Ethical Hacking and Pentesting [COMM3031]

MUKESH KUMAR

Binary Analysis Part-B

4/10/25

Table Of Content

1. Introduction to ELF Format

The Executable and Linkable Format (ELF) is a widely used file format for executable files, object code, shared libraries, and core dumps in Unix systems. The ELF format organizes the contents of a binary file in a way that is efficient for both linking and execution. The file structure is divided into several components, primarily consisting of the ELF header, program headers, sections, and section headers.

2. ELF Header, Sections, and Segments

a) ELF Header:

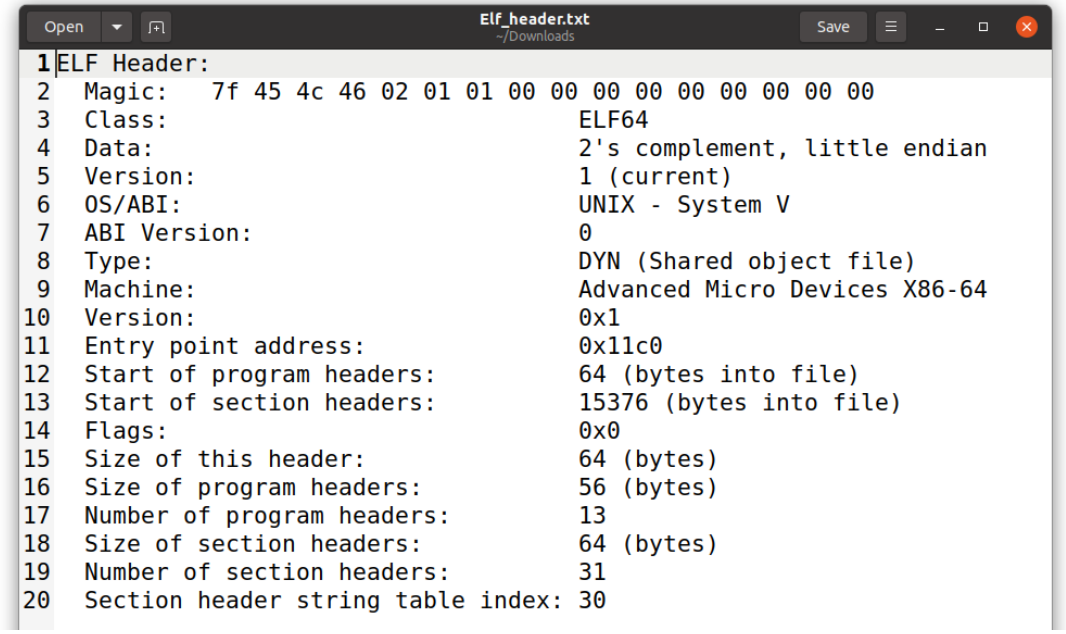


Fig 2.1: Elf Header

The ELF header serves as the foundation of an ELF file, containing crucial metadata that guides the system in interpreting and executing the binary. The provided ELF header specifies that the file type is a dynamically linked shared object (DYN), indicating it is either a shared library or a position-independent executable. The target architecture is **`x86-64`**. Notably, the entry point of the binary is at address **`0x11c0`**, which marks the starting location of execution. The ELF header also provides information regarding program headers, section headers, and ABI details, demonstrating its critical role in binary execution.

b) ELF Sections:

Elf_section.txt
~/Downloads/Part1

```
There are 31 section headers, starting at offset 0x3c10:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        0000000000000318 000318 00001c 00   A  0   0  1
  [ 2] .note.gnu.property NOTE            0000000000000338 000338 000020 00   A  0   0  8
  [ 3] .note.gnu.build-id NOTE            0000000000000358 000358 000024 00   A  0   0  4
  [ 4] .note.ABI-tag     NOTE            000000000000037c 00037c 000020 00   A  0   0  4
  [ 5] .gnu.hash         GNU_HASH        00000000000003a0 0003a0 000028 00   A  6   0  8
  [ 6] .dynsym           DYNSYM          00000000000003c8 0003c8 0001c8 18   A  7   1  8
  [ 7] .dynstr           STRTAB          0000000000000590 000590 000127 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          00000000000006b8 0006b8 000026 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         00000000000006e0 0006e0 000060 00   A  7   3  8
  [10] .rela.dyn         RELA            0000000000000740 000740 0000d8 18   A  6   0  8
  [11] .rela.plt         RELA            0000000000000818 000818 000120 18  AI  6  24  8
  [12] .init             PROGBITS        0000000000001000 001000 00001b 00  AX  0   0  4
  [13] .plt              PROGBITS        0000000000001020 001020 0000d0 10  AX  0   0 16
  [14] .plt.got          PROGBITS        00000000000010f0 0010f0 000010 10  AX  0   0 16
  [15] .plt.sec          PROGBITS        0000000000001100 001100 0000c0 10  AX  0   0 16
  [16] .text             PROGBITS        00000000000011c0 0011c0 000255 00  AX  0   0 16
  [17] .fini             PROGBITS        0000000000001418 001418 00000d 00  AX  0   0  4
  [18] .rodata           PROGBITS        0000000000002000 002000 000063 00   A  0   0  8
  [19] .eh_frame_hdr     PROGBITS        0000000000002064 002064 00004c 00   A  0   0  4
  [20] .eh_frame         PROGBITS        00000000000020b0 0020b0 000128 00   A  0   0  8
  [21] .init_array       INIT_ARRAY      0000000000003d40 002d40 000008 08  WA  0   0  8
  [22] .fini_array       FINI_ARRAY      0000000000003d48 002d48 000008 08  WA  0   0  8
  [23] .dynamic          DYNAMIC         0000000000003d50 002d50 000210 10  WA  7   0  8
  [24] .got              PROGBITS        0000000000003f60 002f60 0000a0 08  WA  0   0  8
  [25] .data             PROGBITS        0000000000004000 003000 000010 00  WA  0   0  8
  [26] .bss              NOBITS          0000000000004010 003010 000010 00  WA  0   0 16
  [27] .comment          PROGBITS        0000000000000000 003010 00002a 01  MS  0   0  1
  [28] .symtab           SYMTAB          0000000000000000 003040 000750 18     29  46  8
  [29] .strtab           STRTAB          0000000000000000 003790 000361 00      0   0  1
  [30] .shstrtab         STRTAB          0000000000000000 003af1 00011a 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

Fig 2.2: Elf Sections

An ELF section is a logically distinct part of an ELF binary, used to store different types of data necessary for program execution and linking.

Among these, the **`.text`** section contains the executable code of the binary. It has a type of PROGBITS (having program code), Its address **`0x11c0`** signifies where the code will be loaded into memory, and the offset **`0x11c0`** determines its location in the file. The section is marked with the **`AX`** flags, meaning it is both allocated in memory and executable.

In the context of the given C program, lines such as **`main()`** [line 12] and the function **`log_keypress()`** [line 7] directly contribute to the content of the **`.text`** section, since all function logic is compiled into executable machine instructions that reside in this segment

The **`.data`** and **`.bss`** sections, though both used for storing variables, serve different purposes. The **`.data`** section contains initialized global and static variables, meaning values are explicitly assigned before execution begins. In contrast, the **`.bss`** section is reserved for uninitialized global and static variables, their memory is allocated but not initialized with specific values. The **`.bss`** section does not occupy space in the actual ELF file but is allocated at runtime, making it more memory-efficient.

c) ELF Segments:

*Elf_segments.txt
~/Downloads/Part1

