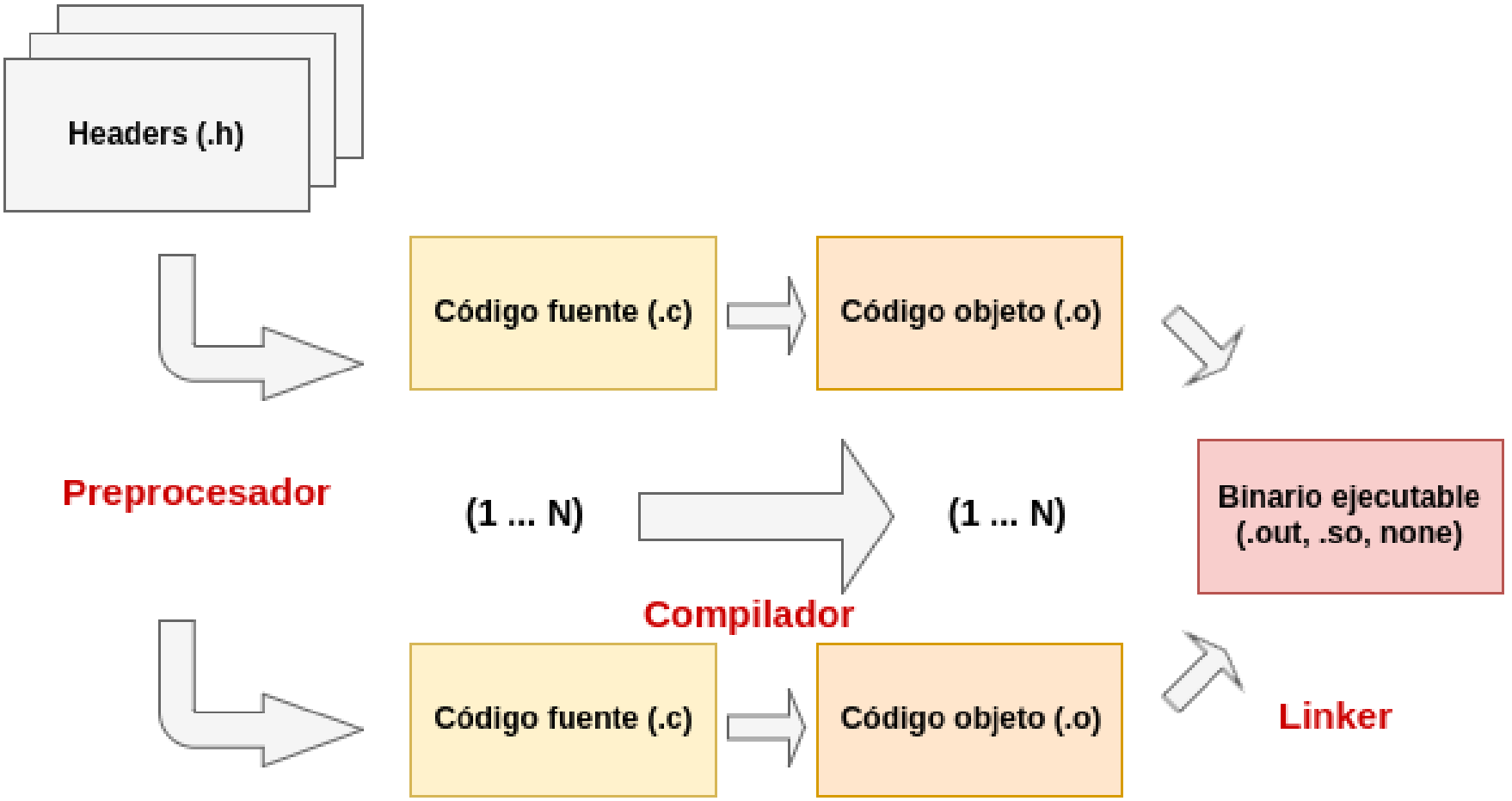
Ingeniería Inversa
Clase 1

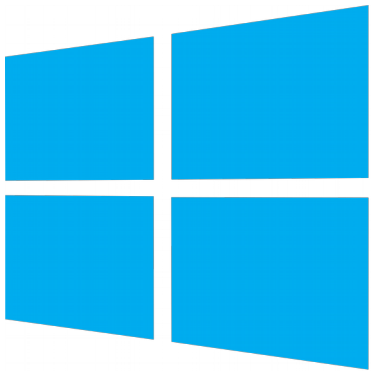
Binarios Ejecutables

martin.uy
Open by default.

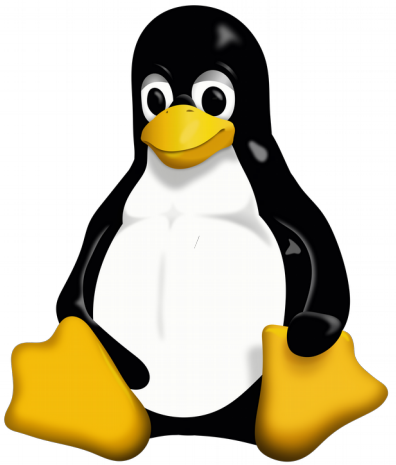
Compilación



Binarios ejecutables



EXE (PE)



ELF

MACH-O

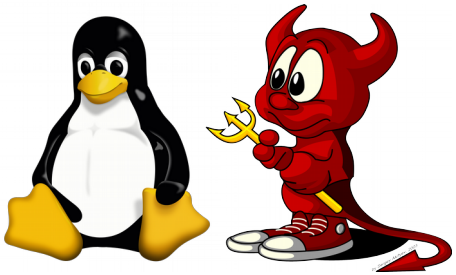
.NET assemblies

CLASSFILE

Binarios ejecutables

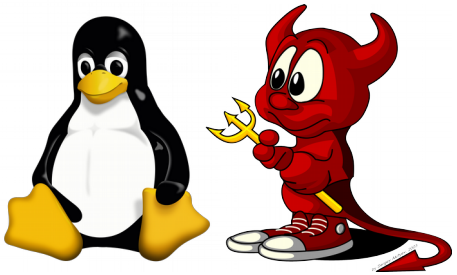
- Instrucciones ejecutables
  - por el procesador (x86, ARM, etc.)
  - por el intérprete de una máquina virtual (JVM, CPython, etc.)
- Datos
  - constantes (embebidos en opcodes de instrucciones o no)
  - variables
- Información de debugging

ELF



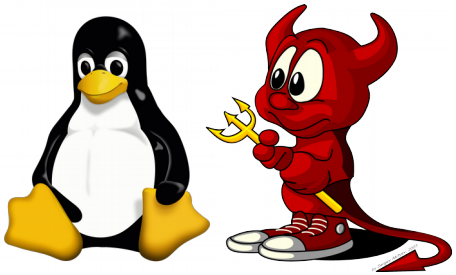
- Executable and Linkable Format (ELF)
  - Especificación de referencia de la Linux Foundation
  - ELF headers
    - <glibc>/elf/elf.h (glibc)
    - /usr/include/linux/elf.h (kernel)
    - man elf
  - Especifica binarios ejecutables, librerías (shared-objects), objetos (.o), módulos del kernel, el kernel en sí mismo
  - Formato binario. ¿Por qué? ¿Podría no serlo?

