Extracting Features from Executable Binary Files

Marcus Botacin[1]

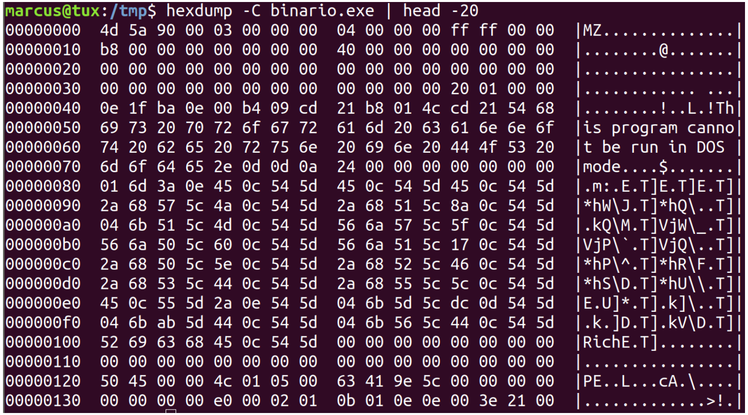
[1]Texas A&M University (TAMU)
botacin@tamu.edu

2022

Why investigating executable binaries?

- Classify malware vs. goodware.
- Forensic analyses.
- Version identification.
- Software phylogeny.
    - What if you do not have the source code?

Does it work?



```
marcus@tux:/tmp$ hexdump -C binario.exe | head -20
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ..............|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 20 01 00 00  |............ ...|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.......|
00000080  01 6d 3a 0e 45 0c 54 5d  45 0c 54 5d 45 0c 54 5d  |.m:.E.T]E.T]E.T]|
00000090  2a 68 57 5c 4a 0c 54 5d  2a 68 51 5c 8a 0c 54 5d  |*hW\J.T]*hQ\..T]|
000000a0  04 6b 51 5c 4d 0c 54 5d  56 6a 57 5c 5f 0c 54 5d  |.kQ\M.T]VjW\_.T]|
000000b0  56 6a 50 5c 60 0c 54 5d  56 6a 51 5c 17 0c 54 5d  |VjP\`.T]VjQ\..T]|
000000c0  2a 68 50 5c 5e 0c 54 5d  2a 68 52 5c 46 0c 54 5d  |*hP\^.T]*hR\F.T]|
000000d0  2a 68 53 5c 44 0c 54 5d  2a 68 55 5c 5c 0c 54 5d  |*hS\D.T]*hU\\.T]|
000000e0  45 0c 55 5d 2a 0e 54 5d  04 6b 5d 5c dc 0d 54 5d  |E.U]*.T].k]\..T]|
000000f0  04 6b ab 5d 44 0c 54 5d  04 6b 56 5c 44 0c 54 5d  |.k.]D.T].kV\D.T]|
00000100  52 69 63 68 45 0c 54 5d  00 00 00 00 00 00 00 00  |RichE.T]........|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000120  50 45 00 00 4c 01 05 00  63 41 9e 5c 00 00 00 00  |PE..L...cA.\....|
00000130  00 00 00 00 e0 00 02 01  0b 01 0e 0e 00 3e 21 00  |.............>!.|
```

How do executable files work?

Definition

- Executable binaries must follow a pre-defined structure to instruct OS about how to load them.

OS must know:

- How many *bytes* it should allocate to load a file.
- What the execution entry point is.

Executable binaries are OS-dependent

Similar, but different...

- Overall, executable binary formats answer the same OS “questions”, but their fields have different sizes and are placed in different *structs*.