```
Elf file type is DYN (Shared object file)
Entry point 0x11c0
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  PHDR           0x000040 0x0000000000000040 0x0000000000000040 0x0002d8 0x0002d8 R   0x8
  INTERP         0x000318 0x0000000000000318 0x0000000000000318 0x00001c 0x00001c R   0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000000000 0x0000000000000000 0x000938 0x000938 R   0x1000
  LOAD           0x001000 0x0000000000001000 0x0000000000001000 0x000425 0x000425 R E 0x1000
  LOAD           0x002000 0x0000000000002000 0x0000000000002000 0x0001d8 0x0001d8 R   0x1000
  LOAD           0x002d40 0x0000000000003d40 0x0000000000003d40 0x0002d0 0x0002e0 RW  0x1000
  DYNAMIC        0x002d50 0x0000000000003d50 0x0000000000003d50 0x000210 0x000210 RW  0x8
  NOTE           0x000338 0x0000000000000338 0x0000000000000338 0x000020 0x000020 R   0x8
  NOTE           0x000358 0x0000000000000358 0x0000000000000358 0x000044 0x000044 R   0x4
  GNU_PROPERTY   0x000338 0x0000000000000338 0x0000000000000338 0x000020 0x000020 R   0x8
  GNU_EH_FRAME   0x002064 0x0000000000002064 0x0000000000002064 0x00004c 0x00004c R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
  GNU_RELRO      0x002d40 0x0000000000003d40 0x0000000000003d40 0x0002c0 0x0002c0 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-
  tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
   03     .init .plt .plt.got .plt.sec .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .data .bss
   06     .dynamic
   07     .note.gnu.property
   08     .note.gnu.build-id .note.ABI-tag
   09     .note.gnu.property
   10     .eh_frame_hdr
   11
   12     .init array .fini array .dynamic .got
```

Fig 2.3: Elf Segments

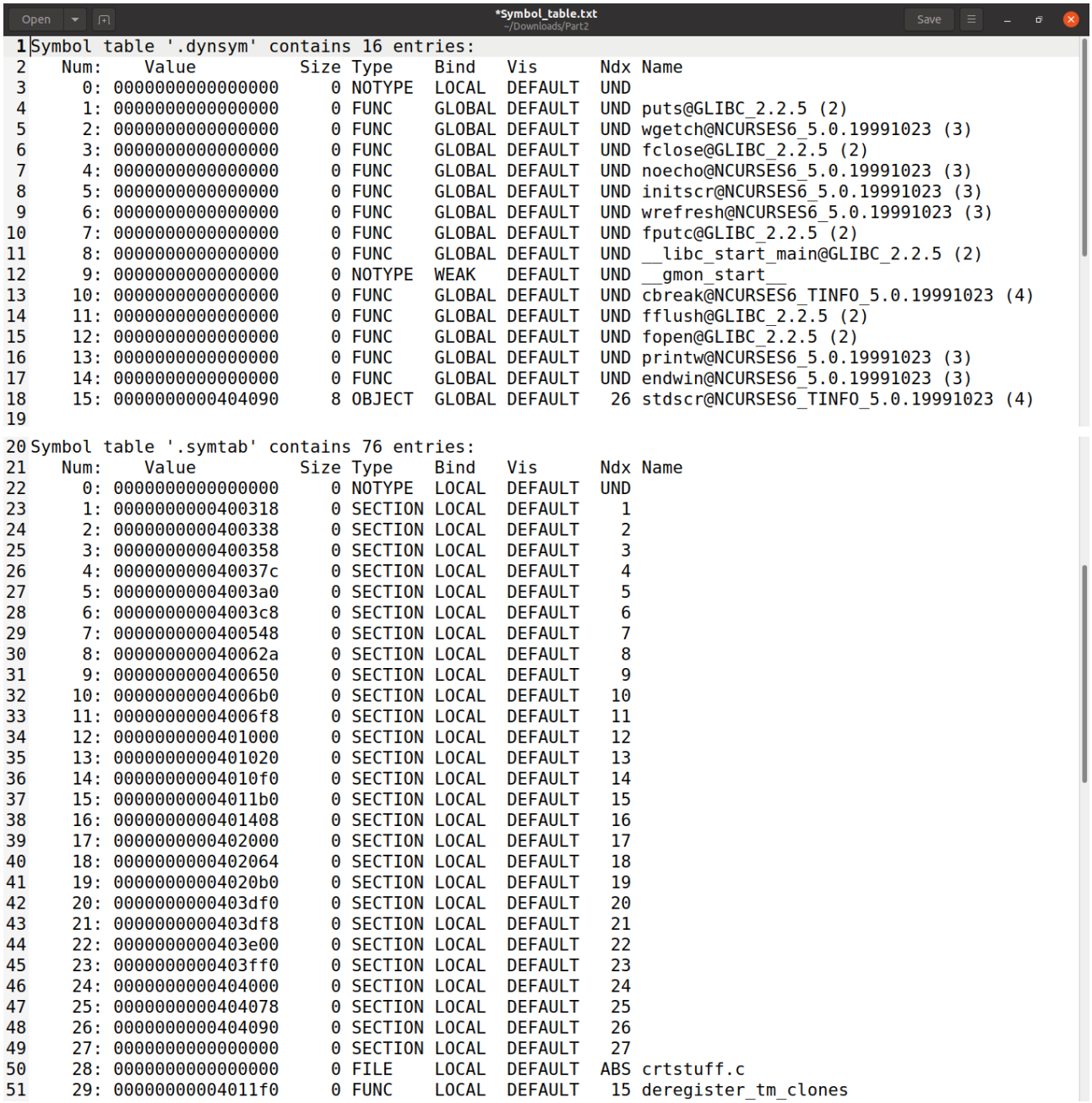
ELF segments represent higher-level memory structures that group sections into logical units during execution. While sections organize data and code within an ELF file, segments serve as a bridge between the file and the process memory, ensuring efficient loading and execution. Sections are grouped into segments based on their usage and access permissions. For instance, executable code sections such as **`.text`** and **`.plt`** are combined, while writable data sections like **`.data`** and **`.bss`** form another segment. This grouping ensures that related sections are mapped into memory efficiently while enforcing necessary access controls.

Segment	Sections Included	Flags	Related C Code
`PT_LOAD (code)`	`.interp`, `.note.*`, `.init`, `.plt`, `.text`, `.fini`, `.rodata`	Executable & Readable	Lines with **logic and print**, e.g., `printw()`, `log_keypress()`, `main()`
`PT_LOAD (data)`	`.data`, `.bss`, `.dynamic`, `.got`, `.init_array`, `.fini_array`	Writable & Readable	Lines with **variable allocations** like `FILE *logfile`, `int key`, memory access
`PT_DYNAMIC`	`.dynamic`	Used for dynamic linking	`#include <ncurses.h>` implies dynamic libraries
`PT_INTERP`	`.interp`	Contains dynamic linker path	Automatically generated by compiler/linker

The **PT_LOAD** segment is critical as it defines which portions of the ELF file should be loaded into memory. It contains executable instructions and initialized data, ensuring that the program's essential components are available during runtime. It also plays a key role in memory protection by specifying read, write, and execute permissions for different regions, preventing unintended modifications to executable code. This segmentation enhances both security and performance by optimizing memory layout and execution efficiency.

3. Inspecting Symbol Table, GOT, and PLT

a) Symbol Table:




```
Symbol table '.dynsym' contains 16 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND wgetch@NCURSES6_5.0.19991023 (3)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fclose@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND noecho@NCURSES6_5.0.19991023 (3)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND initscr@NCURSES6_5.0.19991023 (3)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND wrefresh@NCURSES6_5.0.19991023 (3)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fputc@GLIBC_2.2.5 (2)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     9: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND cbreak@NCURSES6_TINFO_5.0.19991023 (4)
    11: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fflush@GLIBC_2.2.5 (2)
    12: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fopen@GLIBC_2.2.5 (2)
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printw@NCURSES6_5.0.19991023 (3)
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND endwin@NCURSES6_5.0.19991023 (3)
    15: 0000000000404090     8 OBJECT  GLOBAL DEFAULT   26 stdscr@NCURSES6_TINFO_5.0.19991023 (4)

Symbol table '.symtab' contains 76 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000400318     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000400338     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000400358     0 SECTION LOCAL  DEFAULT    3
     4: 000000000040037c     0 SECTION LOCAL  DEFAULT    4
     5: 00000000004003a0     0 SECTION LOCAL  DEFAULT    5
     6: 00000000004003c8     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000400548     0 SECTION LOCAL  DEFAULT    7
     8: 000000000040062a     0 SECTION LOCAL  DEFAULT    8
     9: 0000000000400650     0 SECTION LOCAL  DEFAULT    9
    10: 00000000004006b0     0 SECTION LOCAL  DEFAULT   10
    11: 00000000004006f8     0 SECTION LOCAL  DEFAULT   11
    12: 0000000000401000     0 SECTION LOCAL  DEFAULT   12
    13: 0000000000401020     0 SECTION LOCAL  DEFAULT   13
    14: 00000000004010f0     0 SECTION LOCAL  DEFAULT   14
    15: 00000000004011b0     0 SECTION LOCAL  DEFAULT   15
    16: 0000000000401408     0 SECTION LOCAL  DEFAULT   16
    17: 0000000000402000     0 SECTION LOCAL  DEFAULT   17
    18: 0000000000402064     0 SECTION LOCAL  DEFAULT   18
    19: 00000000004020b0     0 SECTION LOCAL  DEFAULT   19
    20: 0000000000403df0     0 SECTION LOCAL  DEFAULT   20
    21: 0000000000403df8     0 SECTION LOCAL  DEFAULT   21
    22: 0000000000403e00     0 SECTION LOCAL  DEFAULT   22
    23: 0000000000403ff0     0 SECTION LOCAL  DEFAULT   23
    24: 0000000000404000     0 SECTION LOCAL  DEFAULT   24
    25: 0000000000404078     0 SECTION LOCAL  DEFAULT   25
    26: 0000000000404090     0 SECTION LOCAL  DEFAULT   26
    27: 0000000000000000     0 SECTION LOCAL  DEFAULT   27
    28: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    29: 00000000004011f0     0 FUNC    LOCAL  DEFAULT   15 deregister_tm_clones
```