ELF



- Reconocen el formato ELF:
  - kernel
    - al cargar módulos y drivers
    - sys_execve para lanzar binarios ejecutables
  - dynamic loader (ld-linux, glibc)
    - al cargar librerías compartidas
  - bootloader
    - al cargar el kernel
  - compiladores, linkers, ensambladores, debuggers y otras herramientas (objdump, readelf, etc.)

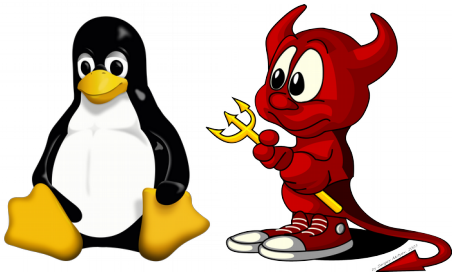
ELF



```
00000000 7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00 .ELF............
00000010 03 00 3E 00 01 00 00 00 70 F0 02 00 00 00 00 00 ..>.....p.......
00000020 40 00 00 00 00 00 00 00 48 54 10 00 00 00 00 00 @.......HT......
00000030 00 00 00 00 40 00 38 00 09 00 40 00 1D 00 1C 00 ....@.8...@.....
00000040 06 00 00 00 05 00 00 00 40 00 00 00 00 00 00 00 ........@.......
00000050 40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 @.......@.......
00000060 F8 01 00 00 00 00 00 00 F8 01 00 00 00 00 00 00 ................
00000070 08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00 ................
00000080 38 02 00 00 00 00 00 00 38 02 00 00 00 00 00 00 8.......8.......
00000090 38 02 00 00 00 00 00 00 1C 00 00 00 00 00 00 00 8...............
000000a0 1C 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
000000b0 01 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00 ................
000000c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000d0 54 75 0F 00 00 00 00 00 54 75 0F 00 00 00 00 00 Tu......Tu......
000000e0 00 00 20 00 00 00 00 00 01 00 00 00 06 00 00 00 .. .............
0000000f A8 7E 0F 00 00 00 00 00 A8 7E 2F 00 00 00 00 00 .~.......~/.....
00000100 A8 7E 2F 00 00 00 00 00 C0 B6 00 00 00 00 00 00 .~/.............
00000110 90 10 01 00 00 00 00 00 00 00 20 00 00 00 00 00 .......... .....
00000120 02 00 00 00 06 00 00 00 80 A6 0F 00 00 00 00 00 ................
00000130 80 A6 2F 00 00 00 00 00 80 A6 2F 00 00 00 00 00 ../......./.....
00000140 E0 01 00 00 00 00 00 00 E0 01 00 00 00 00 00 00 ................
```

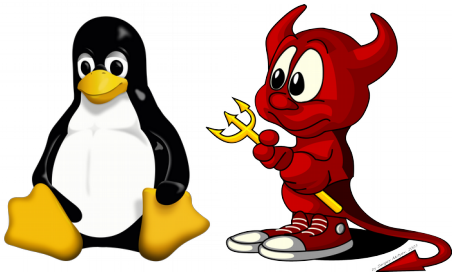
/bin/bash (Linux, x86_64)

ELF



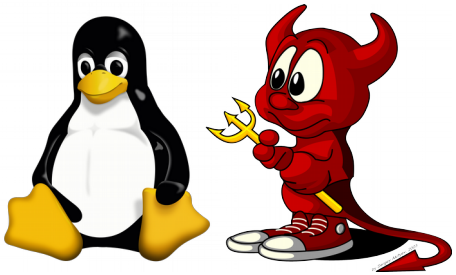
- File header
  - Número mágico → “ELF” (0x7F, 0x45, 0x4C, 0x46)
  - 32 o 64 bits
  - Arquitectura → Linux soporta múltiples plataformas
  - Endianness → ¿Little-endian? ¿Big-endian?
  - Versión de ELF
  - ABI (Application Binary Interface) y versión de ABI
    - Ej. System V

ELF



- Tipo (objeto, binario ejecutable, librería compartida)
- Entry point → primera instrucción ejecutable
- Offset a tablas (en el archivo)
  - Program Headers
  - Section Headers
- Tamaños de las tablas (tamaño de cada entrada, cantidad de entradas)
- Tamaño del ELF header

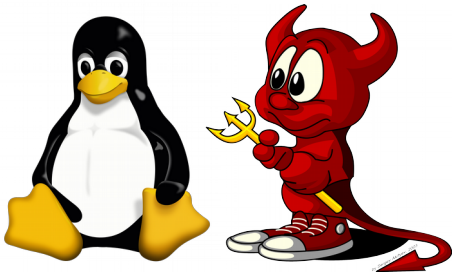
ELF



```
typedef struct
{
  unsigned char  e_ident[EI_NIDENT]; /* Magic number and other info */
  Elf64_Half  e_type;           /* Object file type */
  Elf64_Half  e_machine;        /* Architecture */
  Elf64_Word  e_version;        /* Object file version */
  Elf64_Addr  e_entry;      /* Entry point virtual address */
  Elf64_Off   e_phoff;      /* Program header table file offset */
  Elf64_Off   e_shoff;      /* Section header table file offset */
  Elf64_Word  e_flags;      /* Processor-specific flags */
  Elf64_Half  e_ehsize;     /* ELF header size in bytes */
  Elf64_Half  e_phentsize;      /* Program header table entry size */
  Elf64_Half  e_phnum;      /* Program header table entry count */
  Elf64_Half  e_shentsize;      /* Section header table entry size */
  Elf64_Half  e_shnum;      /* Section header table entry count */
  Elf64_Half  e_shstrndx;       /* Section header string table index */
} Elf64_Ehdr;
```

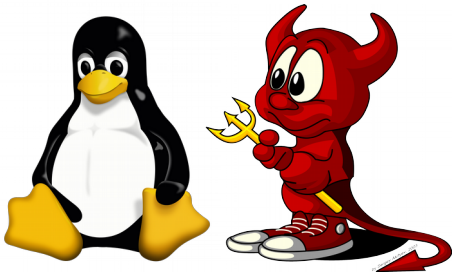
file header - elf.h (glibc)

ELF



¿Cómo interpretar un ELF header? (dado un stream de bytes y una definición de la estructura del header)

ELF



¿Cómo interpretar un ELF header? (dado un stream de bytes y una definición de la estructura del header)

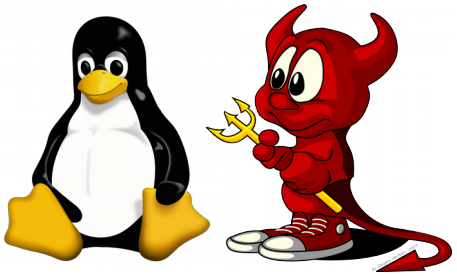
```
char* file_buffer = { 0x7F, 0x45, 0x4C, … };

Elf64_Ehdr *elf_header = (Elf64_Ehdr*)file_buffer;

elf_header->e_type;
```

CAST

Esto es lo que hace el kernel, glibc, objdump, readelf y otras tools para estructuras de largo fijo.

