

Lecture 6

Object File Formats reading ELF byte by byte

Md Mahbub E Noor

Assistant Professor, Department of CSE
University of Barishal

```
● ● ● od -A x -t x1z -N 16 main.o (real)
$ od -A x -t x1z -N 16 main.o
000000 7f 45 4c 46 02 01 01 00
          00 00 00 00 00 00 00 00
          >.ELF.....<
$ ./tiny; echo $?
42
# a whole program in 132 bytes
```

What you will learn today



Name the parts of an ELF file and the job of each one.



Read the ELF header, sections, symbols and relocations with readelf, and by hand.



Explain the linker's view (sections) and the loader's view (segments).



Read ELF files in C with the structs from `<elf.h>`.

Today's plan (60 min)

0-5

Recap of 5.2

5-12

One format for all

12-22

The ELF header

22-40

Sections, symbols, relocations

40-55

Segments, a 132-byte ELF

55-60

Checkpoint, homework

Lab 6: write elfread.c, a small readelf.

Quick recap of Lecture 5.2

Q1

What does INTERP in an executable say?

Which dynamic linker to start: `/lib64/ld-linux-x86-64.so.2`. Today: where that line lives in the file.

Q2

Where does ld.so find the names it must bind?

In the dynamic symbol tables (`nm -D`) of all loaded files. Today: the symbol table format itself.

Q3

What is a JUMP_SLOT relocation?

A note "fill this GOT slot with the address of counta". Today: the 24-byte struct that holds it.

PART 1

One format for every kind of program file

Relocatable, executable, shared object: all ELF.

Four files, one format (real output of file and readelf)

```
file main.o count count_pie libcount.so
```

```
main.o:      ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
count:       ELF 64-bit LSB executable, x86-64, ... dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2 ...
count_pie:   ELF 64-bit LSB pie executable, x86-64, ... dynamically linked ...
libcount.so: ELF 64-bit LSB shared object, x86-64, ... dynamically linked, not stripped
```

File	Made by	e_type	Sections?	Segments?	Size
main.o	gcc -c	REL (1)	yes, 14	no	1 656
count	gcc -no-pie	EXEC (2)	yes, 31	yes, 13	16 064
count_pie	gcc (default)	DYN (3)	yes, 31	yes, 13	16 160
libcount.so	gcc -shared	DYN (3)	yes, 25	yes, 11	15 248
tiny	nasm, by hand	EXEC (2)	none	yes, 1	132

A .o file has no segments: it cannot be run, only linked. tiny has no sections: it can be run, but not linked. Two views of one format (next slide).

Book: CS:APP §7.3–7.4; man 5 elf

You already know what is inside: SIC records → ELF

SIC object program	ELF file	Where to see it
H: name, start, length	ELF header: type, machine, sizes	<code>readelf -h</code>
T: bytes and their address	section contents: <code>.text</code> , <code>.data</code> , <code>.rodata</code>	<code>objdump -d</code> , <code>readelf -x</code>
(RESB / RESW, no T record)	<code>.bss</code> : size only, no bytes in the file	<code>readelf -S (NOBITS)</code>
D: names this file exports	symbol table, GLOBAL defined symbols	<code>readelf -s</code> , <code>nm (T, D, B)</code>
R: names this file needs	symbol table, UND symbols	<code>nm (U)</code>
M: fix this field	relocation entries: <code>.rela.text</code>	<code>readelf -r</code>
E: first instruction	<code>e_entry</code> in the ELF header	<code>readelf -h</code>

SAME AS C

The same information

Everything Beck's loader needed is in an ELF file. Only the packing is different.