```
52     30: 0000000000401220     0 FUNC    LOCAL  DEFAULT   15 register_tm_clones
53     31: 0000000000401260     0 FUNC    LOCAL  DEFAULT   15 __do_global_dtors_aux
54     32: 0000000000404098     1 OBJECT  LOCAL  DEFAULT   26 completed.8060
55     33: 0000000000403df8     0 OBJECT  LOCAL  DEFAULT   21 __do_global_dtors_aux_fin
56     34: 0000000000401290     0 FUNC    LOCAL  DEFAULT   15 frame_dummy
57     35: 0000000000403df0     0 OBJECT  LOCAL  DEFAULT   20 __frame_dummy_init_array_
58     36: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS keylogger.c
59     37: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
60     38: 00000000004021cc     0 OBJECT  LOCAL  DEFAULT   19 __FRAME_END__
61     39: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
62     40: 0000000000403df8     0 NOTYPE  LOCAL  DEFAULT   20 __init_array_end
63     41: 0000000000403e00     0 OBJECT  LOCAL  DEFAULT   22 _DYNAMIC
64     42: 0000000000403df0     0 NOTYPE  LOCAL  DEFAULT   20 __init_array_start
65     43: 0000000000402064     0 NOTYPE  LOCAL  DEFAULT   18 __GNU_EH_FRAME_HDR
66     44: 0000000000404000     0 OBJECT  LOCAL  DEFAULT   24 _GLOBAL_OFFSET_TABLE_
67     45: 0000000000401400     5 FUNC    GLOBAL DEFAULT   15 __libc_csu_fini
68     46: 0000000000404078     0 NOTYPE  WEAK   DEFAULT   25 data_start
69     47: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.2.5
70     48: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND wgetch@@NCURSES6_5.0.1999
71     49: 0000000000404088     0 NOTYPE  GLOBAL DEFAULT   25 _edata
72     50: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fclose@@GLIBC_2.2.5
73     51: 0000000000401408     0 FUNC    GLOBAL HIDDEN    16 _fini
74     52: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND noecho@@NCURSES6_5.0.1999
75     53: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND initscr@@NCURSES6_5.0.199
76     54: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND wrefresh@@NCURSES6_5.0.19
77     55: 0000000000404090     8 OBJECT  GLOBAL DEFAULT   26 stdscr@@NCURSES6_TINFO_5.
78     56: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fputc@@GLIBC_2.2.5
79     57: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
80     58: 0000000000404078     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
81     59: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
82     60: 0000000000404080     0 OBJECT  GLOBAL HIDDEN    25 __dso_handle
83     61: 0000000000402000     4 OBJECT  GLOBAL DEFAULT   17 _IO_stdin_used
84     62: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND cbreak@@NCURSES6_TINFO_5.
85     63: 0000000000401390   101 FUNC    GLOBAL DEFAULT   15 __libc_csu_init
86     64: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fflush@@GLIBC_2.2.5