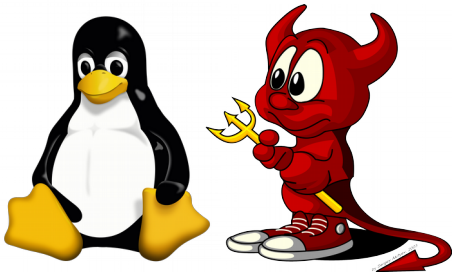
Se incrementa el ptr por sizeof(estructura) para seguir interpretando.

ELF



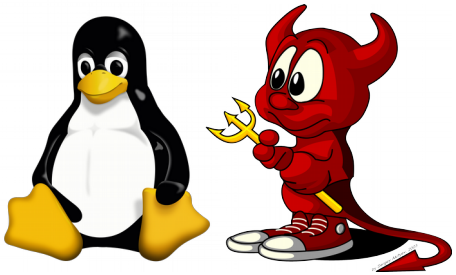
```
[martin@vmlinwork 1]$ readelf -h main
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400400
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6728 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

ELF



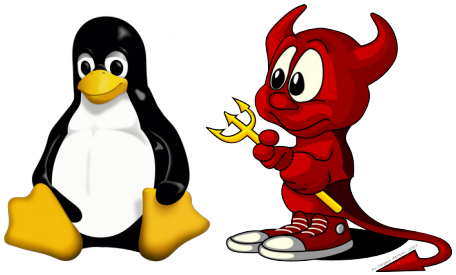
- **Program headers**
  - Tabla con información para preparar al binario para su ejecución, utilizada también en runtime
    - PT_LOAD → mapa de segmentos de memoria virtual al binario ejecutable:
      - offset en el archivo
      - dirección virtual + alineación
      - permisos (R, W, X)
      - tamaño en archivo
      - tamaño en memoria
        - Se completa con 0s si es mayor al tamaño en archivo (ej. variables globales no inicializadas)

ELF



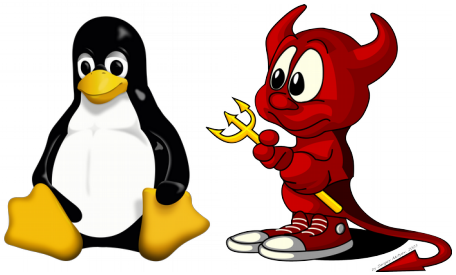
- **Program headers**
  - PT_LOAD
    - ¿Qué significa mapear segmentos de memoria virtual a un archivo?
    - Si el offset en el archivo que queremos mapear es 0xdf8, ¿qué granularidad tenemos para mapearlo a 0x600000?
    - ¿Qué pasa cuando los segmentos se pueden escribir? ¿Se escribe el archivo?

ELF



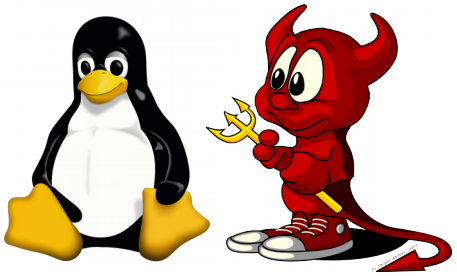
- **Program headers**
  - PT_LOAD
    - Memoria virtual respaldada por un file. En caso de page_fault, el SO carga los datos en memoria desde el archivo. Por lo tanto, el binario ejecutable no se carga en memoria de una pero de a pedazos conforme sea necesario.
    - Granularidad por páginas (ej. 4096 bytes). Por lo tanto, se mapea el archivo desde el offset 0 a la dirección virtual 0x600000.
    - Depende. En este caso, no se escribe el file en disco.

ELF



- ¿Quién usa la información de PT_LOAD en un binario ejecutable?

ELF

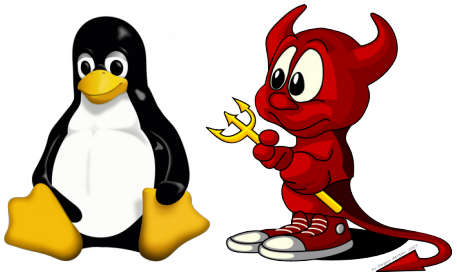


- ¿Quién usa la información de PT_LOAD en un binario ejecutable?

Linux kernel: sys_execve mapea rangos de direcciones virtuales del proceso a un binario ejecutable (datos, instrucciones).

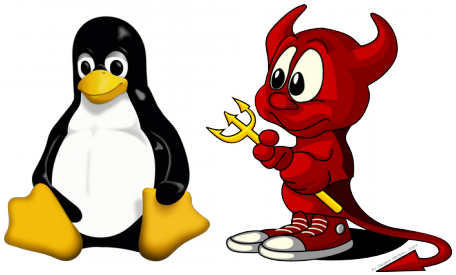
Dynamic loader: hace lo mismo para librerías dinámicas.

ELF



- Nota: los binarios y librerías dinámicas pueden NO especificar a qué direcciones virtuales ser mapeados
  - Position Independent Executables y Position Independent Code
  - Las direcciones se eligen aleatoriamente
  - En este caso, la entrada PT_LOAD no especifica una dirección virtual absoluta (solo offsets, tamaños, permisos, alineación, etc.)

ELF



```
tatic unsigned long elf_map(struct file *filep, unsigned long addr,
        struct elf_phdr *eppnt, int prot, int type,
        unsigned long total_size)

  unsigned long map_addr;
  unsigned long size = eppnt->p_filesz + ELF_PAGEOFFSET(eppnt->p_vaddr);
  unsigned long off = eppnt->p_offset - ELF_PAGEOFFSET(eppnt->p_vaddr);
  addr = ELF_PAGESTART(addr);
  size = ELF_PAGEALIGN(size);

