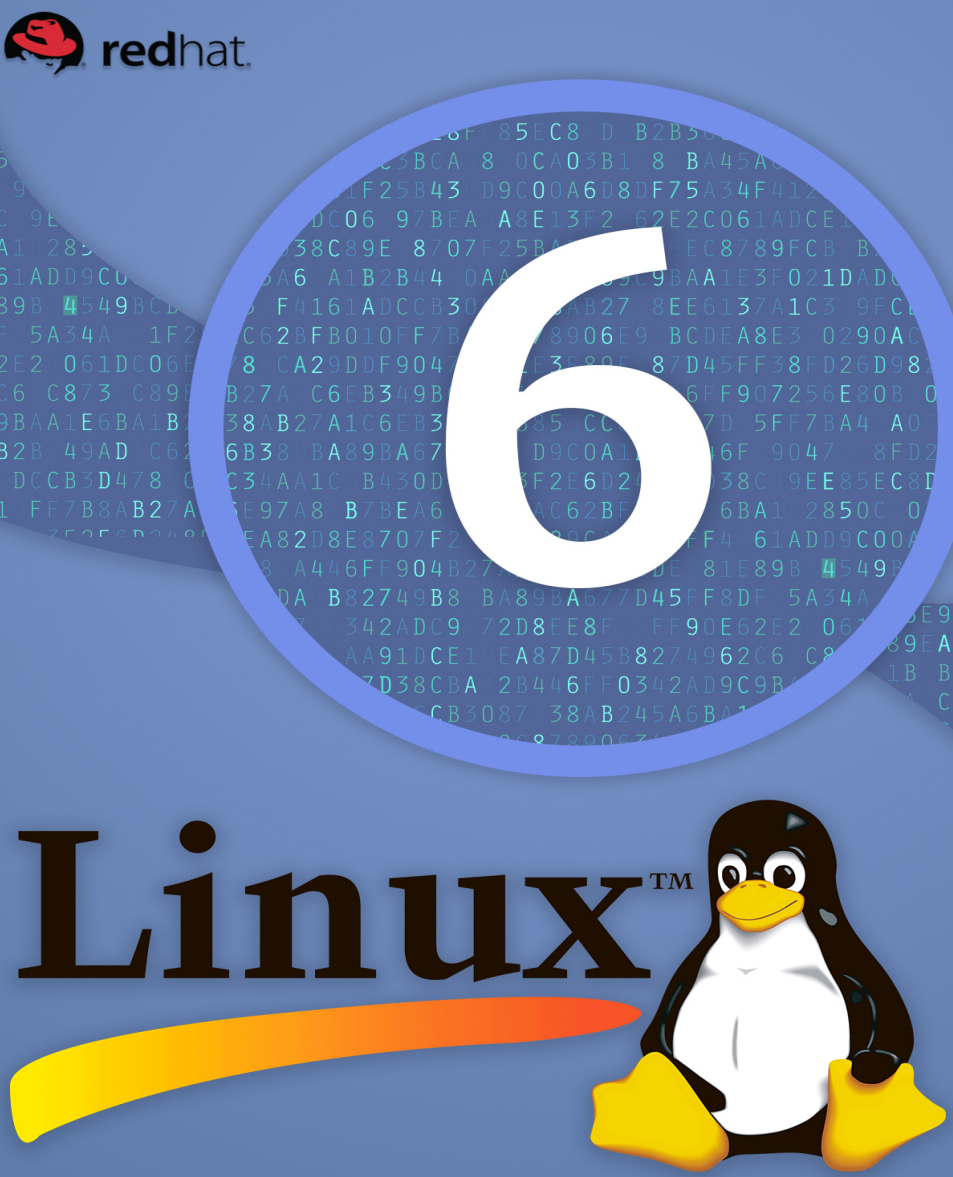
Low-Level Programming on Linux (x86-64)

ELF and Binary Inspection

redhat

debian

6



Prepared by Ayman Alheraki

https://simplifycpp.org

Low-Level Programming on Linux (x86-64) :

ELF and Binary Inspection

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author's Introduction

Executable and Linking Format (ELF) files form the structural foundation of every program executed on a modern Linux system. Everything that ultimately reaches the CPU—executable code, symbol tables, relocation records, dynamic loader metadata, memory segment layouts, and runtime descriptors—originates as a precisely defined arrangement of bytes inside an ELF container. Despite this central role, most developers interact with ELF only indirectly, through linkers, loaders, debuggers, and diagnostic utilities, rarely examining the format directly or verifying its contents at the binary level.

This book was written to remove that abstraction layer entirely.

The objective of ELF and Binary Inspection is to teach the reader how to interpret executable files with the same precision and rigor used by the Linux kernel loader, the dynamic linker, and low-level system tools. Rather than relying on external utilities or library-assisted parsing, this volume treats the ELF file strictly as a binary structure governed by the System V ABI. Every field, offset, size, and boundary is read explicitly, validated manually, and interpreted deterministically. There is no automated decoding, no opaque helper code, and no reliance on a standard library.

By emphasizing Intel-syntax x86-64 assembly and raw Linux syscalls, the reader learns to:

- read executable headers without libc support,

- traverse program and section tables using explicit pointer arithmetic,
- decode string tables and symbol records at the byte level,
- validate offsets and sizes against actual file boundaries,
- construct minimal, deterministic tools for binary inspection,
- and understand ELF as a structured byte stream rather than abstract metadata.

Each chapter introduces a progressively deeper view of the ELF architecture. The discussion begins with header structures and expands to cover program headers, section headers, string tables, symbol tables, relocation records, and dynamic linking metadata. The volume culminates in the construction of complete inspection utilities—including a hex viewer, section information printer, and a fully standalone ELF section viewer—implemented entirely in assembly and backed solely by the Linux syscall ABI.

This deliberate minimalism serves a clear purpose: to enforce absolute clarity about how binaries are structured, validated, and executed.

True understanding of binary formats does not come from reading tool output; it emerges from reconstructing those tools from first principles. By reaching the point where an ELF executable can be parsed, validated, and inspected using only a few hundred lines of assembly, the reader gains the ability to reason about compiled programs at the same level as the system components responsible for loading and executing them.

This volume builds upon foundations established in earlier books in the series—covering computer architecture, assembly language, system calls, and memory management—and forms a critical link in the broader study of low-level programming. Mastery of ELF is essential for debugging, reverse engineering, toolchain development, security analysis, and the construction of minimal or specialized runtimes. It is also fundamental to

understanding how code transitions from static storage on disk to active execution in memory.

The aim of this book is not to reproduce the ELF specification as a reference document, but to guide the reader in writing real, executable, low-level tools that dissect and validate ELF binaries with the same strictness enforced by the Linux kernel itself. Through this approach, the format ceases to be theoretical and becomes tangible: a sequence of bytes that can be read, verified, and understood without dependence on any library or abstraction layer.

By the end of this volume, the reader will possess the practical expertise required to inspect ELF binaries at the most fundamental level—fully aware of every field, offset, invariant, and constraint that governs executable code on modern Linux systems.

Stay Connected

For more discussions and valuable content about ELF and Binary Inspection, I invite you to follow me on LinkedIn:

https://linkedin.com/in/aymanalheraki

You can also visit my personal website:

https://simplifycpp.org

Ayman Alheraki

Chapter 1

File Descriptors

1.1 open / close Basics

Linux exposes a minimal and uniform interface for file access based on the concept of file descriptors. At the lowest level, a file descriptor is a small, non-negative integer returned by the kernel to represent an open file, directory, socket, pipe, device, or any object implementing the Virtual File System (VFS) interface.
While high-level libraries abstract these mechanisms, low-level ELF inspection, loader analysis, and handwritten runtimes require direct and explicit control over descriptor creation, usage, and destruction. Understanding the raw open and close system calls is fundamental to reasoning about executable behavior before and during runtime initialization.

1.1.1 The Role of File Descriptors in ELF Execution

Before an ELF binary executes its first instruction at _start, the Linux kernel establishes a minimal execution context.

This context includes, among other elements:

- Inheriting file descriptors whose FD_CLOEXEC flag is cleared.
- Establishing the standard descriptors:
  - 0 — standard input
  - 1 — standard output
  - 2 — standard error

From a binary inspection standpoint, understanding how file descriptors originate is critical when analyzing:

- dynamic loader activity
- /proc/<pid>/fd mappings
- relocation diagnostics
- file accesses performed before main() via constructors

Descriptor creation via openat is therefore among the earliest observable interactions between an ELF binary and the kernel.

1.1.2 openat — The Modern System Call

At the syscall layer, legacy open(2) no longer exists. Modern Linux kernels expose only:

- openat
- openat2

The C library still provides open() as a compatibility wrapper, but internally it translates to openat. For assembly-level ELF work, the syscall interface must be used directly.

```
SYS_openat = 257
SYS_close  = 3
```

The x86-64 System V ABI syscall calling convention is:

```
rdi = dirfd        (use -100 for AT_FDCWD)
rsi = pathname     (pointer)
rdx = flags
r10 = mode         (used only with O_CREAT / O_TMPFILE)
rax = syscall number
```

1.1.3 Minimal Assembly Example — Open and Close a File

The following example demonstrates:

- invoking openat directly
- bypassing libc entirely
- writing to stdout via write
- explicitly closing the descriptor
- exiting through _exit

All instructions use **GAS Intel syntax**.

Example: open ”sample.txt”, print a message, close it

```
.intel_syntax noprefix

.global _start

.section .data
filename:
    .asciz "sample.txt"
msg:
    .asciz "Opened file successfully\n"

.section .text

_start:
    # rdi = AT_FDCWD (-100)
    mov rdi, -100

    # rsi = pathname
    lea rsi, filename

    # rdx = O_WRONLY | O_CREAT | O_TRUNC
    # O_WRONLY = 1
    # O_CREAT  = 64
    # O_TRUNC  = 512
    mov rdx, 577

    # r10 = mode 0644
    mov r10, 0644

    # rax = SYS_openat
    mov rax, 257
    syscall

    # save returned file descriptor
```

```
    mov r12, rax

    # if rax < 0, jump to exit
    test rax, rax
    js exit_program

    # write message to stdout (fd = 1)
    mov rdi, 1
    lea rsi, msg
    mov rdx, 24
    mov rax, 1          # SYS_write
    syscall

    # close opened file
    mov rdi, r12
    mov rax, 3          # SYS_close
    syscall

exit_program:
    mov rdi, 0
    mov rax, 60         # SYS_exit
    syscall
```

This program demonstrates explicit descriptor management as required in manually linked ELF binaries.