87     65: 00000000004040a0     0 NOTYPE  GLOBAL DEFAULT   26 _end
88     66: 00000000004011e0     5 FUNC    GLOBAL HIDDEN    15 _dl_relocate_static_pie
89     67: 00000000004011b0    47 FUNC    GLOBAL DEFAULT   15 _start
90     68: 0000000000404088     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
91     69: 00000000004012c9   193 FUNC    GLOBAL DEFAULT   15 main
92     70: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fopen@@GLIBC_2.2.5
93     71: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printw@@NCURSES6_5.0.1999
94     72: 0000000000404088     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
95     73: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND endwin@@NCURSES6_5.0.1999
96     74: 0000000000401000     0 FUNC    GLOBAL HIDDEN    12 _init
97     75: 0000000000401296    51 FUNC    GLOBAL DEFAULT   15 log_keypress
```

Fig 3.1: Symbol Table

A symbol table in an ELF binary is a data structure that stores information about symbols, such as functions, variables, and sections. It helps the linker and loader resolve memory addresses and function calls. The table is divided into two parts, **`.symtab`** (which includes all symbols, mainly for debugging) and **`.dynsym`** (which contains only dynamically linked symbols needed at runtime).

Location of dynamic symbols in C program, **`fputc`** [line 8 of C], **`fflush`**[9], **`initscr`**[16], **`cbreak`**[17], and **`noecho`**[18], **`fopen`**[20], **`printw`**[27], **`refresh`**[28], **`getch`**[31], **`fclose`**[38], and **`endwin`**[39].

The **`.symtab`** symbol table also reveals internal structure, such as: **`main`**[12] program's entry point, **`log_keypress`**[7] a user-defined function for logging keypresses. And sections like **`.text`**, **`.data`**, **`.bss`**, **`.rodata`**, **`.fini`**, **`.init`**, and others essential for the memory layout and binary structure.

Additionally, local static variables like logfile and key do not appear with symbolic names, as they are scoped within **`main()`** and optimized by the compiler.

b) Global Offset Table (GOT):

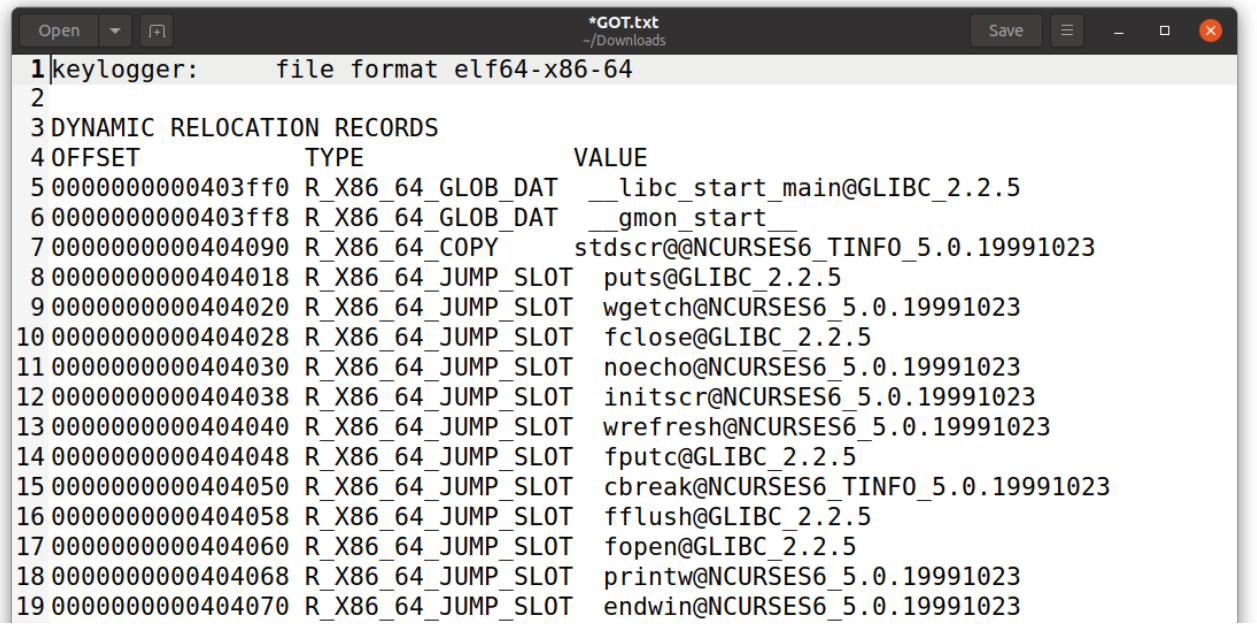
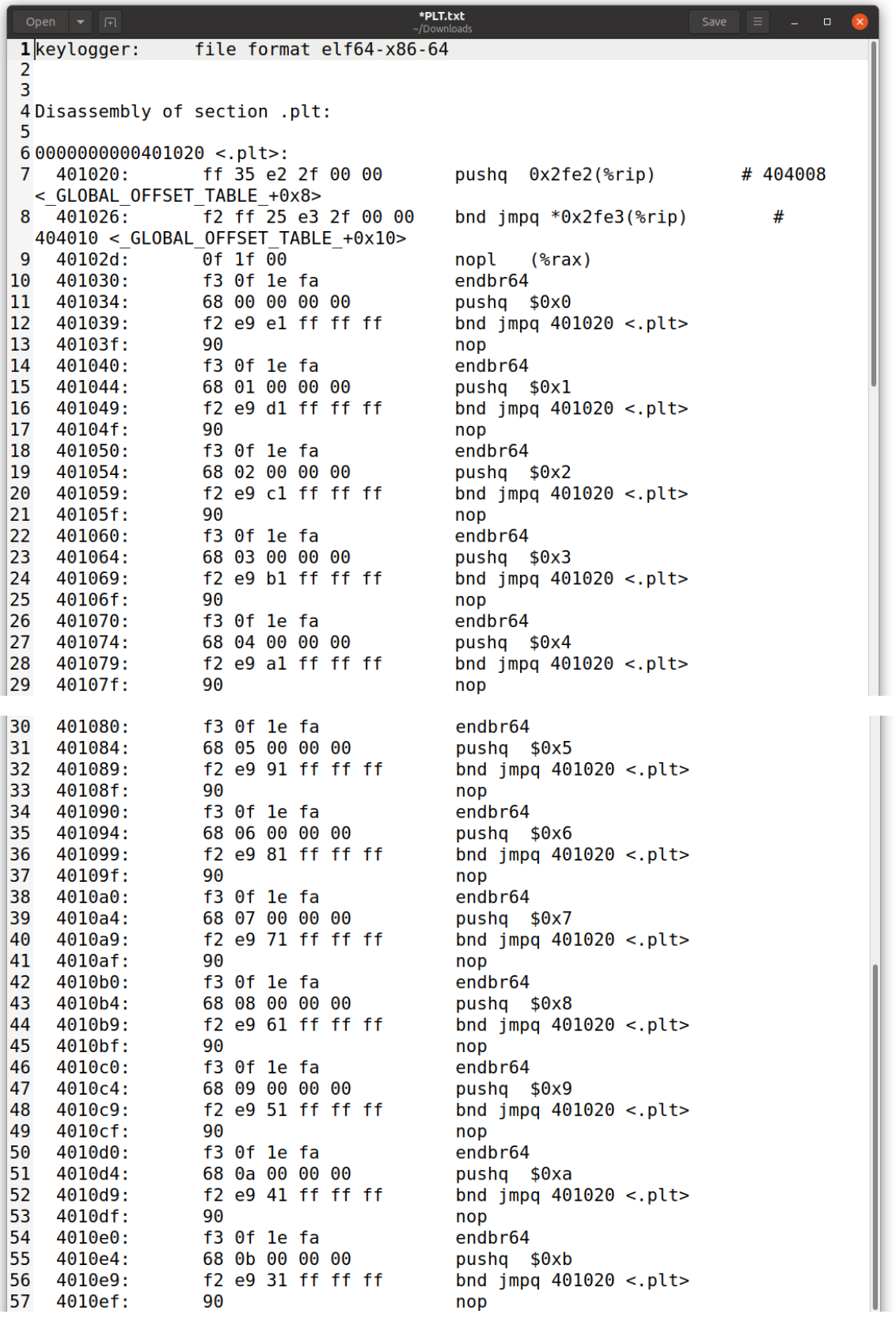


Fig 3.2: Global Offset Table (GOT)

The functions described, including **`puts`**, **`fopen`**, and **`fflush`**, etc are dynamically linked, meaning their addresses are resolved at runtime rather than during compilation. This dynamic linkage enhances memory efficiency, allowing programs to load necessary resources only when required. Functions like **`printf`** and **`main()`** manage output and program initialization, while ncurses functions such as **`wgetch`** and **`cbreak`** and other optimize terminal-based user interaction.

c) Procedure Linkage Table (PLT):