  /* mmap() will return -EINVAL if given a zero size, but a
   * segment with zero filesize is perfectly valid */
  if (!size)
      return addr;
```

kernel_dev [C/C++ Attach to Applicat
vmlinux
Thread #1 1 (CPU#0 [running]) (
elf_map() at binfmt_elf.c:346 0
load_elf_binary() at binfmt_elf.
search_binary_handler() at exe
exec_binprm() at exec.c:1,668
do_execveat_common() at exe
do_execve() at exec.c:1,834 0x
SYSC_execve() at exec.c:1,915
SyS_execve() at exec.c:1,910 0
do_syscall_64() at common.c:2

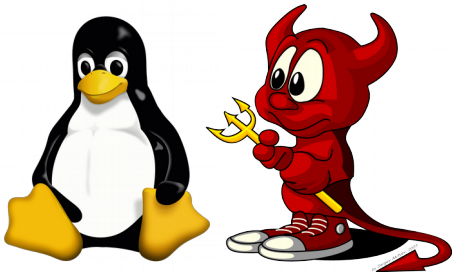
ole Tasks Problems Executables Debugger Console Memory Progress Search

ev [C/C++ Attach to Application] gdb (7.12.1)

0x8048000

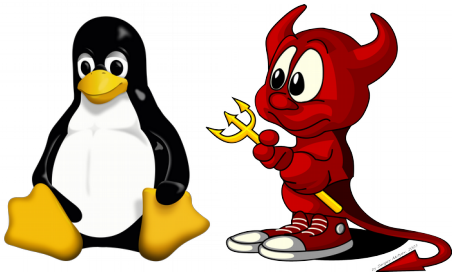
main_32 mapeado en Linux kernel (x86_64) en base a información de PT_LOAD – elf_map – fs/binfmt_elf.c. VirtAddress en PT_LOAD para el segmento exec es 0x08048000

ELF



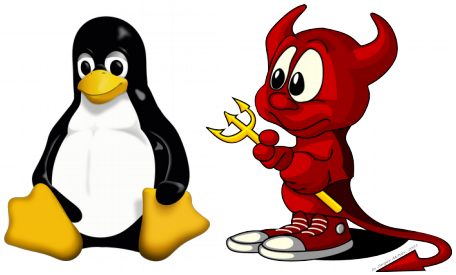
- PT_INTERP → intérprete dinámico del binario (dynamic loader)
  - /lib64/ld-linux-x86-64.so.2
    - <glibc>/elf/ (código fuente)
  - Ejecuta antes que el binario y carga librerías contra las que el binario linkea dinámicamente. Puede no haber (binarios auto-contenidos), pero en general hay.
- PT_DYNAMIC → información para el linker dinámico: ¿en qué dirección virtual del proceso encuentra esta información? (que según PT_LOAD fue mapeada del file a un segmento virtual con permisos de RW)
- PT_PHDR → localización en memoria virtual de la tabla Program Headers

ELF



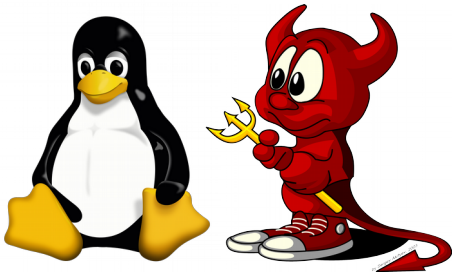
```
[martin@vmlinwork 1]$ readelf -l main

Elf file type is EXEC (Executable file)
Entry point 0x400400
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000006e4 0x00000000000006e4  R E    200000
  LOAD           0x0000000000000e08 0x0000000000600e08 0x0000000000600e08
                 0x000000000000021c 0x0000000000000238  RW     200000
  DYNAMIC        0x0000000000000e20 0x0000000000600e20 0x0000000000600e20
                 0x00000000000001d0 0x00000000000001d0  RW     8
```

ELF



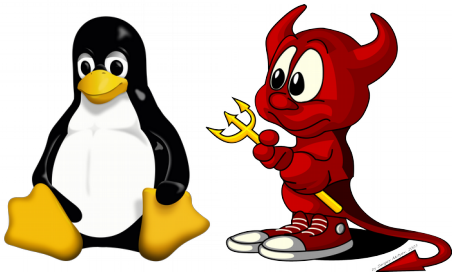
- **Section headers**
  - Mapa de las secciones del binario ejecutable en disco (archivo). Cada sección es una “unidad de información”
    - Nombre de la sección
    - Tipo de la sección. Ejemplo:
      - Tabla de símbolos
      - Tabla de strings
      - Tabla de relocalizaciones
      - Tabla de símbolos dinámicos
      - Tabla de hashes
      - Tabla de linkeo dinámico (referenciada desde los Program Headers)

ELF



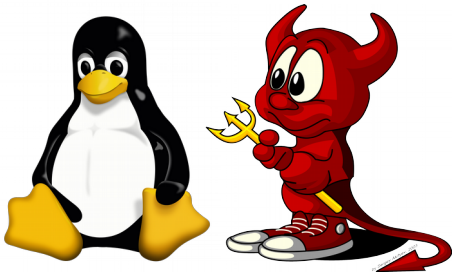
- Contiene
  - Offset de la sección en el archivo
  - Tamaño de la sección en el archivo (en memoria si la sección es de tipo NOBITS)
  - Dirección virtual de la sección
  - Alineación (para ser cargada/mapeada al espacio de direcciones virtuales; en disco las secciones están contiguas)
  - Permisos (Read? Write? Execute?)
  - Tamaño de la entrada, si la sección es una tabla

ELF



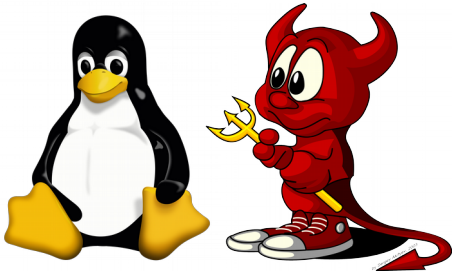
- Esta tabla (en el file) NO es mapeada al espacio de direcciones virtuales del proceso creado
  - Ej: readelf -S main_32
  - “Hay 36 section headers, empezando en el offset 0x2700:”
  - Sin embargo, PT_LOAD mapea como máximo 0x00124 bytes del archivo empezando en el offset 0x000f00. O sea hasta el offset 0x1024.
- Los datos referenciados por esta tabla (las secciones en sí mismas) SÍ pueden estar mapeados. Ejemplo: sección .text. Ver tabla Program Headers y calcular overlap de offsets en el archivo.
- Esta tabla se utiliza al momento de cargar el binario. Ej: ¿dónde se tiene que alocar memoria para la sección .bss con variables globales no inicializadas?

ELF



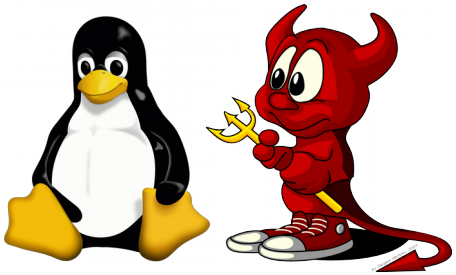
```
typedef struct
{
  Elf64_Word    sh_name;    /* Section name (string tbl
index) */
  Elf64_Word    sh_type;    /* Section type */
  Elf64_Xword   sh_flags;      /* Section flags */
  Elf64_Addr    sh_addr;    /* Section virtual addr at
execution */
  Elf64_Off sh_offset;      /* Section file offset */
  Elf64_Xword   sh_size;    /* Section size in bytes */
  Elf64_Word    sh_link;    /* Link to another section */
  Elf64_Word    sh_info;    /* Additional section
information */
  Elf64_Xword   sh_addralign;     /* Section alignment */
  Elf64_Xword   sh_entsize;   /* Entry size if section
holds table */
} Elf64_Shdr;
```

ELF



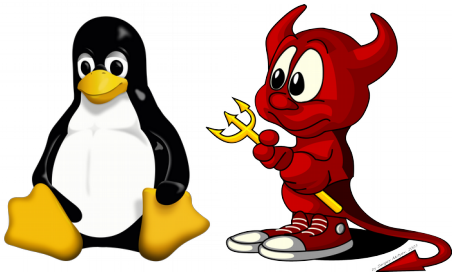
```
[martin@vmlinwork 1]$ readelf -S main
There are 35 section headers, starting at offset 0x2270:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000400298  00000298
       000000000000001c  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000004002b8  000002b8
       0000000000000060  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000400318  00000318
       000000000000003d  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           0000000000400356  00000356
       0000000000000008  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000400360  00000360
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000400380  00000380
       0000000000000030  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             00000000004003b0  000003b0
       0000000000000018  0000000000000018  AI       5    23     8
  [11] .init             PROGBITS         00000000004003c8  000003c8
       0000000000000017  0000000000000000  AX       0     0     4
```

ELF



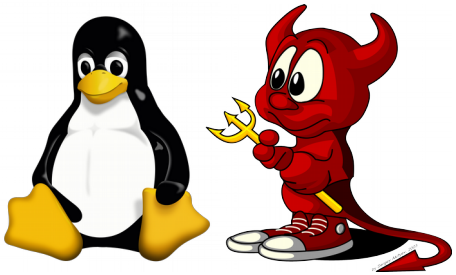
- Tabla de símbolos y Tabla de strings
  - Nombre del símbolo (índice en tabla de strings)
  - Tipo de símbolo (no especificado, función, objeto, file, sección, TLS data, etc.)
  - Visibilidad del símbolo (local, global, débil, etc.)
  - Sección del símbolo
  - Valor del símbolo (ej. offset donde se encuentra dentro de la sección)
  - Tamaño del símbolo

ELF