1.1.4 Understanding Descriptor Lifetime

A file descriptor remains valid until one of the following occurs:

1. The process explicitly calls close(fd).
2. The process executes execve() and the descriptor has FD_CLOEXEC set.

3. The process terminates, at which point the kernel closes all remaining descriptors.

Descriptor leaks can result in:

- exhaustion of available descriptors
- unintended file locks
- incorrect inheritance across execution chains

In low-level ELF inspection, such leaks are common in:

- binary packers
- custom loaders
- minimal runtime stubs

When libc is bypassed, descriptor hygiene becomes the sole responsibility of the assembly author.

1.1.5 Why open/close Matter in Binary Inspection

A file descriptor indexes into the per-process descriptor table, where each entry refers to a kernel-resident struct file.
For binary inspection:

- /proc/<pid>/fd/ reveals live descriptor bindings.
- Tracing openat/close exposes dynamic loader behavior and configuration probing.
- Static analysis benefits from identifying file-related syscalls embedded directly in the .text section.

Precise control of these operations enables deterministic behavior in custom loaders and static binaries.

1.1.6 Summary

openat and close form the foundation of file interaction on Linux. In ELF analysis and binary inspection, they represent the earliest and most reliable interface between executable code and the kernel's VFS layer. Mastery of these calls is essential for understanding descriptor inheritance, runtime behavior, and resource control in minimal environments.

1.2 Buffered vs Raw Access

On Linux, every file descriptor represents a kernel-managed byte stream. User programs interact with this stream in two fundamentally different ways: buffered access via user-space libraries and raw access via direct system calls.
Buffered access improves convenience but obscures kernel-visible behavior. Raw access exposes exact syscall semantics and is therefore mandatory for ELF inspection and handwritten assembly.

1.2.1 Buffered Access: A User-Space Abstraction

Functions such as fopen(), fread(), and fwrite() operate on internal user-space buffers. These buffers introduce behaviors invisible to the kernel:

- read-ahead
- write coalescing
- delayed flushing

From the kernel's perspective:

- buffers do not exist

- syscalls occur only on flush or refill
- multiple writes may collapse into one syscall

Such behavior renders buffered I/O unsuitable for low-level analysis.

1.2.2 Raw Access: Direct System Calls

Raw access uses the kernel interface exactly as defined:

- read
- write
- pread
- pwrite

Each call is:

- synchronous
- directly observable
- governed solely by kernel semantics

For ELF analysis, raw syscalls are the only reliable unit of observation.

1.2.3 Minimal Raw Read Example (Assembly, No libc)

```
.intel_syntax noprefix

.global _start
```

```
.section .data
filename:
    .asciz ”input.bin”

.section .bss
buffer:
    .skip 256

.section .text

_start:
    # openat(”input.bin”, O_RDONLY)
    mov rdi, -100
    lea rsi, filename
    xor rdx, rdx
    xor r10, r10
    mov rax, 257
    syscall
    mov r12, rax

    test rax, rax
    js exit

    # read(fd, buffer, 256)
    mov rdi, r12
    lea rsi, buffer
    mov rdx, 256
    xor rax, rax
    syscall
    mov r13, rax

    # write to stdout
    mov rdi, 1
```

```
    lea rsi, buffer
    mov rdx, r13
    mov rax, 1
    syscall

    # close(fd)
    mov rdi, r12
    mov rax, 3
    syscall

exit:
    mov rdi, 0
    mov rax, 60
    syscall
```

Each syscall corresponds exactly to one kernel-visible operation.

1.2.4 Summary

Buffered I/O exists entirely in user space and obscures true execution behavior. Raw system calls expose the kernel's authoritative semantics and are indispensable for ELF inspection, loader analysis, and low-level debugging. For this booklet, all file interaction relies exclusively on raw syscalls to preserve correctness and determinism.

Chapter 2

Text and Binary I/O

2.1 Reading Lines

Line-oriented reading is a fundamental operation in many binary-inspection tools. Although ELF files are binary objects, inspection workflows frequently involve textual metadata: section lists, symbol dumps, path configuration files, /proc/<pid> interfaces, and diagnostic logs produced by loaders or debuggers. In low-level runtimes—where libc buffering is unavailable—implementing reliable line reading using raw syscalls is essential.

Linux provides no kernel-level concept of a "line." The kernel delivers only raw byte streams. Consequently, any line-oriented reader must implement its own buffering, delimiter detection, and partial-read handling. This section demonstrates a deterministic, libc-free line reader built solely on read(2) for Linux x86-64.

2.1.1 Line Reading at the Syscall Level

The kernel's read() syscall obeys strict rules:

- returns up to the requested number of bytes
- may return fewer bytes (partial read)
- returns 0 at end-of-file
- does not align reads to delimiters
- never interprets data

Textual sources such as pipes or /proc files rarely deliver a complete line in a single syscall. A correct implementation must:

1. repeatedly read into a buffer
2. scan for newline (0x0A)
3. extract completed lines
4. preserve remaining bytes
5. continue reading

Without libc, all of this logic must be written explicitly.

2.1.2 Minimal Low-Level Line Reader Design

A minimal libc-free line reader requires:

- a persistent buffer
- a cursor tracking consumed bytes
- a length counter for valid data

- explicit newline scanning
- logic for partial lines

This pattern is standard in early-boot utilities, static inspection tools, and ELF runtimes that operate before libc initialization.

2.1.3 Complete Assembly Implementation — Read and Print Lines

The following example demonstrates:

- raw read()
- manual newline detection
- per-line output
- no libc usage
- suitability for custom ELF inspectors

Assembly (GAS Intel Syntax, No libc)

```
.intel_syntax noprefix
.global _start

.section .data
filename:
    .asciz "lines.txt"

.section .bss
buffer:
    .skip 512
```

```
line:
    .skip 512

.section .text

# find_newline(rsi = buffer, rdx = length)
# returns:
#   rax = offset of '\n' or -1 if not found
find_newline:
    xor rax, rax
.scan:
    cmp rax, rdx
    je .none
    cmp byte ptr [rsi + rax], 0x0A
    je .found
    inc rax
    jmp .scan
.found:
    ret
.none:
    mov rax, -1
    ret

_start:
    # openat(”lines.txt”, O_RDONLY)
    mov rdi, -100
    lea rsi, filename
    xor rdx, rdx
    xor r10, r10
    mov rax, 257
    syscall
    mov r12, rax
```

```
    test r12, r12
    js exit

    xor r13, r13        # buffer offset
    xor r14, r14        # bytes in buffer

read_loop:
    cmp r14, 0
    jne process_buffer

    # read into buffer
    mov rdi, r12
    lea rsi, buffer
    mov rdx, 512
    xor rax, rax        # SYS_read
    syscall

    test rax, rax
    je exit

    mov r14, rax
    xor r13, r13

process_buffer:
    lea rsi, [buffer + r13]
    mov rdx, r14
    call find_newline

    cmp rax, -1
    je read_loop

    # copy line
    mov rcx, rax
```

```
    lea rsi, [buffer + r13]
    lea rdi, line

.copy:
    test rcx, rcx
    je .done_copy
    mov al, [rsi]
    mov [rdi], al
    inc rsi
    inc rdi
    dec rcx
    jmp .copy
.done_copy:
    mov byte ptr [rdi], 0x0A

    # write line
    mov rdi, 1
    lea rsi, line
    mov rdx, rax
    inc rdx
    mov rax, 1
    syscall

    add r13, rdx
    sub r14, rdx
    jmp read_loop

exit:
    mov rdi, 0
    mov rax, 60
    syscall
```

This implementation provides deterministic, syscall-visible behavior suitable for

minimal runtimes and ELF inspection environments.

2.1.4 Reading Arbitrary Bytes

Binary inspection fundamentally depends on reading raw bytes without interpretation. Unlike text, binary data has no delimiters, alignment markers, or inherent structure. ELF analysis involves reading:

- ELF headers
- program headers
- section tables
- relocation records
- symbol tables
- compressed debug data

At the syscall boundary, the kernel delivers bytes exactly as stored. The reader must respect partial reads, unexpected EOF, and alignment constraints.

2.1.5 Raw Binary Reading Semantics

The kernel guarantees:

- at most the requested byte count
- possible short reads
- zero only at EOF
- no structure awareness

Syscall boundaries never correspond to ELF structure boundaries. Binary parsers must never assume otherwise.

2.1.6 Robust Arbitrary-Size Binary Reader

A minimal binary reader must:

1. loop until the requested range is filled
2. tolerate partial reads
3. stop on EOF
4. avoid libc buffering

2.1.7 Assembly — read_exact

```
.intel_syntax noprefix
.global read_exact

# read_exact(fd=rdi, buf=rsi, count=rdx)
# returns rax = bytes read
read_exact:
    mov r8, rdx        # remaining
    mov r9, rdx        # original count

.loop:
    test r8, r8
    je .done

    xor rax, rax       # SYS_read
    mov rdx, r8
    syscall
```

```
    test rax, rax
    jle .done

    add rsi, rax
    sub r8, rax
    jmp .loop

.done:
    mov rax, r9
    sub rax, r8
    ret
```

This primitive is safe for all binary ELF structures.

2.1.8 Full Example — Reading an ELF Header

```
.intel_syntax noprefix
.global _start
.extern read_exact

.section .data
filename:
    .asciz "input.elf"
newline:
    .asciz "\n"
okmsg:
    .asciz "ELF64 OK\n"
errmsg:
    .asciz "Not ELF\n"

.section .bss
ehdr:
```

```
    .skip 64

.section .text

_start:
    # openat("input.elf", O_RDONLY)
    mov rdi, -100
    lea rsi, filename
    xor rdx, rdx
    xor r10, r10
    mov rax, 257
    syscall
    mov r12, rax
    test r12, r12
    js not_elf

    # read ELF header
    mov rdi, r12
    lea rsi, ehdr
    mov rdx, 64
    call read_exact
    cmp rax, 64
    jne not_elf

    # check magic
    mov eax, dword ptr [ehdr]
    cmp eax, 0x464C457F
    jne not_elf

    # check ELF64
    cmp byte ptr [ehdr + 4], 2
    jne not_elf
```

```
    # print OK
    mov rdi, 1
    lea rsi, okmsg
    mov rdx, 9
    mov rax, 1
    syscall
    jmp exit

not_elf:
    mov rdi, 1
    lea rsi, errmsg
    mov rdx, 8
    mov rax, 1
    syscall

exit:
    mov rdi, 0
    mov rax, 60
    syscall
```

2.1.9 ASCII vs Binary Interpretation

Linux exposes only byte streams. ASCII and binary interpretation exist solely in user space. ELF analysis must strictly separate these domains.