DIFFERENT FROM C

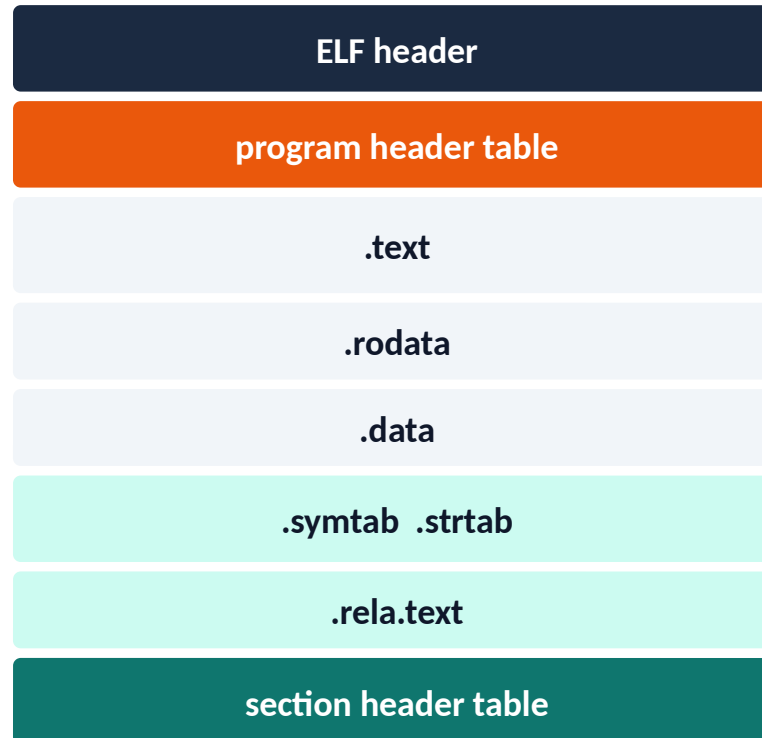
But binary, and much richer

Numbers are stored in binary C structs, not hex text; and there is more: debug info, permissions, the dynamic linker's tables.

Two views of one file: linker and loader

Loader's view: segments

The **program header table** lists a few big pieces: "map file bytes X-Y at address A with permission r-x". The kernel reads only this. A .o file has none.



Linker's view: sections

The **section header table** (at the end) lists many small named pieces: code, data, symbols, relocations. `ld`, `readelf` and `gdb` read it. The kernel ignores it.

C link: like one array seen two ways: as `char text[16064]` (bytes) or as an array of structs. Same bytes, two tables of contents for two different readers.

Book: CS:APP §7.4 (relocatable object files), §7.9 (loading executables)

PART 2

The ELF header: the first 64 bytes

A C struct written to disk.

The first 64 bytes of main.o, and the C struct they fill

```
od -A x -t x1 -N 64 main.o (real)
```

```
000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
000010 01 00 3e 00 01 00 00 00 00 00 00 00 00 00 00
000020 00 00 00 00 00 00 00 00 00 f8 02 00 00 00 00 00
000030 00 00 00 00 40 00 00 00 00 00 40 00 0e 00 0d 00
```

7f 45 4c 46

the magic: 0x7f then "ELF". Every ELF file starts like this.

02 01 01

64-bit, little-endian, ELF version 1.

01 00 at 16

e_type = 1 = REL. Little-endian: read 01 00 as 0x0001.

f8 02 ... at 40

e_shoff = 0x2f8 = 760: the section headers start at byte 760.

```
/usr/include/elf.h
```

```
typedef struct {
    unsigned char e_ident[16]; /* 0 */
    Elf64_Half e_type; /* 16 */
    Elf64_Half e_machine; /* 18 */
    Elf64_Word e_version; /* 20 */
    Elf64_Addr e_entry; /* 24 */
    Elf64_Off e_phoff; /* 32 */
    Elf64_Off e_shoff; /* 40 */
    Elf64_Word e_flags; /* 48 */
    Elf64_Half e_ehsize; /* 52 */
    Elf64_Half e_phentsize; /* 54 */
    Elf64_Half e_phnum; /* 56 */
    Elf64_Half e_shentsize; /* 58 */
    Elf64_Half e_shnum; /* 60 */
    Elf64_Half e_shstrndx; /* 62 */
} Elf64_Ehdr; /* 64 bytes */
```