```
typedef struct
{
  Elf64_Word     st_name;      /* Symbol name (string tbl index) */
  unsigned char  st_info;      /* Symbol type and binding */
  unsigned char  st_other;     /* Symbol visibility */
  Elf64_Section  st_shndx;         /* Section index */
  Elf64_Addr     st_value;     /* Symbol value */
  Elf64_Xword    st_size;      /* Symbol size */
} Elf64_Sym;
```

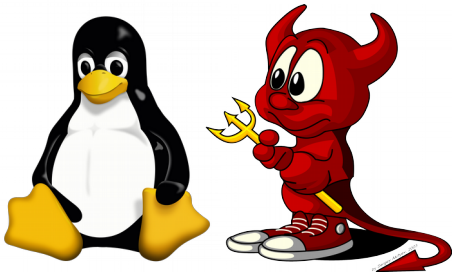
ELF



```c
extern int var_a;

int var_a = 2;

static int var_b = 1;

extern int func_a (void);

static void func_b (void);

int main (void) {
    return func_a();
}

static void func_b (void) {
}
```

main.c

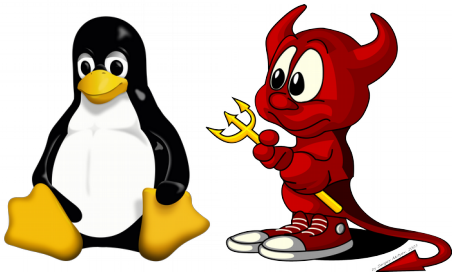
ELF



```
Symbol table '.symtab' contains 13 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    3 var_b
     6: 000000000000000b     7 FUNC    LOCAL  DEFAULT    1 func_b
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
    10: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 var_a
    11: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 main
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND func_a
```

main.o → gcc -o main.o main.c

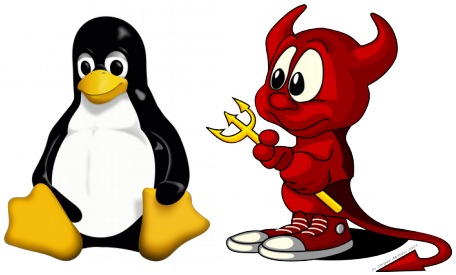
ELF



```
0000:0210 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0000:0220 00 6D 61 69  6E 2E 63 00  76 61 72 5F  62 00 66 75 .main.c.var b.fu
0000:0230 6E 63 5F 62  00 76 61 72  5F 61 00 6D  61 69 6E 00 nc b.var a.main.
0000:0240 66 75 6E 63  5F 61 00 00  05 00 00 00  00 00 00 00 func a..........
0000:0250 02 00 00 00  0C 00 00 00  FC FF FF FF  FF FF FF FF ........üÿÿÿÿÿÿÿ
0000:0260 20 00 00 00  00 00 00 00  02 00 00 00  02 00 00 00  ...............
0000:0270 00 00 00 00  00 00 00 00  40 00 00 00  00 00 00 00 ........@.......
0000:0280 02 00 00 00  02 00 00 00  0B 00 00 00  00 00 00 00 ................
```

main.o → string table

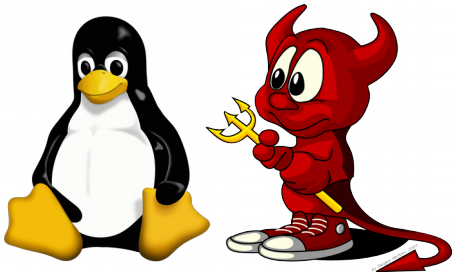
ELF



```
Symbol table '.symtab' contains 71 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND _
    57: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
    58: 0000000000601020     0 NOTYPE  GLOBAL DEFAULT   24 __data_start
    59: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    60: 00000000004005a8     0 OBJECT  GLOBAL HIDDEN    15 __dso_handle
    61: 00000000004005a0     4 OBJECT  GLOBAL DEFAULT   15 _IO_stdin_used
    62: 0000000000400520   101 FUNC    GLOBAL DEFAULT   13 __libc_csu_init
    63: 0000000000601028     0 NOTYPE  GLOBAL DEFAULT   25 _end
    64: 0000000000400400    43 FUNC    GLOBAL DEFAULT   13 _start
    65: 0000000000601024     0 NOTYPE  GLOBAL DEFAULT   25 __bss_start
    66: 00000000004004f6    32 FUNC    GLOBAL DEFAULT   13 main
    67: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    68: 0000000000601028     0 OBJECT  GLOBAL HIDDEN    24 __TMC_END
```

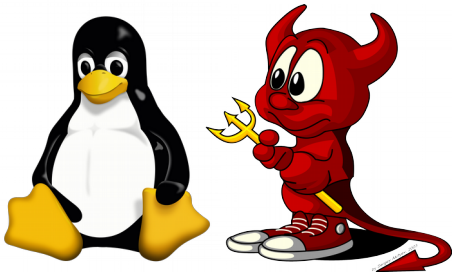
main → binario linkeado (contra glibc)

ELF



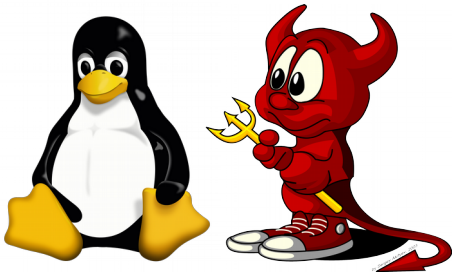
- Tabla Dynamic
  - Es un “mapa” usado por el dynamic linker en tiempo de ejecución.
  - Librerías compartidas requeridas (nombres)
  - Dirección de las tablas en la memoria y tamaños
    - Hashes, Strings, Símbolos, Relocalización, etc.
  - Dirección de la Tablas GOT y PLT en memoria
  - Dirección de las funciones de inicialización y finalización
  - Termina en NULL

ELF



- ¿Información redundante con la Tabla de Secciones?
  - La Tabla de Secciones no está mapeada en la memoria del proceso.
  - La Tabla Dynamic sí lo está. ¿Por qué? Ej.:
    - El kernel carga un binario ejecutable
    - Transfiere el control al intérprete (dynamic loader)
    - El dynamic loader necesita, entre otras cosas, saber contra qué librerías linkea el binario ejecutable y mapearlas a la memoria del proceso (ejemplo: libc)

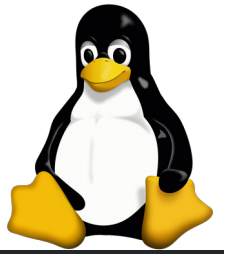
ELF



