Elf Loader

March 14, 2025

1 Introduction

The program in this section serves two purposes:

- Understanding ELF binaries: By compiling and analyzing a simple C program (hello.c), we will extract and examine the ELF header, section header table, program header table, and various other components.
- Demonstrating binfmt: By compiling Python scripts into .pyc files, we explore how they can be executed using binfmt_misc, allowing non-native binaries to run seamlessly.

2 Background

The Executable and Linkable Format (ELF) is the standard binary format used in Unix-like operating systems, including Linux, for executables, object files, shared libraries, and core dumps. An ELF loader is a critical component of the operating system responsible for loading these binaries into memory and preparing them for execution.

3 Programs

3.1 PART 1: Analyzing ELF Layout of Binaries

3.1.1 Compiling the C Program

To generate an ELF binary, we compile `hello.c` using:

```
gcc -o hello hello.c
```

3.1.2 Viewing the ELF Header

The ELF header provides essential metadata about the binary.

```
readelf -h hello
```

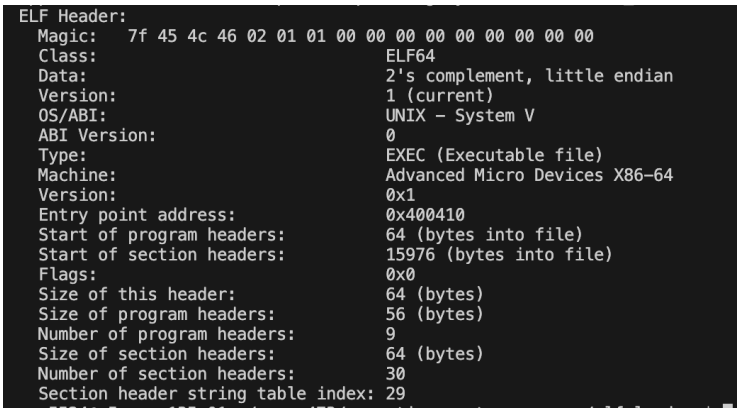


Figure 1: ELF Header: Shows architecture type, entry point, and ELF class (32-bit or 64-bit). (Fig 1).

The ELF Header provides crucial metadata about the binary format, helping the OS loader understand how to process the executable. Below is a breakdown of the key fields:

- **Magic Number (7f 45 4c 46)**: Identifies the file as an ELF binary. The first byte (0x7f) is a marker, followed by "ELF" in ASCII (45 4C 46).
- **Class (ELF64)**: Indicates that this is a 64-bit ELF binary.
- **Data (2's complement, little endian)**: Specifies the byte order (Little Endian format).
- **Version (1 - current)**: ELF format version.
- **OS/ABI (UNIX - System V)**: Target operating system. System V is the standard ABI used on Linux.
- **Type (EXEC - Executable file)**: Denotes this as an executable binary.
- **Machine (Advanced Micro Devices X86-64)**: Target architecture (AMD x86-64).
- **Entry Point Address (0x400410)**: The memory address where execution begins when the program is loaded.
- **Start of Program Headers (64 bytes into file)**: The offset where the program header table begins.
- **Start of Section Headers (15976 bytes into file)**: The offset where the section header table begins.
- **Size of ELF Header (64 bytes)**: The fixed size of the ELF header.

- **Size of Program Headers (56 bytes)**
- **Number of Program Headers (9)**
- **Size of Section Headers (64 bytes)**
- **Number of Section Headers (30)**
- **Section Header String Table Index (29)**

This header information is critical for the loader.