Book: *man 5 elf*; `/usr/include/elf.h`

readelf -h: main.o next to the linked program

Field	main.o	count	What changed by linking
Type	REL	EXEC	now a program, not a piece of one
Entry point	0x0	0x401050	now there is a first instruction (_start)
Program headers	0	13, at byte 64	the loader's table was added
Section headers	14, at byte 760	31, at byte 14080	linker and libc start-up added sections
Header size	64	64	always 64 for 64-bit ELF
Section name table	section 13	section 30	where the section names are (.shstrtab)

● ● ● the struct sizes are fixed (sizes.c, real)

```
Elf64_Ehdr 64   Elf64_Shdr 64   Elf64_Phdr 56
Elf64_Sym  24   Elf64_Rela 24
```

Little-endian: 50 10 40 00 ... read backwards = 0x401050. The same byte order you saw in Lecture 1 with x86 integers.

Reading an ELF header in C: point a struct at the bytes

● ● ● from elfread.c (the lab program)

```
#include <elf.h>
unsigned char *buf = ...whole file...; /* fread */
if (memcmp(buf, ELFMAG, SELFMAG) != 0)
    die("not an ELF file");
Elf64_Ehdr *eh = (Elf64_Ehdr *)buf;
printf("Entry point: 0x%lx\n", eh->e_entry);
Elf64_Shdr *sh = (Elf64_Shdr *) (buf + eh->e_shoff);
/* sh[0] ... sh[eh->e_shnum - 1]: an ordinary array */
```

● ● ● ./elfread main.o (real, first lines)

```
Type:                REL (relocatable: a .o file)
Section headers:      14, at offset 760, 64 bytes each
```

SAME AS C

Same as your 1st-semester C

A struct, a pointer, an array, -> and [i]. An offset in the file becomes buf + offset. Nothing new in the language.

DIFFERENT FROM C

Careful with the file

The bytes come from outside: a damaged file can hold an offset past the end. elfread checks every offset before using it, or it would crash.

PART 3

Sections, symbols and relocations

The linker's view: what the assembler leaves for ld.

The sections of main.o (readelf -S, real)

Nr	Name	Type	Offset	Size	Holds
1	.text	PROGBITS	0x40	0x36	the machine code of main (54 bytes)
2	.rela.text	RELA	0x1f0	0x78	5 relocations for .text (5×24 bytes)
3	.data	PROGBITS	0x76	0	initialised globals: none in main.c
4	.bss	NOBITS	0x76	0	zero globals: size only, never bytes
5	.rodata	PROGBITS	0x76	0xc	"total = %d\n" (12 bytes with \0)
11	.symtab	SYMTAB	0x108	0xc0	8 symbols (8×24 bytes)
12	.strtab	STRTAB	0x1c8	0x21	the symbol names
13	.shstrtab	STRTAB	0x280	0x74	the section names

Check the sizes: $0x78 = 120 = 5 \times 24$ (sizeof Elf64_Rela); $0xc0 = 192 = 8 \times 24$ (sizeof Elf64_Sym). Tables in ELF are just arrays of structs.

Where does each line of C go? (sections.c)

C line	Section	nm	Bind
int counter = 5;	.data	D	GLOBAL
int buffer[1000];	.bss	B	GLOBAL
const char title[] = "BANANA";	.rodata	R	GLOBAL
static int hidden = 7;	.data	d	LOCAL
static int helper(int x) {...}	.text	t	LOCAL
int count_calls(void) {...}	.text	T	GLOBAL
static int calls; (in a function)	.bss	b calls.0	LOCAL
int local = helper(1);	(stack)	—	not in the file

Lower-case = LOCAL

nm prints t, d, b for static names: other files cannot see them.

`static` has two jobs in C

On a global it means "this file only" (LOCAL). On a local variable it means "keep the value": moved to .bss, renamed calls.0.

Locals are not in the file

They are created on the stack at run time (Lecture 11).

Predict before you look: students guess each row, then run `nm sections.o`.

.bss costs nothing in the file

```
● ● ● int buffer[1000]; (real)
$ size sections.o
   text  data   bss  filename
    230     8  4004  sections.o
$ ls -l sections.o
1936 bytes
```

```
● ● ● int buffer[1000] = {1}; (real)
$ size sections_init.o
   text  data   bss  filename
    230  4036     4  sections_init.o
$ ls -l sections_init.o
5976 bytes
```

One value changed, the file grew by 4040 bytes. A zero array needs only its size (NOBITS); an array with even one non-zero value must store all 4000 bytes in .data.