2.1.10 Low-Level Correctness Rules

1. Never apply null-terminated logic to binary data
2. Never assume ASCII unless explicitly defined
3. Always parse by offsets, not delimiters

4. Respect endianness

5. Treat 0x00 as a normal byte unless specified

2.1.11 Summary

Text and binary I/O are fundamentally different interpretations layered over identical kernel byte streams. For ELF inspection, raw binary reading is the primary mechanism, while line-oriented text parsing serves auxiliary roles. Correct low-level tooling depends on respecting this distinction and implementing deterministic syscall-level logic without reliance on libc abstractions.

Chapter 3

ELF Format Basics

3.1 ELF Header Structure

The Executable and Linkable Format (ELF) is the fundamental binary representation used by Linux systems. Every ELF object—whether an executable, shared library, relocatable object, or core dump—begins with a fixed-size header that defines the file's identity, layout, and interpretation rules.
At the lowest level, the ELF header is a binary structure with fixed offsets. It is not text-based and does not rely on delimiters or strings. Correct parsing requires strict adherence to byte offsets, field sizes, and endianness rules defined by the System V ABI for x86-64.

3.1.1 Binary View: No Strings, No Delimiters

Unlike textual formats, the ELF header is:

- fixed length: 64 bytes for ELF64

- little-endian on x86-64
- layout-driven, not delimiter-driven
- composed of integers and identifiers
- containing no textual semantics

Deterministic decoding is therefore possible using raw system calls and fixed-offset reads.

3.1.2 The ELF Magic (e_ident[0..3])

Every ELF file begins with the following four bytes:

```
7F 45 4C 46
```

These correspond to:

- 0x7F
- ASCII 'E'
- ASCII 'L'
- ASCII 'F'

Although printable, these bytes must be interpreted strictly as binary values. Treating them as strings or null-terminated text is incorrect and leads to broken parsing logic.

3.1.3 e_ident: Interpretation Control Block

The first 16 bytes of the ELF header (e_ident) define how all subsequent fields must be interpreted.
Key fields include:

- EI_CLASS — object class (ELF32 or ELF64)
- EI_DATA — byte order (little or big endian)
- EI_VERSION — ELF version
- EI_OSABI — operating system ABI
- EI_ABIVERSION — ABI version

All remaining decoding depends on these bytes. Any inconsistency requires the loader or inspector to reject the file immediately.

3.1.4 Core Structural Fields (Offsets >= 0x10)

Beginning at offset 0x10, the ELF header defines the structural layout of the file:

- e_type — file type
- e_machine — target architecture
- e_version — ELF version
- e_entry — entry point virtual address
- e_phoff — program header table offset
- e_shoff — section header table offset

- e_flags — architecture-specific flags
- e_ehsize — ELF header size
- e_phentsize, e_phnum
- e_shentsize, e_shnum
- e_shstrndx — section name string table index

Every field is positional and must be read exactly at its defined offset.

3.1.5 Minimal Assembly Reader: ELF Magic and Class Validation

The following example reads the ELF header and validates:

- ELF magic
- ELF64 class
- little-endian encoding

This is the minimum required logic for any ELF loader or inspector.

Assembly (GAS Intel Syntax, No libc)

```
.intel_syntax noprefix
.global _start

.section .data
filename:
    .asciz "input.elf"
okmsg:
    .asciz "ELF64 OK\n"
```

```
errmsg:
    .asciz "Not ELF\n"

.section .bss
ehdr:
    .skip 64

.section .text

# read_exact(fd=rdi, buf=rsi, count=rdx)
# returns rax = bytes read
read_exact:
    mov r8, rdx
    mov r9, rdx

.loop:
    test r8, r8
    je .done
    xor rax, rax            # SYS_read
    mov rdx, r8
    syscall
    test rax, rax
    jle .done
    add rsi, rax
    sub r8, rax
    jmp .loop

.done:
    mov rax, r9
    sub rax, r8
    ret
```

```
_start:
    # openat("input.elf", O_RDONLY)
    mov rdi, -100
    lea rsi, filename
    xor rdx, rdx
    xor r10, r10
    mov rax, 257
    syscall
    mov r12, rax
    test r12, r12
    js not_elf

    # read ELF header
    mov rdi, r12
    lea rsi, ehdr
    mov rdx, 64
    call read_exact
    cmp rax, 64
    jne not_elf

    # check magic
    mov eax, dword ptr [ehdr]
    cmp eax, 0x464C457F
    jne not_elf

    # check ELF64 class
    cmp byte ptr [ehdr + 4], 2
    jne not_elf

    # check little endian
    cmp byte ptr [ehdr + 5], 1
    jne not_elf
```

```
    # success
    mov rdi, 1
    lea rsi, okmsg
    mov rdx, 9
    mov rax, 1
    syscall
    jmp exit

not_elf:
    mov rdi, 1
    lea rsi, errmsg
    mov rdx, 8
    mov rax, 1
    syscall

exit:
    mov rdi, 0
    mov rax, 60
    syscall
```

This code demonstrates strict binary validation using fixed offsets and raw system calls.

3.1.6 Structural Significance in ELF Decoding

Once validated, the ELF header provides:

1. Precise Offsets

   The loader learns exactly where program headers and section headers reside in the file.

2. Execution Entry Point

   e_entry defines the virtual address where execution begins after mapping.

3. Machine and ABI Constraints

   Incorrect architecture, ABI, or endianness requires immediate rejection.

4. Format Integrity

   Malformed offsets or sizes indicate corruption or hostile input and must not be trusted.

3.1.7 Low-Level Parsing Requirements

1. No text-based logic
2. No null-terminated assumptions
3. Endianness-aware decoding
4. Exact offset usage
5. Early rejection of malformed headers

3.1.8 Summary

The ELF header defines the identity, architecture, layout, and execution rules of every ELF file. Low-level inspection requires treating it as a pure binary structure with fixed offsets and ABI-defined semantics. Correct parsing depends on raw syscalls, manual decoding, and strict validation.
This header is the gatekeeper for all deeper ELF structures, including program headers, sections, symbols, relocations, and memory mappings.

3.2 readelf -h Output Interpretation

The readelf -h command prints a human-readable representation of the ELF header. It does not infer information—it simply decodes fixed offsets and formats them as text. Understanding this output requires mapping each line back to its binary origin.

3.2.1 Mapping Textual Output to Binary Fields

Each field printed by readelf -h corresponds directly to a fixed offset in the ELF header. For example:

- Magic — bytes 0–3
- Class — byte 4
- Endianness — byte 5
- OSABI — byte 7
- Type — offset 0x10
- Machine — offset 0x12
- Entry — offset 0x18

A low-level inspector must replicate this logic manually using raw reads and offset decoding.

3.2.2 Low-Level Inspector Perspective

readelf is a formatter, not an authority. A correct inspector must:

- decode fields manually

- validate ABI constraints
- reject malformed headers
- avoid trusting textual output

3.2.3 Summary

Understanding ELF headers requires mapping human-readable tools back to their underlying binary structures. Loaders and inspectors must rely on raw decoding, not formatted output, to ensure correctness, security, and deterministic behavior.

3.3 Machine and ABI Indicators

Machine and ABI indicators define how the remainder of an ELF file must be interpreted. They determine architecture compatibility, relocation rules, calling conventions, and execution validity.
These fields appear both in e_ident and in the core header fields beginning at offset 0x10.

3.3.1 Key Indicators

- EI_CLASS — word size
- EI_DATA — endianness
- EI_OSABI — operating system ABI
- e_machine — target architecture
- e_type — file type

Any mismatch between these fields and the runtime environment requires rejection.

3.3.2 Minimal Assembly Validation Example

```
.intel_syntax noprefix
.global _start

.section .data
file:
    .asciz "input.elf"
okmsg:
    .asciz "MACHINE OK\n"
badmsg:
    .asciz "BAD ELF\n"

.section .bss
hdr:
    .skip 64

.section .text

read_exact:
    mov r8, rdx
    mov r9, rdx
.loop:
    test r8, r8
    je .done
    xor rax, rax
    mov rdx, r8
    syscall
    test rax, rax
    jle .done
    add rsi, rax
    sub r8, rax
    jmp .loop
```

```
.done:
    mov rax, r9
    sub rax, r8
    ret


_start:
    mov rdi, -100
    lea rsi, file
    xor rdx, rdx
    xor r10, r10
    mov rax, 257
    syscall
    mov r12, rax
    test r12, r12
    js bad

    mov rdi, r12
    lea rsi, hdr
    mov rdx, 64
    call read_exact
    cmp rax, 64
    jne bad

    cmp byte ptr [hdr + 4], 2
    jne bad

    cmp byte ptr [hdr + 5], 1
    jne bad

    mov ax, word ptr [hdr + 18]
    cmp ax, 0x3E
    jne bad
```

```
    mov rdi, 1
    lea rsi, okmsg
    mov rdx, 11
    mov rax, 1
    syscall
    jmp exit

bad:
    mov rdi, 1
    lea rsi, badmsg
    mov rdx, 8
    mov rax, 1
    syscall

exit:
    mov rdi, 0
    mov rax, 60
    syscall
```

3.3.3 Final Summary

Machine and ABI indicators form the foundation of ELF correctness. They define how the loader interprets instructions, relocations, addresses, and calling conventions. A low-level inspector must validate these fields before attempting any deeper parsing. Failure to do so leads to incorrect decoding, unsafe memory access, and unreliable analysis.

Chapter 4

Program and Section Headers

4.1 PT_LOAD and Memory Segments

In ELF64, the program header table defines how the Linux loader must transform a file on disk into a mapped image in memory. The most important type within this table is PT_LOAD, which describes how portions of the file become executable, writable, or read-only segments in virtual memory.
PT_LOAD entries are the backbone of ELF program loading: the loader maps these segments with mmap() (or equivalent low-level mechanisms), sets permissions based on flags, and prepares the memory image before transferring control to the entry point.
A correct low-level inspector must understand PT_LOAD entries as the foundation of the process memory image, especially when constructing minimal loaders, analyzing ELF binaries, or correlating mappings with /proc/<pid>/maps.