```
typedef struct
{
  Elf64_Sxword  d_tag;          /* Dynamic entry type */
  union
    {
      Elf64_Xword d_val;     /* Integer value */
      Elf64_Addr d_ptr;         /* Address value */
    } d_un;
} Elf64_Dyn;
```

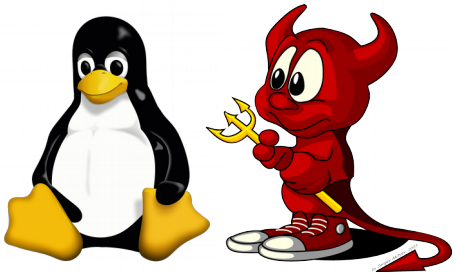
d_tag puede valer: DT_NULL, DT_NEEDED, DT_PLTGOT, DT_STRTAB, DT_SYMTAB, DT_RELA, DT_RELASZ, etc.

ELF



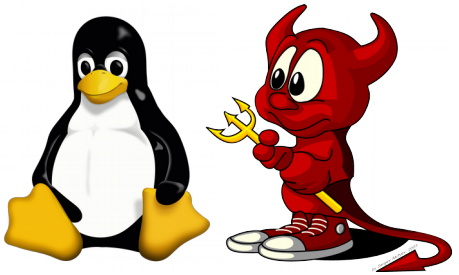
```
[martin@vmlinwork 1]$ readelf -d main

Dynamic section at offset 0xe20 contains 24 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x4003c8
 0x000000000000000d (FINI)               0x400594
 0x0000000000000019 (INIT_ARRAY)         0x600e08
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x600e10
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x400298
 0x0000000000000005 (STRTAB)             0x400318
 0x0000000000000006 (SYMTAB)             0x4002b8
 0x000000000000000a (STRSZ)              61 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x601000
 0x0000000000000002 (PLTRELSZ)           24 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x4003b0
 0x0000000000000007 (RELA)               0x400380
 0x0000000000000008 (RELASZ)             48 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x400360
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x400356
 0x0000000000000000 (NULL)               0x0
```

ELF



Si el dynamic loader tiene que cargar un binario y conoce únicamente su base address virtual,
¿cómo llega a la dynamic table?

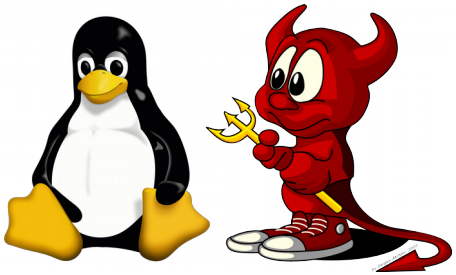
ELF



Si el dynamic loader tiene que cargar un binario y conoce únicamente su base address virtual, **¿cómo llega a la dynamic table?**

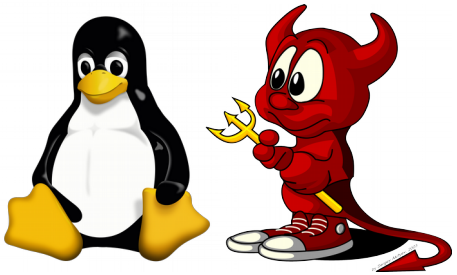
- Empieza leyendo desde la base address
- Salta el File Header (64 bytes) hasta llegar a la tabla Program Headers
- La parsea y encuentra la dirección de la tabla Dynamic
- Luego puede, por ejemplo, ver cuales librerías son requeridas (DT_NEEDED), localizar los nombres de esas librerías (tabla de símbolos y strings) y cargarlas
- El kernel ya hizo el trabajo duro de mapear toda esta información desde el archivo binario ejecutable a la memoria virtual

ELF



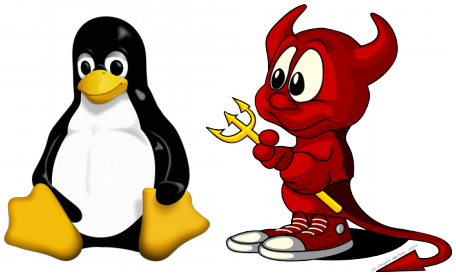
- Tabla de Relocalizaciones
  - Cuando un objeto llama a una función externa, no se sabe la dirección u offset hasta el linkeo
  - Placeholder:
    - CALL ??? → + entrada de relocalización
  - Ídem para variables globales presentes en otros objetos
  - Cuando se linkea, se actualizan los placeholders con las direcciones correctas

ELF



- Relocalizar un binario o librería
  - Las librerías actuales son normalmente PIC (Position Independent Code)
  - Los binarios normalmente no son PIC en x86 y sí lo son en x86_64
  - Si no PIC/PIE, pueden haber direcciones absolutas (ej. CALL dirección-absoluta). Si PIC/PIE, se usa direccionamiento relativo pero las relocalizaciones son aún requeridas para las tablas GOT/PLT
  - No todos los binarios son relocalizables: depende si hay información para relocalizarlos

ELF

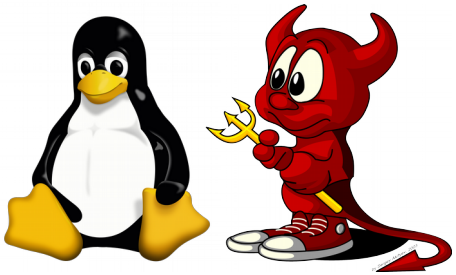


```
Relocation section '.rela.text' at offset 0x248 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name +
000000000005  000c00000002 R_X86_64_PC32     0000000000000000 func_a - 4

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   e8 00 00 00 00          callq  9 <main+0x9>
   9:   5d                      pop    %rbp
   a:   c3                      retq
```

main.o

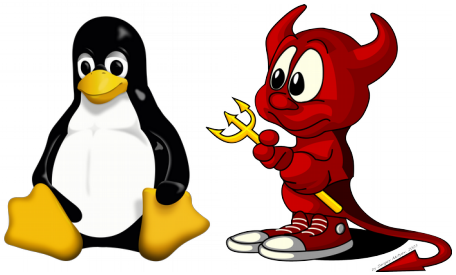
Relocalización para .text: en el offset 5 insertar el valor del símbolo func_a cuando se resuelva (al linkear). El valor es de 32 bits (call near).

ELF



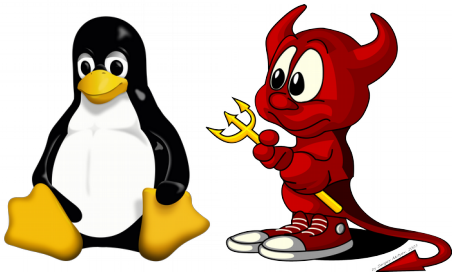
```
[martin@vmlinwork 1]$ readelf -r main