SAME AS C

Like SIC's RESW

No T record, only space reserved.

DIFFERENT FROM C

But zero is guaranteed

C promises globals start at 0; the loader zero-fills .bss.

Names are stored once: string tables

```
readelf -x .strtab main.o (real)
0x00000000 006d6169 6e2e6300 6d61696e 00636f75 .main.c.main.cou
0x00000010 6e746100 746f7461 6c007072 696e7466 nta.total.printf
```

Offset	0	1	8	d	14	1a
String	" "	main.c	main	counta	total	printf

A symbol does not hold its name: it holds **st_name**, an offset into .strtab. The name runs until the next 00 byte: a C string.

Clever detail (real): .shstrtab has no separate ".text". The section .text has sh_name = 0x20, which points into the middle of ".rela.text" (stored at 0x1b): $0x1b + 5 = 0x20$. Shared tails save bytes.

SAME AS C

Same as C strings

`char *name = strtab + sym->st_name;`
gives a normal C string, ending at `'\0'`.
`printf("%s", name)` just works.

DIFFERENT FROM C

Why not store names in the struct?

Names have different lengths; a struct must have a fixed size (24 bytes for a symbol). So the struct stores a number, and the text lives elsewhere.

The symbol table: D and R records in one list

```
readelf -s main.o ; readelf -s data.o (real, shortened)
```

Num:	Value	Size	Type	Bind	Ndx	Name
4:	0000	54	FUNC	GLOBAL	1	main
5:	0000	0	NOTYPE	GLOBAL	UND	counta
6:	0000	0	NOTYPE	GLOBAL	UND	total
7:	0000	0	NOTYPE	GLOBAL	UND	printf
--- data.o						
2:	0000	7	OBJECT	GLOBAL	2	text
3:	0008	4	OBJECT	GLOBAL	2	len

Ndx = a section number

main lives in section 1 (.text) at offset 0: a D record.

Ndx = UND

Defined elsewhere: an R record. Value 0, nobody knows it yet.

```
struct Elf64_Sym (24 bytes)
```

```
Elf64_Word  st_name; /* offset in .strtab */
unsigned char st_info; /* type (low 4) + bind (high 4) */
unsigned char st_other; /* visibility */
Elf64_Half  st_shndx; /* section number, 0 = UND */
Elf64_Addr  st_value; /* offset in that section */
Elf64_Xword st_size; /* bytes */
```

Size and Type

SIC had neither. ELF knows main is a 54-byte function and text a 7-byte object: gdb and the linker use this.

C's static keyword is a linker instruction

● ● ● a1.c and a2.c, both with static helper (real)

```
/* a1.c */ static int helper(void) { return 1; }  
/* a2.c */ static int helper(void) { return 2; }  
$ gcc -o st a1.c a2.c && ./st; echo $?  
3  
$ nm a1.o  
0000000000000000 t helper
```

● ● ● the same without static (real)

```
$ gcc -o st2 b1.c b2.c  
multiple definition of `helper';  
b1.c: first defined here  
$ nm b1.o  
0000000000000000 T helper
```

What really happens: `static` makes the symbol **LOCAL** (lower-case t). LOCAL symbols never enter the linker's global table, so two files can each have their own helper. Without static both are GLOBAL: the duplicate-symbol error of Lecture 5.1.

SAME AS C

Same idea in SIC

A label without EXTDEF stays inside its control section.

DIFFERENT FROM C

Opposite default

SIC hides names unless EXTDEF; C exports them unless static.