```
keylogger:     file format elf64-x86-64


Disassembly of section .plt:

0000000000401020 <.plt>:
  401020:       ff 35 e2 2f 00 00       pushq  0x2fe2(%rip)        # 404008
 <_GLOBAL_OFFSET_TABLE_+0x8>
  401026:       f2 ff 25 e3 2f 00 00    bnd jmpq *0x2fe3(%rip)        #
 404010 <_GLOBAL_OFFSET_TABLE_+0x10>
  40102d:       0f 1f 00                nopl   (%rax)
  401030:       f3 0f 1e fa             endbr64
  401034:       68 00 00 00 00          pushq  $0x0
  401039:       f2 e9 e1 ff ff ff       bnd jmpq 401020 <.plt>
  40103f:       90                      nop
  401040:       f3 0f 1e fa             endbr64
  401044:       68 01 00 00 00          pushq  $0x1
  401049:       f2 e9 d1 ff ff ff       bnd jmpq 401020 <.plt>
  40104f:       90                      nop
  401050:       f3 0f 1e fa             endbr64
  401054:       68 02 00 00 00          pushq  $0x2
  401059:       f2 e9 c1 ff ff ff       bnd jmpq 401020 <.plt>
  40105f:       90                      nop
  401060:       f3 0f 1e fa             endbr64
  401064:       68 03 00 00 00          pushq  $0x3
  401069:       f2 e9 b1 ff ff ff       bnd jmpq 401020 <.plt>
  40106f:       90                      nop
  401070:       f3 0f 1e fa             endbr64
  401074:       68 04 00 00 00          pushq  $0x4
  401079:       f2 e9 a1 ff ff ff       bnd jmpq 401020 <.plt>
  40107f:       90                      nop
  401080:       f3 0f 1e fa             endbr64
  401084:       68 05 00 00 00          pushq  $0x5
  401089:       f2 e9 91 ff ff ff       bnd jmpq 401020 <.plt>
  40108f:       90                      nop
  401090:       f3 0f 1e fa             endbr64
  401094:       68 06 00 00 00          pushq  $0x6
  401099:       f2 e9 81 ff ff ff       bnd jmpq 401020 <.plt>
  40109f:       90                      nop
  4010a0:       f3 0f 1e fa             endbr64
  4010a4:       68 07 00 00 00          pushq  $0x7
  4010a9:       f2 e9 71 ff ff ff       bnd jmpq 401020 <.plt>
  4010af:       90                      nop
  4010b0:       f3 0f 1e fa             endbr64
  4010b4:       68 08 00 00 00          pushq  $0x8
  4010b9:       f2 e9 61 ff ff ff       bnd jmpq 401020 <.plt>
  4010bf:       90                      nop
  4010c0:       f3 0f 1e fa             endbr64
  4010c4:       68 09 00 00 00          pushq  $0x9
  4010c9:       f2 e9 51 ff ff ff       bnd jmpq 401020 <.plt>
  4010cf:       90                      nop
  4010d0:       f3 0f 1e fa             endbr64
  4010d4:       68 0a 00 00 00          pushq  $0xa
  4010d9:       f2 e9 41 ff ff ff       bnd jmpq 401020 <.plt>
  4010df:       90                      nop
  4010e0:       f3 0f 1e fa             endbr64
  4010e4:       68 0b 00 00 00          pushq  $0xb
  4010e9:       f2 e9 31 ff ff ff       bnd jmpq 401020 <.plt>
  4010ef:       90                      nop
```

Fig 3.3: Procedure Linkage Table (PLT)

In dynamically linked executables, the PLT plays a crucial role in resolving function addresses on demand. When a program calls an external function for the first time, the PLT redirects execution

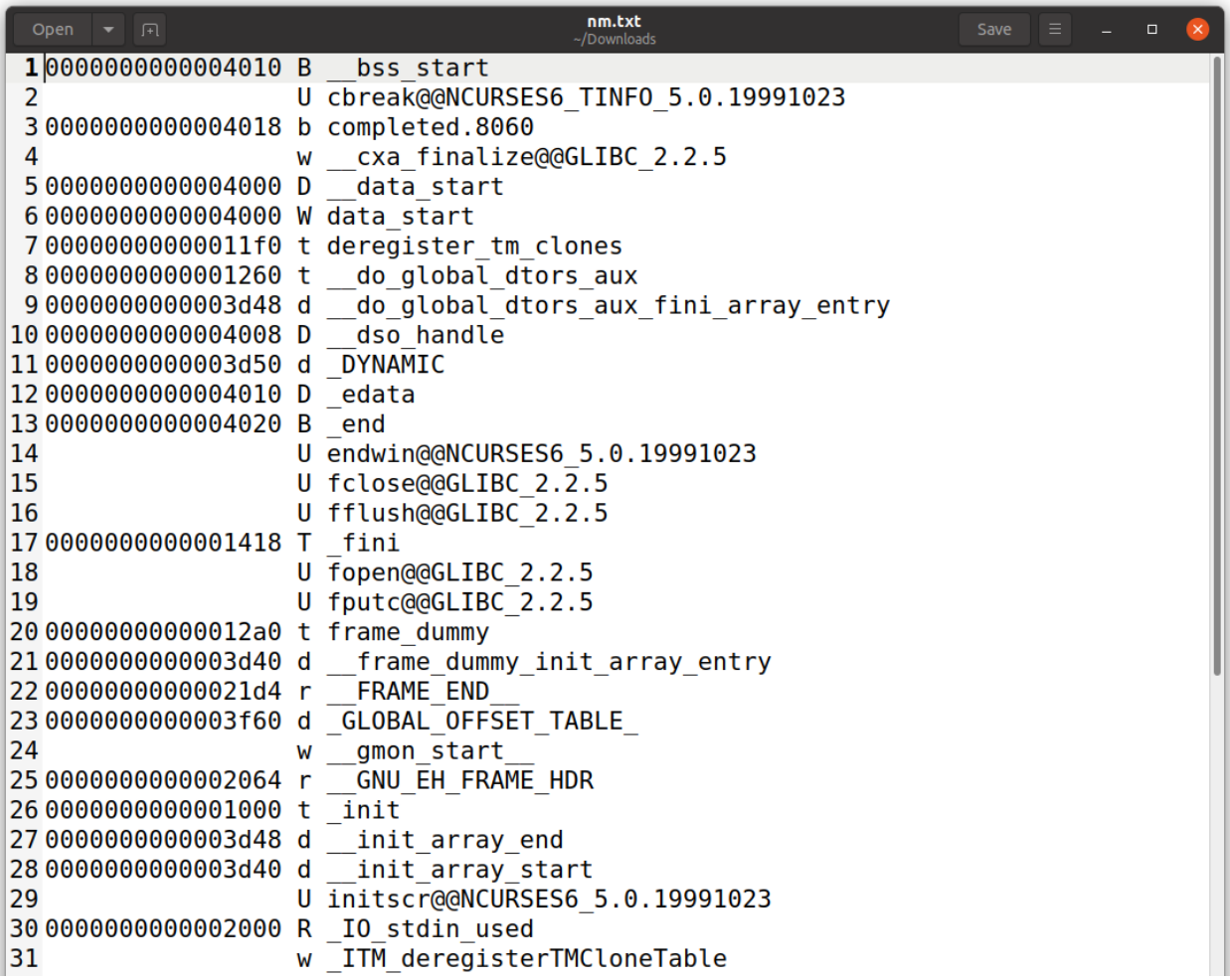
to the GOT, which initially contains a placeholder. This triggers the dynamic linker, which locates the correct function and updates the GOT for future calls.

In this keylogger program, functions like **`fopen`**, **`fputc`**, **`fflush`**, **`printf`**, and ncurses calls (e.g., **`initscr`**, **`printw`**, **`refresh`**) rely on the PLT to link to their actual addresses at runtime. Each PLT entry uses instructions like **`pushq`** and **`jmpq`** to manage this resolution process via the Global Offset Table.

A clear example is at address **`0x401034`**, where **`pushq $0x0`** places an index on the stack pointing to a function in the GOT. The following **`bnd jmpq`** transfers control to the PLT's first entry at **`0x401020`**, which calls the dynamic linker to resolve the actual function address from a shared library.

4. Inspecting Symbols, Dependencies, and Binary Contents:

a) Symbol Table Analysis



