Binary Analysis and Secure Coding

Federico Conti

2025/26

Contents

Introduction

The goal of this assignment is to familiarize yourself with the ELF file format. After unzipping `ELF_files.zip`, you will find several file for your analysis.

Each file hides one *flag*, a string in the format `BASC{...}`, where the content inside the braces is at least eight characters long. In particular, `BASC{3TON5}`, which is something you may come across, is not the intended flag for the corresponding file.

ELF 1

By inspecting the Program Headers, an unusual mapping between segments and sections is visible:

```
readelf -l elf1

## Output
 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   .....
   03     .init .plt .plt.got .plt.sec .text B A S C { s 3 c T i 0 N 5 } .fini
...
```

Here, the flag is embedded directly in the section list of one of the program segments:

ELF2

```
file elf2
  # Output
  elf2: ELF 32-bit LSB executable, ARM,...
```

Since this is an ARM binary, it cannot be executed directly on an x86 host. However, using QEMU for emulation works as expected:

```
qemu-arm ./elf2
  # Output
  BASC{ARMed_&_d4ng3r0uS}
```

ELF3

Attempting to execute elf3 results in an error:

```
./elf3
  #Output
  bash: ./elf3: cannot execute binary file: Exec format error
```

Inspecting the file reveals that it is not recognized as a valid ELF binary:

```
file elf3
  #Output
  elf3: data

readelf -h elf3
  #Output
  readelf: Error: Not an ELF file - it has the wrong magic bytes at the start
```

The magic bytes are indeed corrupted:

```
 xxd -l 32 elf3
  #Output
  00000000: 7f65 6c66 0201 0100 0000 0000 0000 0000  .elf...........
  00000010: 0300 3e00 0100 0000 b010 0000 0000 0000  ..>............
```

Here, the first byte (0x65 = ‘e’) should be 0x45 = ‘E’. We can repair the ELF header as follows:

```
printf '\x7fELF' | dd of=elf3 bs=1 seek=0 count=4 conv=notrunc
```

After fixing the header the magic bytes are correct:

```
xxd -l 32 elf3
  #Output
  00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF.....
```

The file is now executable:

```
./elf3
  #Output
  BASC{cAs3_maTT3rS}
```

## ELF4

Running elf4 causes a segmentation fault:

```
./elf4
 # Output
 Segmentation fault (core dumped)
```

Debugging with GDB+GEF:

```
gef  break *0x401c60
  Breakpoint 1 at 0x401c60 #entry point
gef  continue
Continuing.
  # #Output
  Program terminated with signal SIGSEGV, Segmentation fault.
  The program no longer exists.
```

The memory map shows that the text segment is not executable:

```
gef  vmm
[ Legend:  Code | Stack | Heap ]
Start              End                Offset             Perm Path
0x0000000000400000 0x00000000004bc000 0x0000000000000000 r-- ...
0x00000000004bd000 0x00000000004c3000 0x00000000000bc000 rw- ...
0x00000000004c3000 0x00000000004c4000 0x0000000000000000 rw- [heap]
0x00007ffff7ff9000 0x00007ffff7ffd000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000000000 r-x [vdso]
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
```

Inspecting the Program Headers confirms that the .text section’s segment lacks the executable bit (E): The entry point (0x401c60) lies inside this segment, which explains the crash.

```
readelf -lW elf4
  # Output
  Program Headers:
    Type  Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
    LOAD  0x000000 0x0000000000400000 0x0000000000400000 0x000518 0x000518 R   0x1000
    LOAD  0x001000 0x0000000000401000 0x0000000000401000 0x09366d 0x09366d R   0x1000 #(.text)
    ...
```

From the ELF header:

```
readelf -h elf4
  # Output
  Start of program headers:          64 (bytes into file) # 0x40
  Size of program headers:           56 (bytes) # 0x38
  Number of program headers:         10
```

- `e_phoff` = 64 = 0x40 → the first Program Header starts at offset 0x40 in the file.
- Each Program Header is 56 bytes long = 0x38

Considering that the:

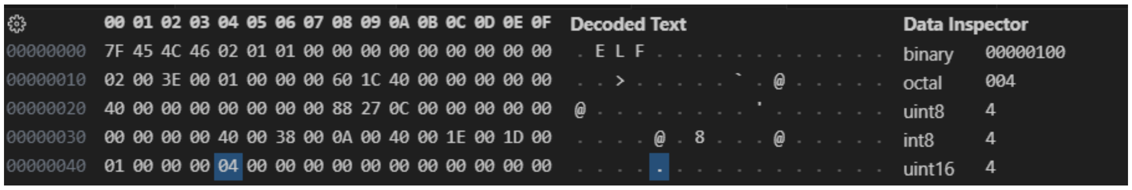
- In the ELF64 format, each Program Header has this structure:

Field	Size	Offset relative to Program Header
`p_type`	4 bytes	+0x00
`p_flags`	4 bytes	+0x04
`p_offset`	8 bytes	+0x08
`p_vaddr`	8 bytes	+0x10

- The possible values for `p_flags` are:

Flag	Decimal Value	Hex	Meaning
R	4	0x4	Read
W	2	0x2	Write
E	1	0x1	Execute

- So the byte to change is at offset: (0x40 + 0x38) + 0x04 = 0x7C
  - Currently the field is worth 0x4 (Read only)



```
         00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text                      Data Inspector
00000000 7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00  . E L F . . . . . . . . . . . .   binary  00000100
00000010 02 00 3E 00 01 00 00 00 60 1C 40 00 00 00 00 00  . . > . . . . . ` . @ . . . . .   octal   004
00000020 40 00 00 00 00 00 00 00 88 27 0C 00 00 00 00 00  @ . . . . . . . . ' . . . . . .   uint8   4
00000030 00 00 00 00 40 00 38 00 0A 00 40 00 1E 00 1D 00  . . . . @ . 8 . . . @ . . . . .   int8    4
00000040 01 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .   uint16  4
```

Using hex editor or dd, we change it to 0x5 (Read + Execute):

```
printf '\x05' | dd of=elf4 bs=1 seek=$((0x7C)) count=1 conv=notrunc

./elf4
  # Output
  BASC{no_eXec_no_party} # success!
```