Relocation section '.rela.dyn' at offset 0x380 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600ff0  000200000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000600ff8  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x3b0 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
```

ELF



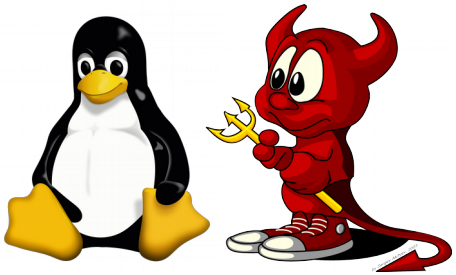
- Secciones de datos
  - .data
    - Variables globales inicializadas
    - .rodata → read-only
  - .bss
    - Variables globales no-inicializadas
    - No están en el archivo, solo en memoria
    - Se inicializan con 0s

ELF



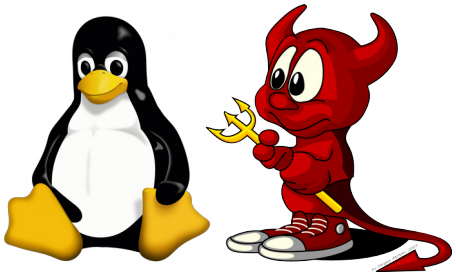
```c
#include <stdio.h>

int a;
int b = 1;
const int c = 2;

int main(int argc, char* argv[]){
    printf("hola\n");
   return 0;
}
```

ELF



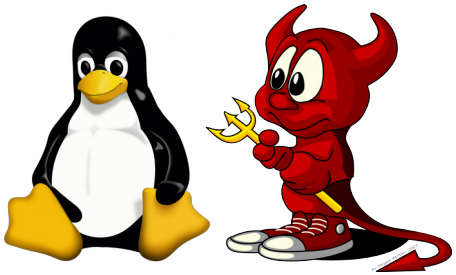
```
[martin@vmlinwork 1]$ readelf -x 25 main

Section '.bss' has no data to dump.
[martin@vmlinwork 1]$ readelf -x 24 main

Hex dump of section '.data':
  0x00601020 00000000 01000000                    ........

[martin@vmlinwork 1]$ readelf -x 15 main

Hex dump of section '.rodata':
  0x004005a0 01000200 00000000 00000000 00000000 ................
  0x004005b0 02000000 686f6c61 00                 ....hola.
```

ELF



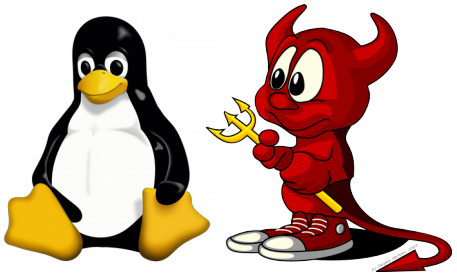
- Instrucciones ejecutables: .text
  - Opcodes de la arquitectura
    - Implementación de funciones
    - Stubs estáticos de la *glibc* (si se linkea)
      - Ej. rutinas de inicialización como *_start* (<glibc>/sysdeps/x86_64/start.S) y *__libc_csu_init* (<glibc>/csu/elf-init.c)
- Segmento legible y ejecutable (no escribible)

ELF



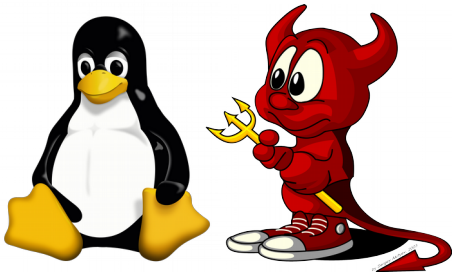
```
[martin@vmlinwork 1]$ objdump -d main | grep -A 14 -e "<_start>"
0000000000400400 <_start>:
  400400:       31 ed                   xor    %ebp,%ebp
  400402:       49 89 d1                mov    %rdx,%r9
  400405:       5e                      pop    %rsi
  400406:       48 89 e2                mov    %rsp,%rdx
  400409:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
  40040d:       50                      push   %rax
  40040e:       54                      push   %rsp
  40040f:       49 c7 c0 90 05 40 00    mov    $0x400590,%r8
  400416:       48 c7 c1 20 05 40 00    mov    $0x400520,%rcx
  40041d:       48 c7 c7 f6 04 40 00    mov    $0x4004f6,%rdi
  400424:       ff 15 c6 0b 20 00       callq  *0x200bc6(%rip)        # 600ff0 <_DYNAMIC+0x1d0>
  40042a:       f4                      hlt
  40042b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

ELF



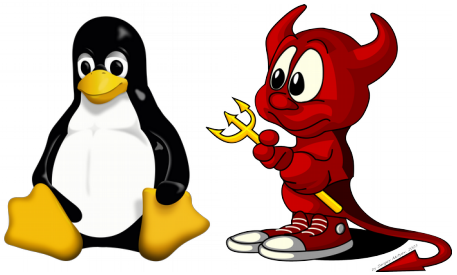
- Global Offset Table (GOT)
  - Un binario o una librería usa funciones y variables globales exportadas por otras librerías
  - Las librerías son Position-Independent-Code (PIC) actualmente
  - En tiempo de compilación del binario o librería, no sabemos en qué dirección virtual van a estar las funciones y variables globales externas
  - CALL ??? → ¿Cómo resolvemos este problema?

ELF



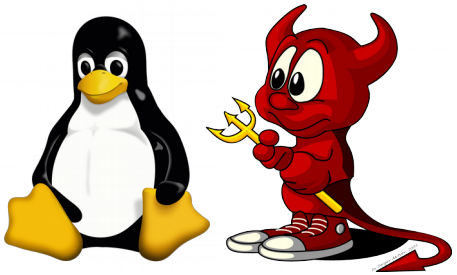
- Global Offset Table (GOT)
  - Tabla en memoria para cada binario o librería
  - Dirección en memoria de las funciones/variables externas (de otras librerías)
  - Se va llenando en *runtime* con valores absolutos (direcciones virtuales)
  - MOV *<entrada-en-GOT-para-la-variable>
    - Direccionamiento relativo a la GOT

ELF



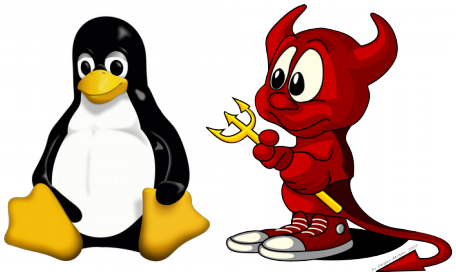
- Procedure Linkage Table (PLT)
  - Trampolines para llamar a funciones externas usando la GOT (un trampolín por función externa)
  - Código ejecutable (segmento ejecutable)
  - En lugar de CALL *<entrada-en-GOT-para-funcion>, es CALL/JMP <entrada-en-PLT-para-la-función>

ELF



- Procedure Linkage Table (PLT)
  - El trampolín ejecuta JMP *<entrada-en-GOT-para-funcion>
  - Cuando la entrada de la GOT está resuelta, se salta a la función
  - Cuando la entrada de la GOT no está resuelta (estado inicial), el trampolín llama al dynamic loader para resolverla