4.1.1 Role of PT_LOAD in Program Execution

A PT_LOAD segment describes:

- where bytes are located in the file,
- where those bytes must appear in memory,
- how much memory must be reserved (including zero-filled .bss),
- what permissions the memory must have,
- alignment constraints for the mapping.

Unlike section headers (which are optional at runtime), PT_LOAD segments are mandatory for execution. They define the actual memory footprint of the process image.

4.1.2 Essential PT_LOAD Fields

Each program header entry provides:

- p_type — must equal PT_LOAD (value 1)
- p_offset — file offset of segment bytes
- p_vaddr — virtual address where the segment maps
- p_paddr — ignored on Linux (historical ABI field)
- p_filesz — number of bytes present in the file
- p_memsz — total bytes reserved in memory ($\geq p\_filesz$)
- p_flags — PF_R / PF_W / PF_X permission mask
- p_align — alignment requirement (power of two)

A correct loader/inspector must validate that these values:

- do not exceed file bounds,
- do not overflow arithmetic,
- satisfy alignment invariants,
- do not create unsafe W^X mappings unless intentionally designed.

4.1.3 How the Linux Loader Maps PT_LOAD Segments

For each PT_LOAD entry, the runtime loader conceptually performs:

1. Determine mapping base:

   ```
   base = align_down(p_vaddr, page_size)
   ```

2. Determine file mapping offset:

   ```
   file_off = align_down(p_offset, page_size)
   ```

3. Compute page offset inside the mapping:

   ```
   page_delta = p_vaddr - base
   ```

4. Map file-backed bytes:

   ```
   map_len = page_delta + p_filesz
   ```

   with mmap(MAP_PRIVATE | MAP_FIXED) (or equivalent).

5. Zero-extend memory:

   ```
   bss_len = p_memsz - p_filesz
   ```

   The tail region becomes .bss (zero-filled).

6. Apply protections:
   Derived from p_flags:
   - PF_R
   - PF_W
   - PF_X

Mappings in /proc/<pid>/maps reflect these decisions.

4.1.4 Why PT_LOAD Matters for Binary Inspection

PT_LOAD defines the true runtime memory image:

- executable code region (.text and related)
- read-only constants (.rodata)
- writable data (.data)
- uninitialized data (.bss)

A professional inspector must verify:

- virtual ranges do not overlap incorrectly,
- segments do not exceed file bounds,
- permissions match expectations (code should not be writable),
- alignment is coherent and power-of-two,
- e_entry lies inside an executable PT_LOAD region.

Incorrect PT_LOAD interpretation causes:

- invalid mapping decisions,
- crashes before transfer to e_entry,
- vulnerabilities (W^X violations),
- incorrect disassembly boundaries.

4.1.5 Minimal Assembly Demo — Extract PT_LOAD Segments

The program below:

1. opens an ELF file
2. reads the ELF header
3. locates the program header table
4. reads and scans entries for PT_LOAD
5. prints p_vaddr and p_memsz as 16-hex-digit values

No libc. Pure syscalls. GAS Intel syntax.

Assembly Example (GAS Intel Syntax)

```
.intel_syntax noprefix
.global _start

.section .data
file:
    .asciz "input.elf"
newline:
    .asciz "\n"
```

```
.section .bss
ehdr:
    .skip 64
phdr_buf:
    .skip 4096
hexbuf:
    .skip 16

.section .text

# write_hex64(rdi=value)
# prints exactly 16 lowercase hex digits (no 0x prefix)
write_hex64:
    mov rcx, 16
    lea rsi, hexbuf
.loop:
    mov rax, rdi
    and rax, 0xF
    cmp al, 9
    jbe .num
    add al, 87          # 'a' - 10
    jmp .store
.num:
    add al, '0'
.store:
    mov byte ptr [rsi + rcx - 1], al
    shr rdi, 4
    dec rcx
    jnz .loop

    mov rdi, 1
    lea rsi, hexbuf
    mov rdx, 16
```

```
    mov rax, 1          # SYS_write
    syscall
    ret

# read_exact(fd=rdi, buf=rsi, count=rdx)
# returns rax = bytes read
read_exact:
    mov r8, rdx
    mov r9, rdx
.loop2:
    test r8, r8
    je .done2
    xor rax, rax        # SYS_read
    mov rdx, r8
    syscall
    test rax, rax
    jle .done2
    add rsi, rax
    sub r8, rax
    jmp .loop2
.done2:
    mov rax, r9
    sub rax, r8
    ret

_start:
    # openat("input.elf", O_RDONLY)
    mov rdi, -100
    lea rsi, file
    xor rdx, rdx
    xor r10, r10
    mov rax, 257
    syscall
```

```
mov r12, rax
test r12, r12
js exit

# read ELF header
mov rdi, r12
lea rsi, ehdr
mov rdx, 64
call read_exact
cmp rax, 64
jne exit

# e_phoff (offset 0x20), e_phentsize (0x36), e_phnum (0x38)
mov r13, qword ptr [ehdr + 32]   # e_phoff
movzx r14d, word ptr [ehdr + 54] # e_phentsize
movzx r15d, word ptr [ehdr + 56] # e_phnum

# validate e_phentsize for ELF64 program headers (expected 56)
cmp r14d, 56
jne exit

# compute total size = e_phentsize * e_phnum
mov eax, r14d
imul eax, r15d
mov rbx, rax                     # total size in rbx
cmp rbx, 4096
ja exit                          # buffer limit for this demo

# lseek(fd, e_phoff, SEEK_SET)
mov rdi, r12
mov rax, 8                       # SYS_lseek
mov rsi, r13
xor rdx, rdx                     # SEEK_SET
```

```
    syscall
    test rax, rax
    js exit

    # read program header table into phdr_buf
    mov rdi, r12
    lea rsi, phdr_buf
    mov rdx, rbx
    call read_exact
    cmp rax, rbx
    jne exit

    # iterate program headers
    lea rsi, phdr_buf               # base pointer
    mov ecx, r15d                   # count

ph_loop:
    test ecx, ecx
    je exit

    # p_type at offset 0
    cmp dword ptr [rsi + 0], 1      # PT_LOAD
    jne .next

    # print p_vaddr (offset 0x10)
    mov rdi, qword ptr [rsi + 16]
    call write_hex64
    mov rdi, 1
    lea rsi, newline
    mov rdx, 1
    mov rax, 1
    syscall
```

```
    # print p_memsz (offset 0x28)
    mov rdi, qword ptr [rsi + 40]
    call write_hex64
    mov rdi, 1
    lea rsi, newline
    mov rdx, 1
    mov rax, 1
    syscall

.next:
    add rsi, r14                # advance by e_phentsize
    dec ecx
    jmp ph_loop

exit:
    mov rdi, 0
    mov rax, 60
    syscall
```

This program performs essential loader-style inspection:

- walks the program header table correctly by e_phentsize,
- filters by PT_LOAD,
- extracts p_vaddr and p_memsz using correct offsets,
- uses no libc and relies only on syscalls.

4.1.6 Low-Level Rules for PT_LOAD Validation

A binary inspector should enforce strict constraints:

1. File Size vs Memory Size:
   p_memsz $\geq$ p_filesz must always hold.

2. Alignment:
   p_align must be a power of two and typically $\geq$ page size.

3. Address Range Safety:
   p_vaddr + p_memsz must not overflow 64-bit arithmetic.

4. Permissions:
   Executable segments should not be writable unless explicitly required.

5. Non-overlapping Segments:
   PT_LOAD ranges must not overlap in a way that breaks mapping logic.

6. Entry Point Validity:
   e_entry must fall within an executable PT_LOAD region.

4.1.7 Summary

PT_LOAD entries define the true runtime memory layout of an ELF program. Without them, execution is impossible. Every loader operation—mapping file bytes into memory, zero-filling .bss, and enforcing memory protections—derives directly from PT_LOAD fields.
A correct low-level inspector must understand these mappings precisely, validate them strictly, and treat them as the authoritative runtime description of the binary.

4.2 Section Names and Attributes

While program headers define runtime mappings, section headers describe the logical organization of the ELF file for linkers, assemblers, and inspection tools. The kernel

loader does not need section headers for execution, but they remain essential for disassembly, symbol resolution, relocation analysis, and debugging workflows.
Each section header entry describes:

- the section's name (as an offset into .shstrtab)
- its type (sh_type)
- its attributes (sh_flags)
- file offsets and sizes
- alignment and linkage metadata

4.2.1 Section Names Reside in a Dedicated String Table

Section names are not stored inline in section headers. Instead:

- sh_name is a 32-bit offset
- indexing into the section-name string table .shstrtab

To resolve section names:

1. read e_shstrndx from the ELF header
2. locate the .shstrtab section header
3. read its sh_offset and sh_size
4. compute name pointers as base + sh_name

4.2.2 Section Attributes (sh_flags)

Important flags include:

- SHF_WRITE — writable
- SHF_ALLOC — contributes to mapped memory
- SHF_EXECINSTR — executable instructions
- SHF_MERGE, SHF_STRINGS — mergeable string data
- SHF_TLS — thread-local storage

The kernel does not load sections by these flags; runtime mapping comes from PT_LOAD. Inspectors, however, use these flags to understand file semantics.

4.2.3 Section Types (sh_type)

Common types include:

- SHT_PROGBITS — code/data bytes in file
- SHT_NOBITS — occupies memory but not file (e.g., .bss)
- SHT_SYMTAB, SHT_DYNSYM — symbol tables
- SHT_STRTAB — string tables
- SHT_RELA, SHT_REL — relocations
- SHT_DYNAMIC — dynamic linking metadata
- SHT_NOTE — ABI and auxiliary notes

4.2.4 Manual Section Name Resolution in Raw Assembly

The program below demonstrates how to:

1. read ELF header
2. read section header table
3. locate .shstrtab
4. print section names by resolving sh_name

No libc. Pure syscalls. GAS Intel syntax.