3.1.3 Viewing the Section Header Table

Lists all sections within the ELF binary.

```
readelf -S hello
```

The **Section Header Table** provides a structured view of the different sections that make up the ELF binary. Each section serves a specific role in the program's execution and linking process. Below is a breakdown of the key sections:

- **Metadata and Linking Sections**
  - `.interp` - Contains the program interpreter path.
  - `.note.ABI-tag` - Stores ABI-related metadata.
  - `.gnu.hash`, `.hash` - Used for symbol lookup in dynamic linking.
  - `.dynsym`, `.dynstr`, `.gnu.version` - Store symbol tables for dynamic linking.
- **Executable Code Sections**
  - `.text` - Contains the main program's executable code.
  - `.init`, `.fini` - Used for initialization and termination routines.
  - `.plt` - Holds the procedure linkage table for dynamic function calls.
- **Data and Read-Only Sections**
  - `.rodata` - Stores read-only data such as string literals.
  - `.data` - Stores initialized global and static variables.
  - `.bss` - Reserves space for uninitialized global variables.
- **Symbol and String Tables**
  - `.symtab` - Contains symbol information (not always present in stripped binaries).
  - `.strtab` - Holds string data for symbol names.
  - `.shstrtab` - Stores section names.

3.1.4 Viewing the Program Header Table

Displays segments loaded into memory.

```
readelf -l hello
```

The Program Header Table describes how the OS loader should map the executable into memory. It defines segments, their locations, sizes, and permissions. This ELF file has 9 program headers, each serving a specific role in execution.

- **PHDR (Program Header Table)** - Defines the location of the program header table itself, helping the ELF loader find other headers.
- **INTERP (Interpreter Path)** - Specifies the dynamic linker used to load shared libraries. In this case, it is `/lib64/ld-linux-x86-64.so.2`.
- **LOAD (Code and Data Segments)**
  - First `LOAD` segment (RX) - Contains executable code (`.text` section).
  - Second `LOAD` segment (RW) - Stores writable data (`.data`, `.bss`).
- **DYNAMIC (Dynamic Linking Information)** - Holds dynamic linking data required at runtime.
- **NOTE (Metadata Section)** - Stores metadata such as ABI information.
- **GNU_EH_FRAME (Exception Handling)** - Contains data for stack unwinding during exceptions.
- **GNU_STACK (Stack Permissions)** - Defines stack properties, marked as **Read/Write (RW) but not executable (NX)** for security.
- **GNU_RELRO (Relocation Read-Only)** - Marks certain sections as **read-only after relocation** to protect against exploits.

3.1.5 Sections and Segments Mapping

Shows how sections map to segments.

```
readelf -l hello
```

The section-to-segment mapping shows how sections (logical units in an ELF file) are grouped into segments (units mapped into memory by the OS loader).

3.1.6 Extracting the Symbol Table

Displays function and variable symbols.

```
readelf -s hello
```

The **symbol table** in an ELF binary provides a list of all symbols (functions, variables, and special markers) used in the program. The key symbol tables in this output are:

- `.dynsym` **(Dynamic Symbol Table)**
  - Contains **symbols required for dynamic linking**.
  - Includes external functions like `printf` and `__libc_start_main`, which are linked dynamically from **GLIBC**.
  - `__gmon_start__` is a weak symbol often related to **profiling/debugging**.
- `.symtab` **(Full Symbol Table)**
  - Contains **all symbols** in the executable, including:
    - **Functions**: `main`, `_start`, `deregister_tm_clones`, etc.
    - **Global Variables**: `_edata`, `_bss_start`, `_end`.
    - **Sections and markers**: `.init`, `.plt`, `.text`.
  - Includes **file references** such as `hello.c` and `crtstuff.c`, indicating where symbols originated.

3.1.7 Extracting the String Table

Lists string literals stored in the binary.

```
readelf -p .strtab hello
```

The String Table in an ELF binary stores symbol names, section names, and dynamic linking information in a compact format. Instead of storing full names in the symbol table, ELF uses offsets into the string table to save space. The section string table names sections, while the symbol string tables name functions and variables.

3.1.8 Viewing Relocation Entries

Lists relocations used for dynamic linking.

```
readelf -r hello
```

The relocation table in an ELF binary contains entries that help resolve symbol addresses at runtime. These entries modify specific memory locations to contain correct addresses before the program executes.