ELF



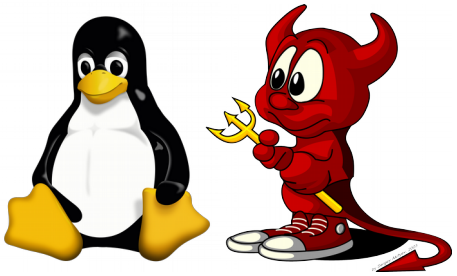
```
int main(void) {
    int ret = -1;
    pthread_t thread_info;
    pthread_attr_t thread_attr;

    printf("Main thread PID: %d\n", getpid());
    printf("Main thread TID: %d\n", syscall(__NR_gettid));
```

```
0000000000400a59:   push   %rbp
0000000000400a5a:   mov    %rsp,%rbp
0000000000400a5d:   sub    $0x60,%rsp
 51                   int ret = -1;
0000000000400a61:   movl   $0xffffffff,-0x4(%rbp)
 55                   printf("Main thread PID: %d\n", getpid
0000000000400a68:   callq  0x4007e0 <getpid@plt>
```

“main” ELF (x86_64) – llamada a getpid de ejemplo a través de PLT

Segmento PLT

ELF



```
00000000004007ac:   nopl    0x0(%rax)
pthread_create@plt:
00000000004007b0:   jmpq    *0x201862(%rip)        # 0x602018
00000000004007b6:   pushq   $0x0
00000000004007bb:   jmpq    0x4007a0
puts@plt:
00000000004007c0:   jmpq    *0x20185a(%rip)        # 0x602020
00000000004007c6:   pushq   $0x1
00000000004007cb:   jmpq    0x4007a0
sigaction@plt:
00000000004007d0:   jmpq    *0x201852(%rip)        # 0x602028
00000000004007d6:   pushq   $0x2
00000000004007db:   jmpq    0x4007a0
getpid@plt:
00000000004007e0:   jmpq    *0x20184a(%rip)        # 0x602030
00000000004007e6:   pushq   $0x3
00000000004007eb:   jmpq    0x4007a0
printf@plt:
```

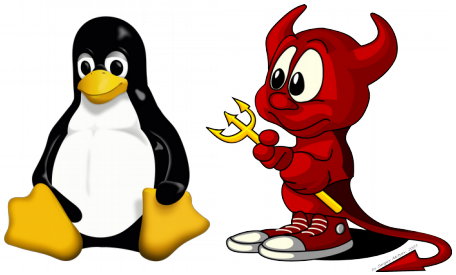
Número de la entrada de relocalización

ir al dynamic loader

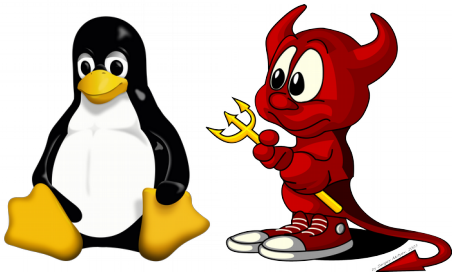
Tomar dirección de entrada en GOT PLT

```
Relocation section '.rela.plt' at offset 0x668 contains 12 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000602018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 pthread_create@GLIBC_2.2.5 +
000000602020  000300000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000602028  000400000007 R_X86_64_JUMP_SLO 0000000000000000 sigaction@GLIBC_2.2.5 + 0
000000602030  000500000007 R_X86_64_JUMP_SLO 0000000000000000 getpid@GLIBC_2.2.5 + 0
000000602038  000600000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000602040  000800000007 R_X86_64_JUMP_SLO 0000000000000000 pthread_attr_init@GLIBC_2.2.5
000000602048  000900000007 R_X86_64_JUMP_SLO 0000000000000000 syscall@GLIBC_2.2.5 + 0
000000602050  000a00000007 R_X86_64_JUMP_SLO 0000000000000000 sigemptyset@GLIBC_2.2.5 + 0
```

Dynamic Loader

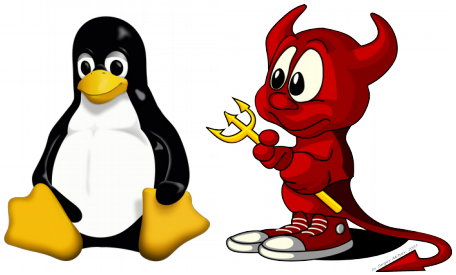


- Dynamic loader
  - Como intérprete, ejecuta en primer lugar
    - Luego transfiere el control al entry point del binario
  - Carga librerías con las que el binario linkea dinámicamente
  - Resuelve símbolos
  - Carga librerías durante tiempo de ejecución (dlopen)
  - Llena en runtime tabla GOT (usa información de relocalización del binario para saber dónde escribir la dirección de memoria una vez que resolvió el símbolo)



Demo 1.1

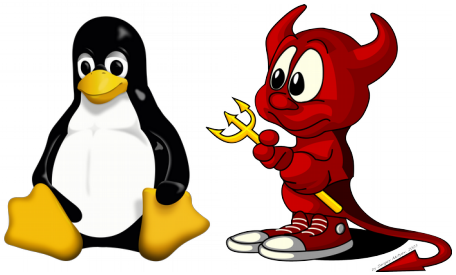
Kernel interpretando formato ELF para un binario ejecutable (sys_execve)



Demo 1.2

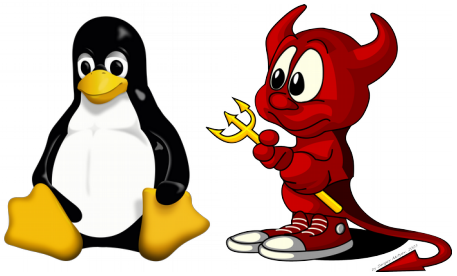
Llamada por GOT + PLT (dynamic linker)

ELF



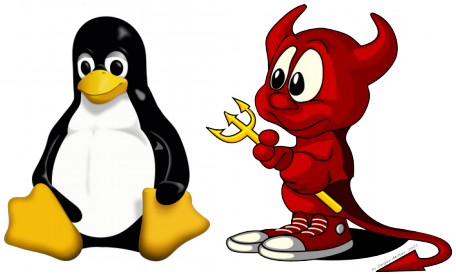
- `$ gcc -o main -g main.c`
- `$ echo $?`
- `$ readelf -a ./main`
- `$ objdump -d ./main | grep -A 500 -e "<_start>"`
- `$ gdb main`
  - `(gdb) break main`
  - `(gdb) run`
  - `(gdb) x/10xb 0x400000`
- `$ cat /proc/<PID>/maps`

Lab 1.1



- Patchear “main” para:
  - Retornar 5 (en lugar de 0)
  - quedar en un loop inifinito
  - Imprimir “ʊ” en *stdout*
- Debuggear con *gdb* cada uno de los casos
- Patchear en runtime con gdb cada uno de los casos anteriores

Lab 1.2



- Debuggear al dynamic loader parseando ELF al cargar una librería compartida
- Debuggear al kernel parseando ELF al cargar un módulo (.ko)

Referencias

- https://refspecs.linuxfoundation.org
- https://sourceware.org/git/?p=glibc.git;a=blob_plain;f=elf/elf.h;hb=HEAD
- https://linux.die.net/man/5/elf