Assembly Example — Resolve and Print Section Names

```
.intel_syntax noprefix
.global _start

.section .data
file:
    .asciz "input.elf"
newline:
    .asciz "\n"

.section .bss
ehdr:
    .skip 64
shdr_buf:
    .skip 8192
shstrtab:
    .skip 8192

.section .text
```

```
# read_exact(fd=rdi, buf=rsi, count=rdx)
# returns rax = bytes read
read_exact:
    mov r8, rdx
    mov r9, rdx
.loop:
    test r8, r8
    je .done
    xor rax, rax            # SYS_read
    mov rdx, r8
    syscall
    test rax, rax
    jle .done
    add rsi, rax
    sub r8, rax
    jmp .loop
.done:
    mov rax, r9
    sub rax, r8
    ret

# write_cstr(rsi=ptr)
# prints until NUL byte
write_cstr:
    xor rdx, rdx
.count:
    cmp byte ptr [rsi + rdx], 0
    je .print
    inc rdx
    jmp .count
.print:
    mov rdi, 1
```

```
    mov rax, 1              # SYS_write
    syscall
    ret

_start:
    # openat("input.elf", O_RDONLY)
    mov rdi, -100
    lea rsi, file
    xor rdx, rdx
    xor r10, r10
    mov rax, 257
    syscall
    mov r12, rax
    test r12, r12
    js exit

    # read ELF header
    mov rdi, r12
    lea rsi, ehdr
    mov rdx, 64
    call read_exact
    cmp rax, 64
    jne exit

    # e_shoff (0x28), e_shentsize (0x3A), e_shnum (0x3C), e_shstrndx (0x3E)
    mov r13, qword ptr [ehdr + 40]    # e_shoff
    movzx r14d, word ptr [ehdr + 58]  # e_shentsize
    movzx r15d, word ptr [ehdr + 60]  # e_shnum
    movzx ebx,  word ptr [ehdr + 62]  # e_shstrndx

    # validate shentsize for ELF64 section headers (expected 64)
    cmp r14d, 64
    jne exit
```

```
# total size = e_shentsize * e_shnum
mov eax, r14d
imul eax, r15d
mov r10, rax
cmp r10, 8192
ja exit

# lseek(fd, e_shoff, SEEK_SET)
mov rdi, r12
mov rax, 8              # SYS_lseek
mov rsi, r13
xor rdx, rdx
syscall
test rax, rax
js exit

# read section header table
mov rdi, r12
lea rsi, shdr_buf
mov rdx, r10
call read_exact
cmp rax, r10
jne exit

# locate .shstrtab section header:
# shdr_addr = shdr_buf + (e_shstrndx * e_shentsize)
mov eax, ebx
imul rax, r14
lea r11, [shdr_buf + rax]

# read .shstrtab contents using sh_offset/sh_size
mov r13, qword ptr [r11 + 24]     # sh_offset
```

```
    mov r10, qword ptr [r11 + 32]     # sh_size
    cmp r10, 8192
    ja exit

    mov rdi, r12
    mov rax, 8                        # SYS_lseek
    mov rsi, r13
    xor rdx, rdx
    syscall
    test rax, rax
    js exit

    mov rdi, r12
    lea rsi, shstrtab
    mov rdx, r10
    call read_exact
    cmp rax, r10
    jne exit

    # iterate all section headers and print names
    xor ecx, ecx                      # index = 0

sec_loop:
    cmp ecx, r15d
    je exit

    mov eax, ecx
    imul rax, r14
    lea rsi, [shdr_buf + rax]         # current shdr

    mov eax, dword ptr [rsi + 0]      # sh_name
    lea rsi, [shstrtab + rax]         # name pointer
    call write_cstr
```

```
    mov rdi, 1
    lea rsi, newline
    mov rdx, 1
    mov rax, 1
    syscall

    inc ecx
    jmp sec_loop

exit:
    mov rdi, 0
    mov rax, 60
    syscall
```

This demonstrates section-name decoding from first principles using only:

- fixed-offset ELF64 decoding
- manual indexing into .shstrtab
- syscalls only (no glibc)

4.2.5 Why Section Names Matter in Inspection

Section names link ELF internals to meaningful concepts:

- .text — executable instructions
- .rodata — constants and strings
- .data — initialized globals
- .bss — uninitialized data

- .plt, .got — dynamic linking machinery
- .symtab, .dynsym — symbol tables
- .debug_* — DWARF debug information

The kernel does not use section names at runtime, but inspection tools do for disassembly labeling, relocation analysis, symbol resolution, and static auditing.

4.2.6 Summary

Program headers define runtime memory mapping, while section headers define file-level organization for tooling. PT_LOAD entries are authoritative for execution. Section names and attributes are essential for professional static analysis, disassembly, relocation decoding, and debugging metadata extraction. A correct inspector must understand both layers while never confusing section layout with the true runtime memory image.

4.3 Executable vs Linkable Sections

In ELF64, sections serve two distinct roles:

1. Executable sections contribute to the runtime mapped image and may contain CPU instructions.
2. Linkable sections provide metadata (symbols, relocations, debug information) and are not required for execution.

The Linux loader uses program headers, not section headers, to construct the executing image. Sections exist primarily for linkers and tools.
A low-level inspector must therefore separate:

- executable memory (PT_LOAD-defined)
- static metadata sections (non-allocated)

4.3.1 Attributes That Distinguish the Two Classes

Executable sections typically satisfy:

- sh_flags & SHF_ALLOC
- sh_flags & SHF_EXECINSTR

Linkable-only sections typically satisfy:

- !(sh_flags & SHF_ALLOC)

These distinctions are essential for correct disassembly boundaries and for detecting malformed or deceptive ELF layouts.

4.3.2 Minimal Assembly Example — Print Executable Sections Only

The program below:

1. loads the ELF header
2. walks section headers
3. resolves names via .shstrtab
4. prints only sections with SHF_ALLOC | SHF_EXECINSTR

No libc. Pure syscalls. GAS Intel syntax.

Assembly Example

```
.intel_syntax noprefix
.global _start

.section .data
file:
    .asciz "input.elf"
newline:
    .asciz "\n"

.section .bss
ehdr:
    .skip 64
shdr_buf:
    .skip 8192
shstrtab:
    .skip 8192

.section .text

read_exact:
    mov r8, rdx
    mov r9, rdx
.loop:
    test r8, r8
    je .done
    xor rax, rax
    mov rdx, r8
    syscall
    test rax, rax
    jle .done
    add rsi, rax
    sub r8, rax
    jmp .loop
```

```
.done:
    mov rax, r9
    sub rax, r8
    ret

write_cstr:
    xor rdx, rdx
.count:
    cmp byte ptr [rsi + rdx], 0
    je .print
    inc rdx
    jmp .count
.print:
    mov rdi, 1
    mov rax, 1
    syscall
    ret

_start:
    mov rdi, -100
    lea rsi, file
    xor rdx, rdx
    xor r10, r10
    mov rax, 257
    syscall
    mov r12, rax
    test r12, r12
    js exit

    mov rdi, r12
    lea rsi, ehdr
    mov rdx, 64
    call read_exact
```

```
cmp rax, 64
jne exit

mov r13, qword ptr [ehdr + 40]    # e_shoff
movzx r14d, word ptr [ehdr + 58]  # e_shentsize
movzx r15d, word ptr [ehdr + 60]  # e_shnum
movzx ebx,  word ptr [ehdr + 62]  # e_shstrndx

cmp r14d, 64
jne exit

mov eax, r14d
imul eax, r15d
mov r10, rax
cmp r10, 8192
ja exit

mov rdi, r12
mov rax, 8
mov rsi, r13
xor rdx, rdx
syscall

mov rdi, r12
lea rsi, shdr_buf
mov rdx, r10
call read_exact

mov eax, ebx
imul rax, r14
lea r11, [shdr_buf + rax]         # shstrtab shdr
mov r13, qword ptr [r11 + 24]     # sh_offset
mov r10, qword ptr [r11 + 32]     # sh_size
```

```
    cmp r10, 8192
    ja exit

    mov rdi, r12
    mov rax, 8
    mov rsi, r13
    xor rdx, rdx
    syscall

    mov rdi, r12
    lea rsi, shstrtab
    mov rdx, r10
    call read_exact

    xor ecx, ecx

sec_loop:
    cmp ecx, r15d
    je exit

    mov eax, ecx
    imul rax, r14
    lea rsi, [shdr_buf + rax]          # current shdr

    mov rax, qword ptr [rsi + 8]       # sh_flags
    and rax, 0x6                       # SHF_ALLOC(2) | SHF_EXECINSTR(4)
    cmp rax, 0x6
    jne .next

    mov eax, dword ptr [rsi + 0]       # sh_name
    lea rsi, [shstrtab + rax]
    call write_cstr
```

```
    mov rdi, 1
    lea rsi, newline
    mov rdx, 1
    mov rax, 1
    syscall

.next:
    inc ecx
    jmp sec_loop

exit:
    mov rdi, 0
    mov rax, 60
    syscall
```

4.3.3 Summary

Executable sections are defined by binary attributes such as SHF_ALLOC and SHF_EXECINSTR, while linkable sections exist primarily for tooling and metadata. Program headers remain the authoritative source of runtime memory mappings. A correct inspector must treat these layers separately and validate them against each other to avoid incorrect or unsafe conclusions.

Chapter 5

Symbol Tables

5.1 .symtab vs .dynsym

ELF files typically contain two distinct symbol tables, each serving a different role in the toolchain and runtime model:

1. .symtab — the complete static symbol table
2. .dynsym — the runtime dynamic symbol table

Although both tables use the same entry structure (Elf64_Sym), their purpose, visibility, and consumers differ fundamentally. A correct low-level inspector must treat them as logically independent entities.

5.1.1 Purpose and Meaning

- .symtab — Static Symbol Table

  The .symtab section contains:

  - all symbols produced during compilation and linking
  - local, global, and static symbols
  - function names, global variables, local labels
  - compiler-generated and optimization artifacts
  - cross-object linkage metadata

  This table is intended for:

  - linkers
  - debuggers
  - disassemblers
  - static analyzers
  - reverse-engineering tools

  The Linux kernel never loads .symtab at runtime. It has no effect on execution.

- .dynsym — Dynamic Symbol Table

  The .dynsym section contains only the minimum set of symbols required at runtime:

  - externally visible functions
  - imported library symbols
  - exported globals used by other objects

  Typical entries include:

  - printf

  - malloc
  - shared-library global variables

  This table is tightly coupled with:

  - .dynstr
  - .rela.plt / .rela.dyn
  - .got / .got.plt
  - .hash or .gnu.hash
  - the dynamic loader (ld.so)

  Unlike .symtab, .dynsym is required at runtime.