- `.rela.dyn` **(Relocation for Dynamic Linking)**
  - Used for **global symbols** that must be resolved at runtime.
  - Entries:

    - `__libc_start_main@GLIBC_2.2.5` - The entry point of the C runtime.
    - `__gmon_start__` - Used for profiling/debugging.

- `.rela.plt` **(Relocation for Procedure Linkage Table - PLT)**
  - Used for dynamically linked functions in shared libraries.
  - Entry:
    - `printf@GLIBC_2.2.5` - This function call needs to be resolved at runtime.

3.2 PART 2: Demonstrating binfmt with Compiled Python Program

3.2.1 Compiling the Python Script

We compile the Python script into bytecode using:

```
python -m py_compile example.py
```

3.2.2 Registering Python Bytecode with binfmt_misc

To enable direct execution of compiled Python files, we register them in binfmt:

```
sudo mount binfmt_misc -t binfmt_misc /proc/sys/fs/binfmt_misc
echo ':python_magic:E::\x03\xf3\x0d\x0a::/usr/bin/python3:' | sudo tee /proc/sys
```

3.2.3 Making Python Bytecode Executable

After registration, we make `example.pyc` executable:

```
chmod +x example.pyc
./example.pyc
```

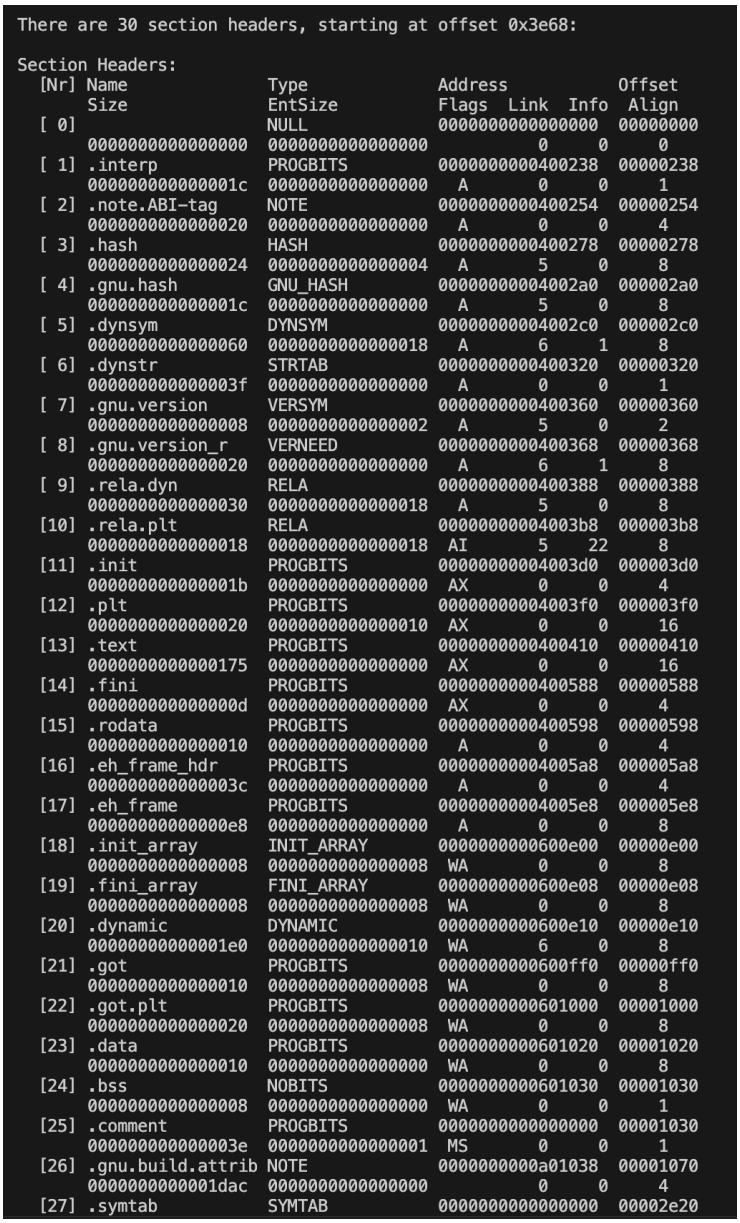
4 Takeaway

ELF Analysis:

- Read ELF metadata using readelf and objdump.
- Understand headers, segments, symbols, and relocation.

binfmt_misc:

- Enables execution of compiled Python bytecode without explicitly invoking python.
- Registers a magic number to map .pyc files to Python.



```
There are 30 section headers, starting at offset 0x3e68:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .hash             HASH             0000000000400278  00000278
       0000000000000024  0000000000000004   A       5     0     8
  [ 4] .gnu.hash         GNU_HASH         00000000004002a0  000002a0
       000000000000001c  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000004002c0  000002c0
       0000000000000060  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000400320  00000320
       000000000000003f  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           0000000000400360  00000360
       0000000000000008  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000400368  00000368
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000400388  00000388
       0000000000000030  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             00000000004003b8  000003b8
       0000000000000018  0000000000000018  AI       5    22     8
  [11] .init             PROGBITS         00000000004003d0  000003d0
       000000000000001b  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         00000000004003f0  000003f0
       0000000000000020  0000000000000010  AX       0     0     16
  [13] .text             PROGBITS         0000000000400410  00000410
       0000000000000175  0000000000000000  AX       0     0     16
  [14] .fini             PROGBITS         0000000000400588  00000588
       000000000000000d  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS         0000000000400598  00000598
       0000000000000010  0000000000000000   A       0     0     4
  [16] .eh_frame_hdr     PROGBITS         00000000004005a8  000005a8
       000000000000003c  0000000000000000   A       0     0     4
  [17] .eh_frame         PROGBITS         00000000004005e8  000005e8
       00000000000000e8  0000000000000000   A       0     0     8
  [18] .init_array       INIT_ARRAY       0000000000600e00  00000e00
       0000000000000008  0000000000000008  WA       0     0     8
  [19] .fini_array       FINI_ARRAY       0000000000600e08  00000e08
       0000000000000008  0000000000000008  WA       0     0     8
  [20] .dynamic          DYNAMIC          0000000000600e10  00000e10
       00000000000001e0  0000000000000010  WA       6     0     8
  [21] .got              PROGBITS         0000000000600ff0  00000ff0
       0000000000000010  0000000000000008  WA       0     0     8
  [22] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000020  0000000000000008  WA       0     0     8
  [23] .data             PROGBITS         0000000000601020  00001020
       0000000000000010  0000000000000000  WA       0     0     8
  [24] .bss              NOBITS           0000000000601030  00001030
       0000000000000008  0000000000000000  WA       0     0     1
  [25] .comment          PROGBITS         0000000000000000  00001030
       000000000000003e  0000000000000001  MS       0     0     1
  [26] .gnu.build.attrib NOTE             0000000000a01038  00001070
       0000000000001dac  0000000000000000           0     0     4
  [27] .symtab           SYMTAB           0000000000000000  00002e20
```

Figure 2: Section Header Table: Shows sections such as .text, .data, .bss. 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)