Relocations: the M records of ELF

```
● ● ● readelf -r main.o (real, first 3 of 5)
```

Offset	Info	Type	Sym + Addend
0009	0000000500000004	R_X86_64_PLT32	counta - 4
000f	0000000600000002	R_X86_64_PC32	total - 4
0015	0000000600000002	R_X86_64_PC32	total - 4

```
● ● ● struct Elf64_Rela (24 bytes)
```

```
Elf64_Addr  r_offset; /* where to patch          */  
Elf64_Xword r_info;  /* symbol << 32 | type */  
Elf64_Sxword r_addend; /* a number to add      */
```

1

r_info 0x0000000500000004: symbol 5 (counta), type 4 (PLT32)

2

Patch the 4 bytes at offset 9: the hole in e8 00 00 00 00

3

Value = **S + A - P**: symbol + addend - place

The numbers from Lecture 5.1: S = counta = 0x40116c, A = -4, P = 0x401136 + 9 = 0x40113f. 0x40116c - 4 - 0x40113f = **0x29**: exactly the e8 29 00 00 00 ld wrote. A relocation is an M record that also says **how** to compute the value.

Book: CS:APP §7.7 (relocation)

PART 4

Segments: what the kernel loads

Program headers, the entry point, and a program with no sections at all.

Program headers of count: four LOAD segments

Type	Offset	VirtAddr	FileSiz	MemSiz	Flags	Sections inside
INTERP	0x318	0x400318	0x1c	0x1c	R	.interp: "/lib64/ld-linux-x86-64.so.2"
LOAD	0x0	0x400000	0x500	0x500	R	headers, .dynsym, .rela.plt, ...
LOAD	0x1000	0x401000	0x1bd	0x1bd	R E	.init .plt .plt.sec .text .fini
LOAD	0x2000	0x402000	0x114	0x114	R	.rodata .eh_frame
LOAD	0x2df8	0x403df8	0x22c	0x238	RW	.dynamic .got .got.plt .data .bss
DYNAMIC	0x2e08	0x403e08	0x1d0	0x1d0	RW	.dynamic: the table for ld.so

Grouped by permission

Many sections, few segments: all code together (R E), all writable data together (RW). The kernel maps each LOAD with mmap.

MemSiz > FileSiz

$0x238 - 0x22c = 12$ bytes of .bss: no bytes in the file, zero-filled in memory.

Offset $\equiv$ address (mod 0x1000)

0x2df8 and 0x403df8 end the same: pages of the file map onto pages of memory.

Book: CS:APP §7.8–7.9 (executable object files, loading)

e_entry points to _start, not to main

```
objdump -d count (real, excerpt)
```

```
000000000401050 <_start>:
```

```
401054: 31 ed      xor    %ebp,%ebp
40105d: 48 83 e4 f0 and    $0xfffff0,%rsp
```

```
...
```

```
401068: 48 c7 c7 36 11 40 00 mov    $0x401136,%rdi
40106f: ff 15 63 2f 00 00    call  *... <__libc_start_main>
401075: f4          hlt
```

Entry point 0x401050 = _start

A small piece of code from the C library (crt1.o), linked into every C program.

0x401136 = main

_start passes main's address to __libc_start_main, which sets up the C library, calls main, then calls exit with main's return value.

C link: who calls main? You always wrote `int main(void)` as if it were the start. It is the start of **your** code only. `return 0;` in main becomes `exit(0)` in `__libc_start_main`.

hlt is never reached

`__libc_start_main` never returns. If it did, `hlt` (a privileged instruction) would crash the program.

Which parts does a program need to run?

● ● ● strip removes .symtab and .strtab (real)

```
$ cp count count_strip && strip count_strip
$ ls -l count count_strip
16064 count      14496 count_strip
$ ./count_strip
total = 3
$ nm count_strip
nm: count_strip: no symbols
```

● ● ● but the dynamic symbols stay (real)

```
$ nm -D count_strip
w __gmon_start__
U __libc_start_main@GLIBC_2.34
U printf@GLIBC_2.2.5
$ readelf -d count_pie | head -3
(NEEDED) Shared library: [libc.so.6]
```

Part	Who needs it	Can be removed?
program headers, LOAD segments	kernel	no
.dynsym, .dynamic, .rela.plt	ld.so (run-time linking)	no
.symtab, .strtab	ld, gdb, nm	yes: strip
.debug_* (gcc -g adds 6 of them)	gdb	yes: strip