5.1.2 Linker vs Loader Responsibilities

- Linker (Build Time)

  The linker uses .symtab to:

  - resolve references between object files
  - compute relocations
  - enforce visibility rules
  - perform dead-code elimination
  - generate .dynsym from exported symbols

- Loader (Runtime)

  The loader completely ignores .symtab. It uses only:

  - .dynsym

  - .dynstr
  - dynamic relocation entries

For inspection tools:

  - .symtab explains how the binary was built
  - .dynsym explains how the binary executes

5.1.3 Structural Differences

Although both tables use Elf64_Sym entries:

- .symtab usually contains many more symbols
- .dynsym is aligned with dynamic hash tables
- .symtab symbols may reference non-loaded sections
- .dynsym symbols must correspond to relocations
- .symtab names are stored in .strtab
- .dynsym names are stored in .dynstr

The loader never consults .strtab.

5.1.4 Why Both Tables Matter

- .symtab reveals:
  - internal structure of the binary
  - local function boundaries

  - compiler intent and optimization artifacts
  - symbols used for static analysis and debugging

- .dynsym reveals:

  - imported and exported symbols
  - dynamic dependencies
  - PLT/GOT participants
  - runtime relocation targets

5.1.5 Summary

.symtab and .dynsym share a common structure but serve entirely different roles:

- .symtab — full static symbol information, tooling only
- .dynsym — minimal runtime symbol set, required for execution

A correct ELF inspector must never confuse these tables or assume that symbols visible during static analysis necessarily participate in runtime behavior.

5.2 Resolving Symbol Addresses

Resolving a symbol address means converting a symbol entry into a concrete runtime address. This may refer to:

- a function entry point
- a global variable

- a dynamically relocated symbol
- a relocation target patched at load time

Symbol resolution requires combining symbol metadata with the memory layout defined by PT_LOAD segments and relocations.

5.2.1 Raw Symbol Structure (Elf64_Sym)

Each symbol entry contains:

- st_name — offset into a string table
- st_info — binding and type
- st_other — visibility
- st_shndx — section index
- st_value — section-relative address
- st_size — size of the symbol object

The symbol itself does not contain an absolute runtime address.

5.2.2 Static Symbol Resolution

For .symtab symbols with st_shndx != SHN_UNDEF:

```
runtime_address =
    section_header.sh_addr +
    st_value
```

This is valid only for sections marked SHF_ALLOC. Sections such as .symtab, .strtab, and .debug_* are never mapped.

5.2.3 Undefined Symbols

Symbols with st_shndx = SHN_UNDEF:

- must be resolved by the linker (static)
- or by the dynamic loader using .dynsym

Static resolution happens at link time; dynamic resolution happens at runtime.

5.2.4 Dynamic Symbol Resolution

Dynamic symbols are resolved using:

- .dynsym
- .dynstr
- PLT and GOT
- relocation entries
- dynamic hash tables

At runtime, the loader computes final addresses by applying relocation rules. The value stored in st_value is generally not the final address.

5.2.5 Summary

Symbol resolution is the process of translating symbolic metadata into runtime addresses. A correct inspector must:

- interpret st_shndx correctly

- apply section base addresses
- respect PT_LOAD mappings
- follow relocation logic for dynamic symbols
- never assume st_value is absolute

Symbol addresses only become meaningful in the context of the ELF memory layout.

5.3 String Tables

String tables store names only. They contain no executable data and are ignored by the kernel at runtime.
They provide names for:

- symbols (.strtab, .dynstr)
- sections (.shstrtab)

5.3.1 Structure of String Tables

A string table is a contiguous array of null-terminated strings:

```
00 name0 00 name1 00 name2 00 ...
```

The first byte is always zero, representing the empty string.
To resolve a name:

```
name = &string_table[offset]
```

5.3.2 Types of ELF String Tables

1. .strtab

   Used by .symtab and static tools.

2. .dynstr

   Used by .dynsym and the dynamic loader.

3. .shstrtab

   Used by section headers to name sections.

5.3.3 Summary

String tables are simple offset-indexed name repositories. A correct ELF inspector must:

- locate the correct string table
- map symbol tables to their corresponding strings
- resolve names manually using offsets
- never assume the kernel interprets string data

All symbol names, section names, and dynamic identifiers ultimately reside in string tables, making them essential for any ELF inspection workflow.

Chapter 6

Hex Viewer Implementation

6.1 Offsets and Rows

A hex viewer is one of the most fundamental tools in low-level binary inspection. When analyzing ELF files, raw binaries, or memory dumps, the ability to display bytes in a stable, deterministic layout is essential. The canonical model is a row-based grid, where each row begins with a file offset, followed by a fixed number of bytes rendered in hexadecimal and optionally mirrored in ASCII.
Offset computation and row generation form the foundation of the entire viewer. Before interpreting ELF headers, sections, or disassembly, the viewer must reliably access arbitrary file positions and present a precise, byte-accurate representation.

6.1.1 Linear File Positioning and Row Boundaries

A file is a linear stream of bytes indexed from offset zero. A hex viewer divides this stream into rows, each representing a fixed-width window over the file. A common width is 16 bytes per row, though 8, 32, or 64 bytes may be used in specialized

contexts.
Given:

- offset — absolute byte position in the file
- row_size — bytes per row (typically 16)
- total_size — file size in bytes

Each row begins at:

```
row_base = row_index * row_size
```

and covers:

```
[row_base ... row_base + row_size - 1]
```

If the final row exceeds the file size, only available bytes are displayed.
Offsets are purely numeric file positions. They are not memory addresses, ELF virtual addresses, or section-relative offsets. The hex viewer intentionally exposes the raw byte layout without interpretation.

6.1.2 Hexadecimal Offset Encoding

Offsets are displayed in hexadecimal because:

- ELF structures use hexadecimal offsets
- memory pages and segments are hex-aligned
- program and section headers store offsets in hex
- binary inspection operates at byte granularity

Each row prefix must display:

- a fixed-width hexadecimal offset (8 or 16 digits)
- a separator, typically :

Example:

```
00000000:  7f 45 4c 46 ...
00000010:  02 01 01 03 ...
```

This allows immediate correlation with ELF fields such as e_phoff, e_shoff, and p_offset.

6.1.3 Raw Reading Loop for Row Generation

A hex viewer relies on precise sequential reads. Using raw syscalls (read, lseek), a minimal viewer must:

1. seek to the desired offset
2. read a complete row or until EOF
3. print the offset
4. print each byte in hexadecimal
5. optionally print ASCII equivalents

Partial reads are legal at the syscall level and must be handled explicitly. The viewer must retry reads until the row is complete or EOF is reached.

6.1.4 Minimal Assembly Implementation — Row Reader

Below is a corrected minimal hex-viewer core that:

- reads 16 bytes per row
- prints the row offset
- prints each byte in hexadecimal
- uses raw syscalls only
- handles EOF correctly

Intel-Syntax Assembly — Pure Syscalls

```
.intel_syntax noprefix
.global _start

.section .data
file:       .asciz "input.bin"
hexchars:   .asciz "0123456789abcdef"
newline:    .byte 10
space:      .byte ' '

.section .bss
rowbuf:     .skip 16
offbuf:     .skip 16
tmp:        .skip 1

.section .text
```

```
;---------------------------------------------
; print_hex64: RDI = 64-bit value
;---------------------------------------------
print_hex64:
    mov rcx, 16
    lea rbx, [rel hexchars]
    lea rsi, [rel offbuf]
.loop:
    mov rax, rdi
    and rax, 0xF
    mov al, byte ptr [rbx + rax]
    mov [rsi + rcx - 1], al
    shr rdi, 4
    dec rcx
    jnz .loop

    mov rdi, 1
    mov rdx, 16
    mov rax, 1
    syscall
    ret

;---------------------------------------------
; print_hex_byte: RDI = byte value
;---------------------------------------------
print_hex_byte:
    lea rbx, [rel hexchars]
```

```
mov rax, rdi
shr rax, 4
mov al, byte ptr [rbx + rax]
mov [tmp], al
mov rdi, 1
lea rsi, [tmp]
mov rdx, 1
mov rax, 1
syscall

mov rax, rdi
and rax, 0xF
mov al, byte ptr [rbx + rax]
mov [tmp], al
mov rdi, 1
lea rsi, [tmp]
mov rdx, 1
mov rax, 1
syscall

mov rdi, 1
lea rsi, [space]
mov rdx, 1
mov rax, 1
syscall
ret
```

```
;---------------------------------------------
; read_row: RDI = fd, returns RAX = bytes read
;---------------------------------------------
read_row:
    lea rsi, [rel rowbuf]
    mov rdx, 16
    mov rax, 0
    syscall
    ret

_start:
    ; open file
    mov rdi, -100
    lea rsi, [rel file]
    xor rdx, rdx
    xor r10, r10
    mov rax, 257
    syscall
    mov r12, rax

    xor r13, r13     ; offset

row_loop:
    mov rdi, r12
    call read_row
    test rax, rax
    jle done
```

```
    mov rdi, r13
    call print_hex64

    mov rdi, 1
    lea rsi, [space]
    mov rdx, 1
    mov rax, 1
    syscall

    mov rcx, rax
    lea rbx, [rel rowbuf]

.byte_loop:
    movzx rdi, byte ptr [rbx]
    call print_hex_byte
    inc rbx
    loop .byte_loop

    mov rdi, 1
    lea rsi, [newline]
    mov rdx, 1
    mov rax, 1
    syscall

    add r13, rax
    jmp row_loop

done:
```

```
mov rdi, 0
mov rax, 60
syscall
```

This code demonstrates:

- deterministic offset computation
- row-based file traversal
- explicit hex formatting
- syscall-only I/O

It forms the structural core of any serious hex viewer.

6.1.5 Why Offsets and Rows Matter for ELF Inspection

Hex rows allow direct correlation between raw bytes and:

- ELF headers at fixed offsets
- program header tables
- section contents
- relocation records
- PLT and GOT regions
- ABI-mandated alignment boundaries

Since ELF analysis frequently jumps to arbitrary offsets (e_phoff, e_shoff, sh_offset, p_offset), a viewer must translate offsets into rows accurately and consistently.

6.1.6 Summary

Offsets and rows are the foundation of any low-level hex viewer. A correct implementation must:

- treat files as linear byte streams
- compute offsets deterministically
- present offsets in hexadecimal
- read rows using raw syscalls
- generate stable, byte-accurate output

All higher-level features—ASCII columns, navigation, ELF overlays—build on this foundation.