OS and Formats

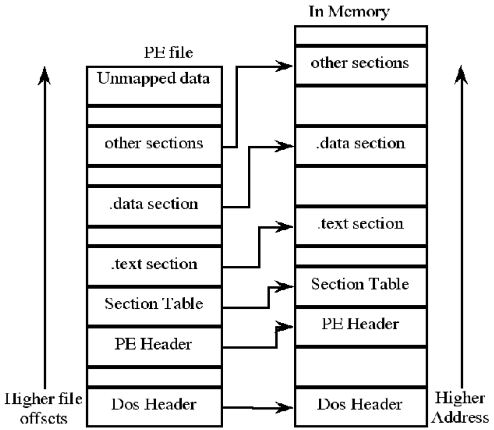
- **Linux**: Executable and Linkable Format (ELF).
- **Windows**: Portable Executable (PE).[a]

[a] **Deep Dive:** Wikipedia knows everything about these formats!

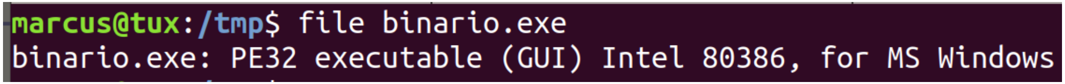
The PE format

Basic Structure

- **Headers**: Answers and Pointers to data.
- **Sections**: The data.



How the OS interprets an executable binary file



```
marcus@tux:/tmp$ file binario.exe
binario.exe: PE32 executable (GUI) Intel 80386, for MS Windows
```

How you should interpret

```c
typedef struct {
    uint16_t e_magic;
    uint16_t e_cblp;
    uint16_t e_cp;
    uint16_t e_crlc;
    uint16_t e_cparhdr;
    uint16_t e_minalloc;
    uint16_t e_maxalloc;
    uint16_t e_ss;
    uint16_t e_sp;
    uint16_t e_csum;
```

```c
    uint16_t e_ip;
    uint16_t e_cs;
    uint16_t e_lfarlc;
    uint16_t e_ovno;
    uint16_t e_res[4];
    uint16_t e_oemid;
    uint16_t e_oeminfo;
    uint16_t e_res2[10];
    uint32_t e_lfanew;
} IMAGE_DOS_HEADER;
```

What's inside the sections?

A generalization

- **.text**: Instructions.
- **.data**: Initialized data (constants?).
- **.bss**: Non-initialized data (variables?)
- **Others**: Binaries might have different number and names of sections.

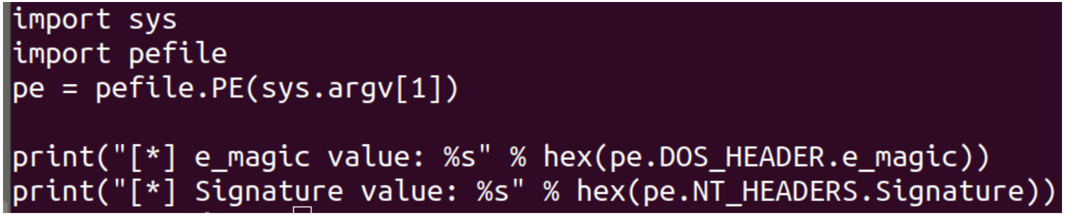
What can we discover?

- If the *magic* `MZ` is identified in a buffer: Code Injection.
- If a *timestamp* older than 1970 is identified: Tampered Binary.
- If the *checksum* mismatches: Tampered Binary.
- If sections have `RWX` permissions: Self-Modifying Code (SMC).

How to handle executable binaries?

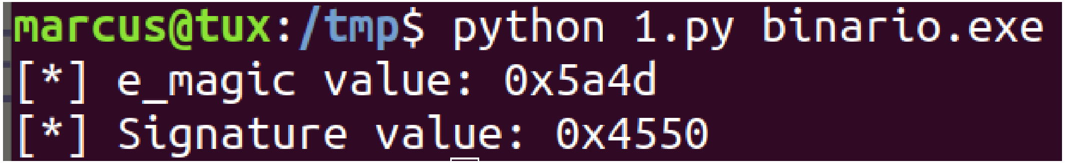
- **ELF:** `pyelftools` (`https://github.com/eliben/pyelftools`)
- **PE:** `pefile` (`https://github.com/erocarrera/pefile`)
- **Multi:** `lief` (`https://lief.quarkslab.com/`)

pefile Example (1/3)