```
32                  w _ITM_registerTMCloneTable
33 0000000000001410 T __libc_csu_fini
34 00000000000013a0 T __libc_csu_init
35                  U __libc_start_main@@GLIBC_2.2.5
36 00000000000012a9 T log_keypress
37 00000000000012dc T main
38                  U noecho@@NCURSES6_5.0.19991023
39                  U printw@@NCURSES6_5.0.19991023
40                  U puts@@GLIBC_2.2.5
41 0000000000001220 t register_tm_clones
42 00000000000011c0 T _start
43 0000000000004010 B stdscr@@NCURSES6_TINFO_5.0.19991023
44 0000000000004010 D __TMC_END__
45                  U wgetch@@NCURSES6_5.0.19991023
46                  U wrefresh@@NCURSES6 5.0.19991023
```

Fig 4.1: Symbol Table Analysis

The nm command provides detailed information about the symbols in the binary, including memory addresses and symbol types. The symbol table consists of various symbols categorized by their characteristics. For example, B denotes uninitialized variables (i.e. **`bss_start`**). Undefined symbols, indicated by U (i.e. **`fput`**), resolved at runtime from external libraries.

Weak symbols, marked by W, references that the linker can replace with other definitions if needed. The T symbol represents text section entries.

The function **`log_keypress`** [line 7 of C] appears in the symbol table as (**`0x12a9`**), with T symbol. It is a user-defined function used to log key presses into a file and is directly compiled into the binary. The function **`printf`** [line 22 and 40] also appears in the symbol table as U **`printf@GLIBC_2.2.5`**. While **`log_keypress`** is compiled into the program, **`printf`** is linked from the C standard library at runtime via the Procedure Linkage Table.

b) Shared Library Dependencies

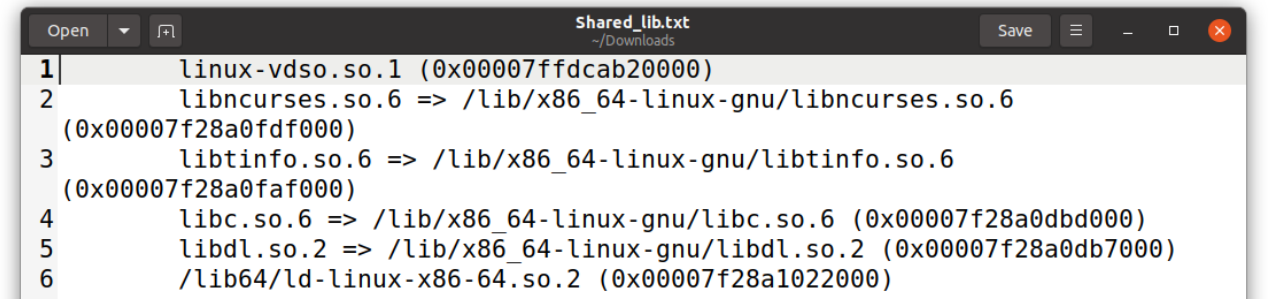


Fig 4.2: Shared Library Dependencies

The ldd command, which details the shared libraries required by a binary during execution. It lists the names and paths of libraries which are essential for the program's functionality. Along with the library names and file paths, the output includes the memory addresses at which these libraries are mapped during runtime. For example, **`libncurses.so.6`** is located in **`/lib/x86_64-linux-gnu/`**, and its address in memory is shown as **`0x00007f28a0dfd000`**.

These are required to support standard C functions such as **`printf`**, **`fopen`**, and **`fputc`**, as well as ncurses-based UI functions like **`initscr`** and **`printw`**, all of which appear in the provided C code between lines 11 and 36. If the program is compiled using the static flag, all necessary library code is embedded directly into the binary, removing the need for external shared libraries at runtime.

This increases the binary size but makes it more portable across systems lacking the required libraries.

c) Hexadecimal View

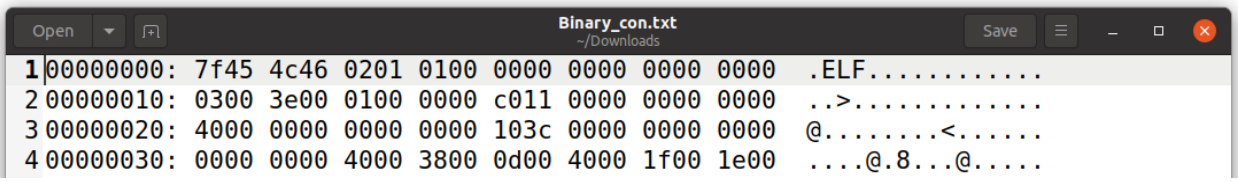


Fig 4.3: Hexadecimal View

The xxd command gives a hexadecimal representation of an ELF file. The first few bytes, **`7f 45 4c 46`**, are the ELF magic number. Following this, additional bytes provide metadata about the file, such as its architecture, byte order, and version. This header is automatically generated by the compiler when the C source file is compiled into an executable

The rest of the dump includes memory addresses and offsets that help describe the structure of the ELF binary. These addresses point to the various sections of the binary, such as the program and section headers, which define the layout of the executable in memory. This includes details on where the executable code, data, and other important segments are located in memory when the program is executed.

The right side of the dump shows the ASCII equivalent of the hexadecimal values, providing a human-readable view, offering clues about the program's functionality, such as function names or error messages.

Appendix A [https://github.com/gabriel-logan/Keylogger-full-working/tree/main]

This C program is a basic keylogger using **ncurses** to capture and log terminal keypresses to log.txt file. It initializes ncurses for direct key input, opens the log file (exiting on failure), displays an exit message, and enters a loop. Each keystroke is captured using **getch()** and logged via **log_keypress()**. The loop ends on ESC keypress. Finally, it closes the log file and ends the ncurses session, confirming the exit.
This code closely resembles the how the original keylogger works. which is having more the 200 lines of code.