The smallest program: an ELF file written by hand

tiny.asm (nasm -f bin), shortened

```
org 0x400000
ehdr: db 0x7F, "ELF", 2, 1, 1, 0
      times 8 db 0
      dw 2          ; e_type: EXEC
      dw 0x3E       ; e_machine: x86-64
      dd 1
      dq _start     ; e_entry
      dq phdr - $$   ; e_phoff = 64
      dq 0          ; e_shoff: no sections
      ... sizes, e_phnum = 1 ...
phdr: dd 1, 5        ; PT_LOAD, R+X
      dq 0, $$, $$   ; offset, vaddr, paddr
      dq filesize, filesize, 0x1000
_start: mov edi, 42  ; exit status
        mov eax, 60  ; 60 = exit
        syscall
```

build and run (real)

```
$ nasm -f bin tiny.asm -o tiny
$ chmod +x tiny && ./tiny; echo $?
42
$ ls -l tiny
132 bytes
$ strace ./tiny
execve("./tiny", ...) = 0
exit(42)
```

What this proves

An executable needs only an **ELF header** (64 bytes), one **program header** (56) and the **code** (12). $64 + 56 + 12 = 132$. No sections, no symbols, no libc, no main: the kernel does not need them.

Watch the kernel read tiny, byte by byte

How Linux reads a 132-byte program

ELF HEADERPROGRAM HEADERLOADRUN

the file tiny (od -A x -t x1)

0000	7f	45	4c	46	02	01	01	00	00	00	00	00	00	00	00
0010	02	00	3e	00	01	00	00	00	78	00	40	00	00	00	00
0020	40	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0030	00	00	00	00	40	00	38	00	01	00	00	00	00	00	00
0040	01	00	00	00	05	00	00	00	00	00	00	00	00	00	00
0050	00	00	40	00	00	00	00	00	00	00	40	00	00	00	00
0060	84	00	00	00	00	00	00	00	84	00	00	00	00	00	00
0070	00	10	00	00	00	00	00	00	bf	2a	00	00	00	b8	3c
0080	00	00	0f	05											

ELF header 0-63

program header 64-119

code 120-131

struct Elf64_Phdr (56 bytes)

p_type1 = PT_LOAD

p_flags5 = R + X

p_offset

p_vaddr

p_paddr

p_filesz

p_memsz

p_align

$p_flags = 5 = 4 \text{ (read)} + 1 \text{ (execute)}$. No 2 (write): the code cannot change itself.

Watch for

1. Every number is read **backwards** (little-endian).
2. **e_phoff** sends the kernel to byte 64.
3. **e_shoff = 0**: sections are never needed.
4. **PT_LOAD** maps the file at 0x400000, r-x.
5. Three instructions end in **exit(42)**.

Click the video to play or pause. About 2 minutes, no sound.

When the bytes are wrong (real errors)

1. Change one byte of the magic

```
bash
$ cp count bad
$ printf 'X' | dd of=bad bs=1
seek=1 \
$ conv=notrunc
$ ./bad
cannot execute binary file:
Exec format error
$ readelf -h bad
Not an ELF file - it has the
wrong magic bytes at the start
```

7f 58 4c 46 is not ELF. The kernel checks 4 bytes and stops.

2. Run a .o file

```
bash
$ chmod +x main.o
$ ./main.o
cannot execute binary file:
Exec format error
```

A real ELF file, but REL with no program headers: there is nothing the kernel could load.

3. Wrong machine (homework)

```
bash
$ # tiny with e_machine = 0xB7
$ ./t_arm
cannot execute binary file:
Exec format error
```

0xB7 = 183 = 64-bit ARM. Right format, wrong CPU.

Other systems, same ideas

Format	Used by	Typical files	Same parts as ELF?
ELF	Linux, BSD, Android, most embedded systems	.o (no ext.) .so core	—
PE / COFF	Windows	.obj .exe .dll	header, sections, symbols, relocations, import table (like the GOT)
Mach-O	macOS, iOS	.o (no ext.) .dylib	header, load commands (like program headers), segments with sections
a.out	early Unix	a.out	a tiny header, text, data, symbols, relocations: the name gcc still uses

Why a.out? When you run `gcc hello.c` without `-o`, the output is still called `a.out`: the name of the old Unix format, kept for tradition. Today that file is ELF (check it with `file a.out`). Whatever the format, you now know what to look for: a header, code and data, symbols, relocations, and a list of what to load where.