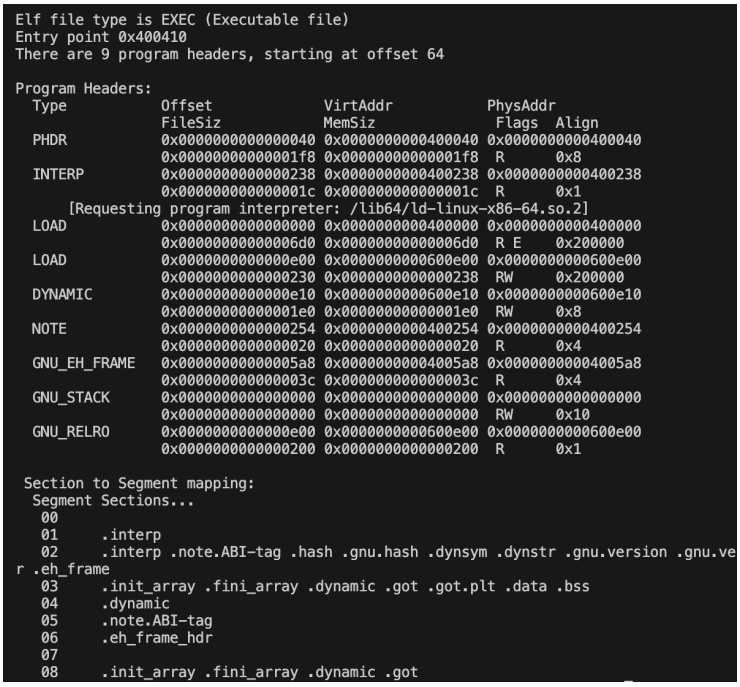
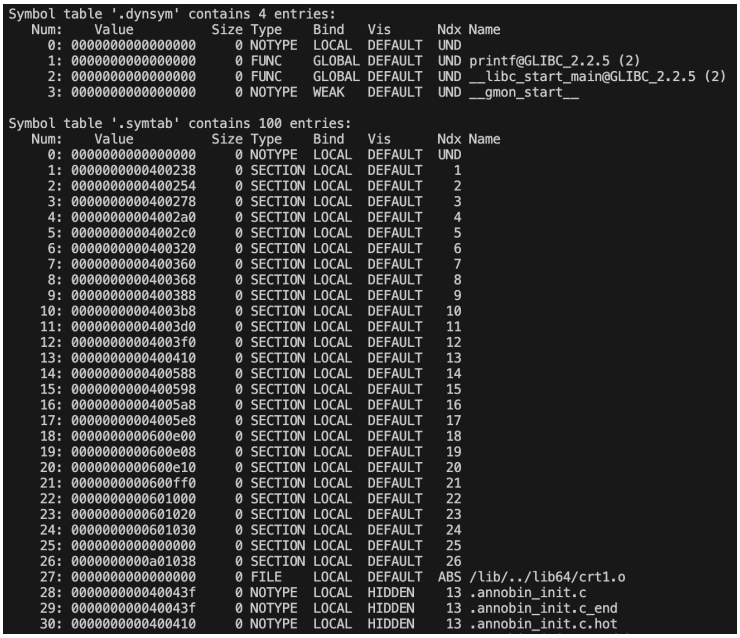


Figure 3: Program Header Table: Lists executable segments and memory protection attributes.

Figure 4: Mapping between sections and segments in ELF.



```
Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

Symbol table '.symtab' contains 100 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000400238     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000400254     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000400278     0 SECTION LOCAL  DEFAULT    3
     4: 00000000004002a0     0 SECTION LOCAL  DEFAULT    4
     5: 00000000004002c0     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000400320     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000400360     0 SECTION LOCAL  DEFAULT    7
     8: 0000000000400368     0 SECTION LOCAL  DEFAULT    8
     9: 0000000000400388     0 SECTION LOCAL  DEFAULT    9
    10: 00000000004003b8     0 SECTION LOCAL  DEFAULT   10
    11: 00000000004003d0     0 SECTION LOCAL  DEFAULT   11
    12: 00000000004003f0     0 SECTION LOCAL  DEFAULT   12
    13: 0000000000400410     0 SECTION LOCAL  DEFAULT   13
    14: 0000000000400588     0 SECTION LOCAL  DEFAULT   14
    15: 0000000000400598     0 SECTION LOCAL  DEFAULT   15
    16: 00000000004005a8     0 SECTION LOCAL  DEFAULT   16
    17: 00000000004005e8     0 SECTION LOCAL  DEFAULT   17
    18: 0000000000600e00     0 SECTION LOCAL  DEFAULT   18
    19: 0000000000600e08     0 SECTION LOCAL  DEFAULT   19
    20: 0000000000600e10     0 SECTION LOCAL  DEFAULT   20
    21: 0000000000600ff0     0 SECTION LOCAL  DEFAULT   21
    22: 0000000000601000     0 SECTION LOCAL  DEFAULT   22
    23: 0000000000601020     0 SECTION LOCAL  DEFAULT   23
    24: 0000000000601030     0 SECTION LOCAL  DEFAULT   24
    25: 0000000000000000     0 SECTION LOCAL  DEFAULT   25
    26: 000000000a01038      0 SECTION LOCAL  DEFAULT   26
    27: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /lib/../lib64/crt1.o
    28: 000000000040043f     0 NOTYPE  LOCAL  HIDDEN    13 .annobin_init.c
    29: 000000000040043f     0 NOTYPE  LOCAL  HIDDEN    13 .annobin_init.c_end
    30: 0000000000400410     0 NOTYPE  LOCAL  HIDDEN    13 .annobin_init.c.hot
```