Chapter 7

Building an ELF Section Inspector

7.1 Parsing Section Headers

Section headers form the structural map of an ELF file. Unlike program headers, which describe loadable segments used at runtime, section headers describe the logical organization required by linkers, debuggers, and static analysis tools. A section inspector must therefore parse section headers precisely, strictly following the binary layout defined by the ELF specification and avoiding any libc abstractions.
At its core, section-header parsing is a deterministic procedure:

1. Read the ELF header (Elf64_Ehdr).
2. Locate the section header table via e_shoff.
3. Determine entry count and size using e_shnum and e_shentsize.
4. Read the complete section header array.
5. Locate the section-name string table using e_shstrndx.

6. Resolve section names by indexing into .shstrtab.

Each step must treat offsets, sizes, and fields exactly as stored in the file, with no assumptions about alignment, padding, or higher-level meaning.

7.1.1 Required ELF Header Fields

A minimal section inspector requires only four fields from the ELF header:

- e_shoff — file offset of the section header table
- e_shentsize — size of one Elf64_Shdr entry
- e_shnum — number of section header entries
- e_shstrndx — index of the section-name string table

All values are fixed-size integers encoded in little-endian format on x86-64. Once loaded, they allow direct navigation to any section header without interpretation or heuristics.

7.1.2 Structure of Elf64_Shdr

Each ELF64 section header is a fixed 64-byte structure containing:

- sh_name — offset into .shstrtab (32-bit)
- sh_type — section semantic type
- sh_flags — attribute flags
- sh_addr — virtual address (if allocated)
- sh_offset — file offset of section data

- sh_size — size of section data
- sh_link, sh_info — auxiliary linkage fields
- sh_addralign, sh_entsize — alignment and entry size

A section inspector must treat these fields as raw binary data. No field may be assumed to be ASCII-aligned, null-terminated, or implicitly validated.

7.1.3 Minimal Workflow for Parsing Section Headers

A complete parsing routine performs the following operations:

1. Open the ELF file.
2. Read the ELF header (first 64 bytes).
3. Seek to the section header table using e_shoff.
4. Read all section headers into a contiguous buffer.
5. Locate the .shstrtab section using e_shstrndx.
6. Read the .shstrtab contents.
7. For each section header:
    - extract raw fields
    - resolve the section name via sh_name

The result is a fully decoded list of sections combining binary metadata with human-readable identifiers.

7.1.4 Pure Assembly Implementation — Reading All Section Headers

The following Intel-syntax assembly code demonstrates a minimal but correct ELF section parser. It:

- opens an ELF file
- reads the ELF header
- loads the section header table
- loads .shstrtab
- prints section names
- uses raw syscalls only

Intel Assembly — Minimal ELF Section Header Parser

```
.intel_syntax noprefix
.global _start

.section .data
file:       .asciz "input.elf"
newline:    .byte '\n'

.section .bss
ehdr:       .skip 64
shdrs:      .skip 8192
shstrtab:   .skip 8192
```

```
.section .text

write_cstr:
    xor rdx, rdx
.loop:
    cmp byte ptr [rsi + rdx], 0
    je .emit
    inc rdx
    jmp .loop
.emit:
    mov rdi, 1
    mov rax, 1
    syscall
    ret

_start:
    ; open file
    mov rdi, -100
    lea rsi, [rel file]
    xor rdx, rdx
    xor r10, r10
    mov rax, 257
    syscall
    mov r12, rax

    ; read ELF header
    mov rdi, r12
    lea rsi, [rel ehdr]
```

```
mov rdx, 64
mov rax, 0
syscall

; extract section table metadata
mov rbx, [ehdr + 40]          ; e_shoff
movzx r13, word ptr [ehdr + 58] ; e_shentsize
movzx r14, word ptr [ehdr + 60] ; e_shnum
movzx r15, word ptr [ehdr + 62] ; e_shstrndx

; seek to section header table
mov rdi, r12
mov rax, 8
mov rsi, rbx
xor rdx, rdx
syscall

; read section headers
mov rdi, r12
lea rsi, [rel shdrs]
mov rdx, 8192
mov rax, 0
syscall

; locate shstrtab header
mov rax, r15
imul rax, r13
lea r8, [rel shdrs + rax]
```

```
    ; read shstrtab
    mov rbx, [r8 + 24]
    mov rcx, [r8 + 32]

    mov rdi, r12
    mov rax, 8
    mov rsi, rbx
    xor rdx, rdx
    syscall

    mov rdi, r12
    lea rsi, [rel shstrtab]
    mov rdx, rcx
    mov rax, 0
    syscall

    ; iterate and print section names
    xor rbx, rbx
.loop_sec:
    cmp rbx, r14
    je .done

    mov rax, rbx
    imul rax, r13
    lea r8, [rel shdrs + rax]

    mov eax, dword ptr [r8]
```

```
    lea rsi, [rel shstrtab + rax]
    call write_cstr

    mov rdi, 1
    lea rsi, [rel newline]
    mov rdx, 1
    mov rax, 1
    syscall

    inc rbx
    jmp .loop_sec

.done:
    mov rdi, 0
    mov rax, 60
    syscall
```

7.1.5 Key Low-Level Rules for Parsing Section Headers

A correct section inspector must enforce the following rules:

1. Never trust section names without validating .shstrtab.
2. Always advance section headers using e_shentsize.
3. Locate .shstrtab exclusively via e_shstrndx.
4. Treat sh_offset values strictly as file offsets.
5. Read all fields exactly as stored; never reinterpret structure layout.

7.1.6 Why Section Header Parsing Is Foundational

Correct section parsing enables all higher-level ELF analysis:

- locating symbol tables
- resolving string tables
- identifying relocation records
- separating code and data regions
- validating linker output

Without an accurate section parser, no reliable ELF inspection is possible.

7.1.7 Summary

Building an ELF section inspector requires:

- precise reading of ELF header metadata
- exact navigation using raw offsets
- strict interpretation of Elf64_Shdr entries
- name resolution via .shstrtab
- complete avoidance of libc abstractions

This foundational capability enables all subsequent ELF analysis stages, including symbol inspection, relocation processing, and memory layout verification.

Chapter 8

Final Project

8.1 Standalone ELF Section Viewer (No libc)

The objective of this final project is to implement a fully standalone ELF section viewer written entirely in x86-64 assembly and built directly on the Linux syscall interface. The program uses no C library, no runtime, no dynamic linker, and no helper abstractions. It interacts exclusively with the kernel.
This viewer represents the culmination of all concepts developed throughout this book:

- manual ELF parsing
- strict boundary validation
- raw offset-based navigation
- controlled, deterministic formatting
- defensive handling of malformed binaries

A tool of this nature is more than a technical exercise. It is a proof of complete understanding. Every byte read, every structure decoded, and every string printed is the result of deliberate interaction with the kernel, mirroring how the loader itself reasons about ELF files.

8.1.1 Architectural Overview

The standalone ELF viewer is organized into four explicit subsystems, each responsible for a single stage in the parsing and reporting pipeline:

1. File Access Layer

   All interaction with the filesystem is performed using raw syscalls:

   - openat — open the ELF file
   - read — load headers and tables
   - lseek — reposition the file cursor
   - write — emit output

   There is no buffering. Every read and write is explicit and validated.

2. Validation Layer

   Before any interpretation occurs, the viewer validates:

   - ELF magic bytes
   - ELF class (64-bit)
   - section header table location and size
   - .shstrtab boundaries
   - basic structural sanity of the file

Malformed or malicious binaries are rejected immediately.

3. Extraction Layer

   Only after validation does the viewer extract:

   - section headers (Elf64_Shdr)
   - section names via .shstrtab
   - section metadata (offsets and sizes)

   All extraction is performed using raw offsets and fixed-size structures.

4. Presentation Layer

   The final layer formats and prints:

   - section index
   - resolved section name
   - file offset (sh_offset)
   - section size (sh_size)

   All numeric values are rendered manually as fixed-width hexadecimal. Output is deterministic and suitable for scripting and analysis.

8.1.2 Complete Integrated Example — Standalone ELF Viewer

The following listing is a complete, self-contained ELF section viewer. It integrates all previous components into a single, publication-ready assembly program.
The implementation:

- uses only Linux syscalls

- performs explicit validation
- avoids libc, PLT, GOT, and dynamic linking
- produces stable, low-level output

Everything is manual. Everything is explicit.