LAB 6

Write a small readelf in C

elfread.c: about 200 lines, five TODOs, checked against the real readelf.

Lab 6 tasks, step by step

1

TODO 1: the magic

`memcmp(buf, ELF_MAGIC, SELF_MAGIC)`. Test: `./elfread elfread.c` must say "not an ELF file".

2

TODO 2: sections

Point an `Elf64_Shdr` array at `e_shoff`; names from `.shstrtab`. Compare with `readelf -S main.o`.

3

TODO 3: symbols

Use `sh_link` to find the names; `Elf64_ST_TYPE` and `Elf64_ST_BIND`. Compare with `readelf -s`.

4

TODO 4: relocations

Split `r_info` with `Elf64_R_SYM` and `Elf64_R_TYPE`. Check `counta - 4` at offset 9.

5

TODO 5: program headers

Same pattern as TODO 2 with `Elf64_Phdr`. Try it on `count` and on `tiny`.

6

Check everything

`./elfread main.o | diff - expected_main.o.txt` (and `sections.o`, `tiny`): no output means done.

Submit: elfread.c and its output for count. Extra: run it on a damaged file (bad) and on /bin/ls.

Checkpoint: test yourself

Q1 The first four bytes of every ELF file?

→ 7f 45 4c 46 (0x7f 'E' 'L' 'F')

Q2 Which table does the kernel read: sections or program headers?

→ program headers (segments)

Q3 `static int n = 3;` at file scope: section and binding?

→ .data, LOCAL (nm: d)

Q4 `r_info = 0x0000000700000004`: symbol and type?

→ symbol 7 (printf in main.o), type 4 = PLT32

Q5 Why does a 4000-byte zero array not make the .o file bigger?

→ it is in .bss (NOBITS): only its size is stored

Key ideas from today



.o files, executables and .so files are all ELF: header, sections and/or segments.



The ELF header is a 64-byte C struct; every table after it is an array of structs.



Sections are for the linker; segments (program headers) are for the kernel.



.symtab = D and R records; .rela = M records with a formula: $S + A - P$.



C's static makes a symbol LOCAL; .bss stores only a size.



A running program needs only a header, a program header and code: 132 bytes.

Homework 6 (due before Lecture 7.1)

1. Decode a header

From `od -A x -t x1 -N 64 count` (lab files) find by hand: type, entry point, `e_phoff`, `e_shoff`, `e_phnum`, `e_shnum`. Then check with `readelf -h`.

2. Predict

For each global and static name in `sections.c`, predict section, nm letter and binding before running `nm`. Explain calls.0.

3. Relocate by hand

`main.o`'s relocation at offset 0xf: total – 4, type PC32. `main` is at 0x401136, total at 0x404028. Compute the 4 bytes `ld` writes, then find them in `objdump -d count`.

4. Break tiny

Make three copies of `tiny.asm`: exit status 7; `p_flags` = 4; `e_machine` = 0xB7. Predict, run, and explain each result.

5. Extend elfread

Add an option `-x NAME` that prints a section in hex like `readelf -x`. Test it on `.rodata` and `.strtab` of `main.o`.

Next lecture and reading

NEXT: LECTURE 7.1

Part B begins: the Linux kernel, an overview

Part A followed a program from source to an ELF file. Part B follows it into a running system: what the kernel is made of (7.1), then how the machine boots and starts the first process (7.2).

Main text

R. E. Bryant & D. R. O'Hallaron, **CS:APP**, 3rd ed., §7.4–7.9 (ELF, symbols, relocation, loading).

Reference

`man 5 elf` and `/usr/include/elf.h` on the lab PC: the structs from today, offline.

Beck

L. L. Beck, **System Software**, 3rd ed., §3.5 (loader examples) for comparison.

Lab files

`lab6_files.zip`: starter, demos, `tiny.asm`, `run_all6.sh`, expected outputs.