Figure 5: Symbol Table: Shows function symbols, global variables, and linkage information.

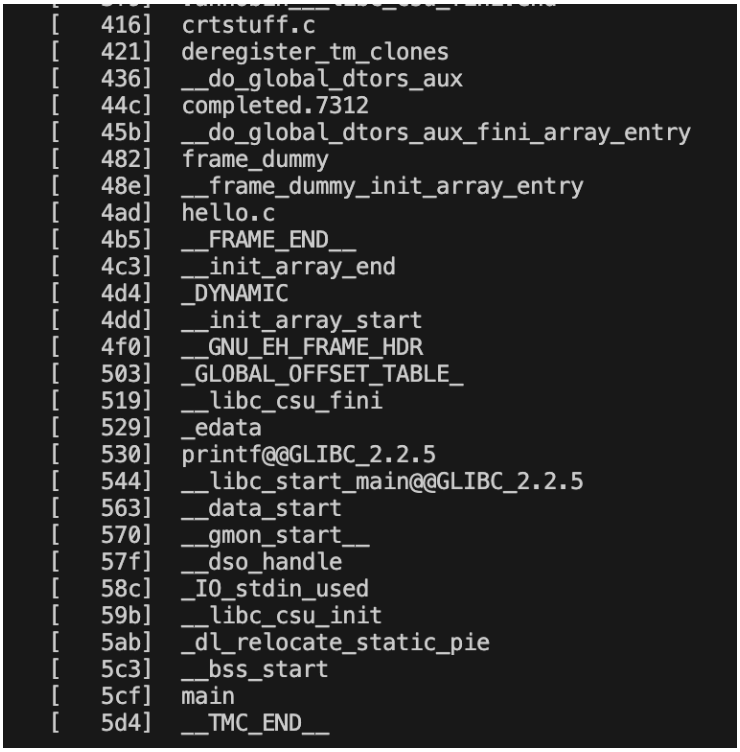


Figure 6: String Table: Displays strings embedded within the ELF binary.

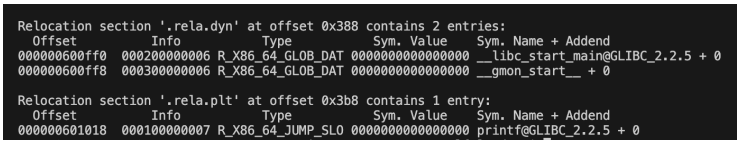


Figure 7: Relocation Table: Shows addresses modified at runtime.

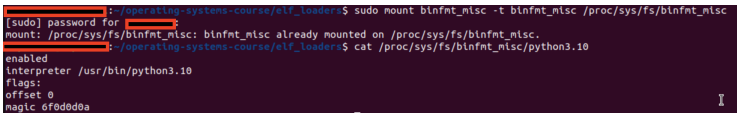


Figure 8: Mount.