8.1.3 Full ELF Section Viewer — Intel Syntax (No libc)

```
.intel_syntax noprefix
.global _start

;=============================================================
;  STATIC DATA
;=============================================================
.section .data
file:         .asciz "input.elf"
newline:      .byte '\n'
space:        .byte ' '
lbr:          .byte '['
rbr:          .byte ']'
hexchars:     .asciz "0123456789abcdef"

msg_bad_elf: .asciz "ERR: not a valid ELF64 file\n"
msg_short:    .asciz "ERR: truncated file\n"
msg_open:     .asciz "ERR: unable to open file\n"

.section .bss
ehdr:         .skip 64
```

```
shdrs:      .skip 16384
shstr:      .skip 16384
hexbuf:     .skip 16


;=============================================================
;  BASIC WRITE ROUTINES
;=============================================================
.section .text

write_cstr:
    xor rdx, rdx
.count:
    cmp byte ptr [rsi + rdx], 0
    je .emit
    inc rdx
    jmp .count
.emit:
    mov rdi, 1
    mov rax, 1
    syscall
    ret


write_hex64:
    push rcx
    push rbx
```

```
    lea rcx, [hexbuf + 16]
    lea rbx, [rel hexchars]

.loop:
    dec rcx
    mov rax, rdi
    and rax, 0xF
    mov al, byte ptr [rbx + rax]
    mov [rcx], al
    shr rdi, 4
    cmp rcx, hexbuf
    jne .loop

    mov rdi, 1
    lea rsi, [rel hexbuf]
    mov rdx, 16
    mov rax, 1
    syscall

    pop rbx
    pop rcx
    ret



;=============================================================
;  ERROR HANDLERS
;=============================================================
fatal_open:
```

```
    mov rdi, 2
    lea rsi, [rel msg_open]
    call write_cstr
    mov rdi, 1
    mov rax, 60
    syscall

fatal_bad_elf:
    mov rdi, 2
    lea rsi, [rel msg_bad_elf]
    call write_cstr
    mov rdi, 1
    mov rax, 60
    syscall

fatal_short:
    mov rdi, 2
    lea rsi, [rel msg_short]
    call write_cstr
    mov rdi, 1
    mov rax, 60
    syscall



;=============================================================
;  PROGRAM ENTRY
;=============================================================
_start:
```

```
; open file
mov rdi, -100
lea rsi, [rel file]
xor rdx, rdx
xor r10, r10
mov rax, 257
syscall
cmp rax, 0
jl fatal_open
mov r12, rax          ; fd

; read ELF header
mov rdi, r12
lea rsi, [rel ehdr]
mov rdx, 64
xor rax, rax
syscall
cmp rax, 64
jne fatal_short

; validate ELF magic
mov eax, dword ptr [ehdr]
cmp eax, 0x464C457F   ; "\x7FELF"
jne fatal_bad_elf

; validate ELF64
mov al, byte ptr [ehdr + 4]
cmp al, 2
```

```
jne fatal_bad_elf

; extract section header metadata
mov rbx, [ehdr + 40]            ; e_shoff
movzx r13, word ptr [ehdr + 58]  ; e_shentsize
movzx r14, word ptr [ehdr + 60]  ; e_shnum
movzx r15, word ptr [ehdr + 62]  ; e_shstrndx

; seek to section header table
mov rdi, r12
mov rax, 8
mov rsi, rbx
xor rdx, rdx
syscall

; read section headers
mov rdi, r12
lea rsi, [rel shdrs]
mov rdx, 16384
xor rax, rax
syscall
cmp rax, 0
jle fatal_short

; locate shstrtab header
mov rax, r15
imul rax, r13
lea r8, [rel shdrs + rax]
```

```
    mov r9,  [r8 + 24]    ; sh_offset
    mov r10, [r8 + 32]    ; sh_size

    ; read shstrtab
    mov rdi, r12
    mov rax, 8
    mov rsi, r9
    xor rdx, rdx
    syscall

    mov rdi, r12
    lea rsi, [rel shstr]
    mov rdx, r10
    xor rax, rax
    syscall
    cmp rax, 0
    jle fatal_short

    ;=============================================================
    ; PRINT SECTIONS
    ;=============================================================
    xor rbx, rbx

section_loop:
    cmp rbx, r14
    je done
```

```
mov rax, rbx
imul rax, r13
lea r8, [rel shdrs + rax]

; resolve name
mov eax, dword ptr [r8]
lea rsi, [rel shstr + rax]

; print "[index] "
mov rdi, 1
lea rsi, [rel lbr]
mov rdx, 1
mov rax, 1
syscall

mov rdi, rbx
call write_hex64

mov rdi, 1
lea rsi, [rel rbr]
mov rdx, 1
mov rax, 1
syscall

mov rdi, 1
lea rsi, [rel space]
mov rdx, 1
mov rax, 1
```

```
syscall

; print section name
mov rdi, 1
call write_cstr

; print offset
mov rdi, 1
lea rsi, [rel space]
mov rdx, 1
mov rax, 1
syscall

mov rdi, [r8 + 24]
call write_hex64

; print size
mov rdi, 1
lea rsi, [rel space]
mov rdx, 1
mov rax, 1
syscall

mov rdi, [r8 + 32]
call write_hex64

; newline
mov rdi, 1
```

```
    lea rsi, [rel newline]
    mov rdx, 1
    mov rax, 1
    syscall

    inc rbx
    jmp section_loop

done:
    mov rdi, 0
    mov rax, 60
    syscall
```

8.1.4 Summary

This final project demonstrates a complete, low-level ELF section viewer implemented without libc. It proves mastery of:

- ELF binary structure
- raw syscall-based I/O
- deterministic offset navigation
- defensive validation of untrusted input
- manual hexadecimal formatting

Such a tool mirrors the reasoning performed by loaders, linkers, and debuggers at their lowest level. It provides a solid foundation for more advanced ELF analysis tasks, including symbol resolution, relocation processing, and memory-mapping inspection.

Conclusion

ELF is not merely a file format; it is the structural backbone of the executable environment on modern Linux systems. Understanding ELF at the byte level—without abstractions, without toolchain assistance, and without reliance on libc—demands a degree of precision that reveals how software truly becomes executable code. This book has demonstrated that correct binary interpretation is a disciplined process grounded in raw offsets, fixed-size structures, ABI-defined conventions, and predictable kernel behavior.

Throughout this volume, ELF was approached from the perspective of the kernel and the loader rather than from that of high-level utilities. By handling every field manually—validating magic bytes, enforcing boundary checks on section tables, walking string tables by offset, decoding symbols and relocations, constructing program-header mappings, and implementing a complete hex viewer and section inspector—we operated at the precise level where the ELF specification meets the machine.

Several principles emerged as fundamental to correct ELF inspection.

Deterministic Interpretation of Byte Streams

Every structure in an ELF file is a fixed-size binary record. Correct analysis requires strict adherence to alignment rules, endianness, and ABI-defined offsets. No inference,

assumption, or heuristic is acceptable when interpreting executable metadata. Precision at this level is non-negotiable.

Explicit Control Over I/O

By relying exclusively on Linux syscalls, all file access and output formatting became fully transparent. Manual reads, explicit lseek positioning, and hand-written hexadecimal conversion routines exposed the internal mechanics behind tools such as readelf and objdump. This explicit control removes all ambiguity about how data moves from disk to analysis.

Trust Nothing Without Validation

Malformed or adversarial ELF files are common in security research and real-world analysis. Proper inspection requires rejecting any structure that violates size, offset, or alignment constraints. Failure to validate inputs leads directly to undefined behavior and unreliable analysis. Defensive parsing is therefore an essential requirement, not an optional enhancement.

Structural Correlation Across ELF Components

ELF becomes meaningful only when its independent regions are connected and interpreted together:

- ELF header → program headers → loadable segments
- ELF header → section headers → symbolic and debugging structures
- string tables → section names and symbol names

- relocation entries → dynamic linking behavior
- PT_LOAD segments → executable memory layout

Only by navigating these relationships does an inspector uncover the complete execution model of a binary.

Minimal Tools Reveal Maximal Understanding

The final project—constructing a standalone ELF section viewer using only assembly and syscalls—demonstrated that high-level analysis tools can be rebuilt entirely from first principles. Writing these components manually provides absolute clarity into the structure, constraints, and behavior of ELF binaries, free from hidden abstractions.

Final Perspective

ELF inspection is not about producing formatted output or decoding symbols in isolation; it is about developing a direct understanding of binary structure and the execution model of a system. With the techniques presented in this book, the reader now possesses the ability to:

- build custom ELF readers, inspectors, and analyzers,
- debug loader and linking issues at the byte level,
- validate binary integrity manually,
- reason precisely about memory mappings and segment layouts,
- and implement binary-analysis tools without any runtime dependencies.

This knowledge is increasingly critical in environments where trust, correctness, and predictability are paramount—such as security auditing, reverse engineering, toolchain development, embedded systems, and minimal-runtime execution environments.
Book 6 serves as a central link in this series. It unifies low-level assembly, system calls, memory architecture, and runtime execution into a coherent model centered on the ELF format. Mastery of ELF provides the structural and conceptual foundation necessary to understand every subsequent stage of program execution, from the first byte on disk to the final instruction executed on the CPU.

References

The following works constitute the authoritative technical sources consulted in the preparation of this volume. Together, they define the Linux ABI, the ELF specification, the x86-64 execution model, toolchain behavior, and modern loader and relocation semantics as implemented in post-2020 Linux systems.

1. System V Application Binary Interface: AMD64 Architecture Processor Supplement

   The Open Group / AMD64 ABI Committee
   Defines the ELF64 object format, including program headers, section headers, symbol tables, relocation records, dynamic linking entries, and all binary-level conventions required for correct execution on modern x86-64 systems.

2. Intel 64 and IA-32 Architectures Software Developer's Manual

   Intel Corporation
   Provides the definitive specification of instruction semantics, memory models, addressing modes, register behavior, and execution rules relied upon by all assembly implementations and low-level analyses presented in this book.

3. Linux Kernel Documentation: ELF Loading, binfmt_elf, Memory Management, and System Call Interface

Linux Kernel Project
Documents the kernel's internal handling of ELF executables, including PT_LOAD validation, page alignment, relocation processing, interpreter loading, and low-level constraints enforced by modern Linux kernels.

4. The ELF Object File Format: Toolchain Implementation Notes

   GNU Binutils Maintainers
   LLVM Project
   Provides practical, implementation-level insight into ELF handling, including section layout assumptions, relocation semantics, symbol resolution rules, and metadata interpretation used by contemporary linkers, assemblers, and disassemblers.

5. Linkers and Loaders — Modern Annotated Editions

   John R. Levine (original author); updated technical annotations
   Although originally published earlier, modern annotated editions and commentary continue to serve as a definitive explanation of executable formats, linking models, relocation processing, and loader design principles that remain applicable to current systems.

6. The Linux Programming Interface (Updated Technical Printings)

   Michael Kerrisk
   Later printings reflect contemporary syscall behavior, memory layout considerations, and kernel interfaces, providing an authoritative reference for raw syscall usage and low-level process interaction.

7. ELF Handling in Modern Linux Toolchains

   Red Hat Toolchain Engineering Documentation

Describes modern RHEL and Fedora toolchain behavior, including hardened relocation processing, dynamic loader constraints, and section-validation rules adopted in recent Linux distributions.

8. Reverse Engineering for Modern Linux Systems

   Post-2020 technical texts and professional guides
   Covers ELF metadata interpretation, file carving, binary safety auditing, and low-level inspection workflows consistent with the parsing and validation techniques demonstrated in this volume.

9. Assembly Systems Programming Manuals for x86-64

   GAS and Intel syntax specifications
   Used as structural references for instruction usage, register conventions, operand encodings, calling conventions, and system-level assembly practices applied throughout the book.

10. Internal Source Analysis: Linux Loader and Toolchain Codebases

    Linux Kernel, GNU Binutils, glibc, and musl source code
    Examined to cross-verify interpretations of ELF structures, relocation flows, dynamic linking behavior, and ABI compliance reflected in the parsing logic and assembly implementations presented in this work.

Closing Note

The sources listed above form the authoritative technical foundation on which this volume is built. Each provides essential insight into ELF structure, loader design, ABI rules, system-call behavior, and executable semantics on modern Linux systems.

Taken together, they ensure that every definition, field description, assembly routine, and binary-inspection method presented in this book accurately reflects post-2020, production-grade Linux behavior on the x86-64 architecture.