección ya se encuentra resuelta



## Como se resuelven los llamados a rutinas externas

☑ Vemos que:

- ✓ La carga se hace bajo demanda
  - ◆ Cuando se utiliza la función, su código es subido al espacio de memoria
- ✓ Ver que la sección “.got” se cargo en un segmento que tiene la propiedad de ser Writable
  - ◆ No así la sección “.plt” que está en Read Only



# Material Adicional

- [http://www.linuxforums.org/misc/understanding\\_elf\\_using\\_readelf\\_and\\_objdump.html](http://www.linuxforums.org/misc/understanding_elf_using_readelf_and_objdump.html)
- [http://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](http://en.wikipedia.org/wiki/Executable_and_Linkable_Format)