```c
#include <stdio.h>
#include <stdlib.h>
#include <ncurses.h>        // library used for terminal-based UI handling
#define ESC_KEY 27      //represents the ASCII value of the ESC key

void log_keypress(FILE *logfile, int key) {     //function writes a pressed key to a log file
    fputc(key, logfile);
    fflush(logfile);
}

int main() {            //entry point of the program
    FILE *logfile;          //pointer to a file that will store keypress logs
    int key;                //Store the key that the user presses.

    initscr();      //initializes the ncurses
    cbreak();       //disables line buffering
    noecho();       //prevents keypresses from being displayed

    logfile = fopen("log.txt", "w");        //Open log file for writing
    if (logfile == NULL) {
        printf("Error opening log file.\n");
        endwin();
        return 1;
    }

    printw("Press ESC to exit.\n");         //Display instruction on screen
    refresh();

    while (1) {             //Infinite loop to capture keypresses
        key = getch();
        if (key == ESC_KEY) {
            break;
        }
        log_keypress(logfile, key);
    }

    fclose(logfile);        //Close the file and exit ncurses mode
    endwin();
    printf("Log saved successfully in log.txt.\n");
    return 0;
}

```

Appendix B

To run this C program you need to install few extra libraries of C and all other Instructions are given below.

1. **Introduction to ELF Format**

   $ sudo apt update
   $ sudo apt-get install libncurses5-dev
   $ gcc -o keylogger keylogger.c -lncurses

2. **ELF Header, Sections, and Segments**

   $ gcc -o keylogger keylogger.c -lncurses
   $ readelf -h keylogger > Elf_header.txt
   $ readelf --sections --wide keylogger > Elf_section.txt
   $ readelf -l --wide keylogger > Elf_segments.txt

3. Inspecting Symbol Table, GOT, and PLT

   $ gcc -o keylogger keylogger.c -no-pie -lncurses
   $ readelf --syms keylogger > Symbol_table.txt
   $ objdump -R keylogger
   $ objdump -j .plt -d keylogger > PLT.txt

4. **Inspecting Symbols, Dependencies, and Binary Contents**

   $ gcc -o keylogger keylogger.c -lncurses
   $ nm keylogger > nm.txt
   $ ldd keylogger > Shared_lib.txt
   $ xxd -l 64 keylogger > Binary_con.txt