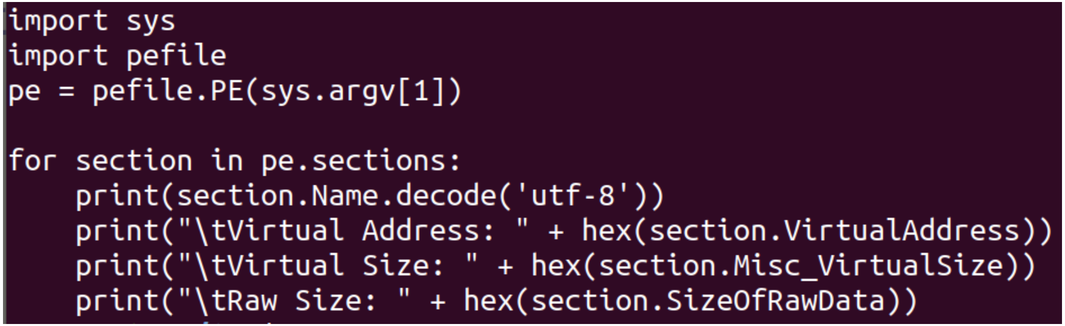
```
import sys
import pefile
pe = pefile.PE(sys.argv[1])

print("[*] e_magic value: %s" % hex(pe.DOS_HEADER.e_magic))
print("[*] Signature value: %s" % hex(pe.NT_HEADERS.Signature))
```



```
marcus@tux:/tmp$ python 1.py binario.exe
[*] e_magic value: 0x5a4d
[*] Signature value: 0x4550
```

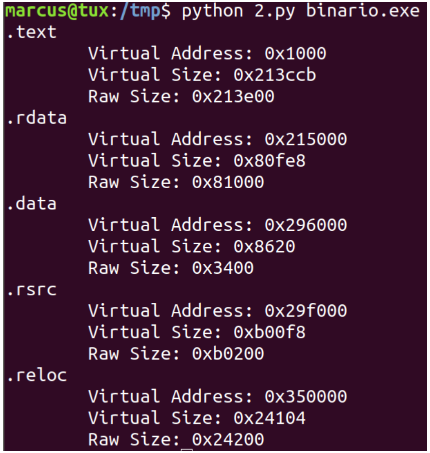
pefile Example (2/3)



```
import sys
import pefile
pe = pefile.PE(sys.argv[1])

for section in pe.sections:
    print(section.Name.decode('utf-8'))
    print("\tVirtual Address: " + hex(section.VirtualAddress))
    print("\tVirtual Size: " + hex(section.Misc_VirtualSize))
    print("\tRaw Size: " + hex(section.SizeOfRawData))
```

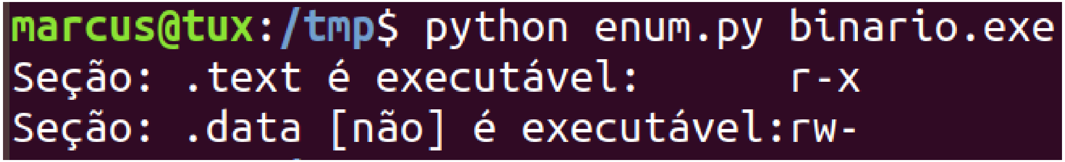
pefile Example (3/3)



```
marcus@tux:/tmp$ python 2.py binario.exe
.text
        Virtual Address: 0x1000
        Virtual Size: 0x213ccb
        Raw Size: 0x213e00
.rdata
        Virtual Address: 0x215000
        Virtual Size: 0x80fe8
        Raw Size: 0x81000
.data
        Virtual Address: 0x296000
        Virtual Size: 0x8620
        Raw Size: 0x3400
.rsrc
        Virtual Address: 0x29f000
        Virtual Size: 0xb00f8
        Raw Size: 0xb0200
.reloc
        Virtual Address: 0x350000
        Virtual Size: 0x24104
        Raw Size: 0x24200
```

Try Yourself (1/2)

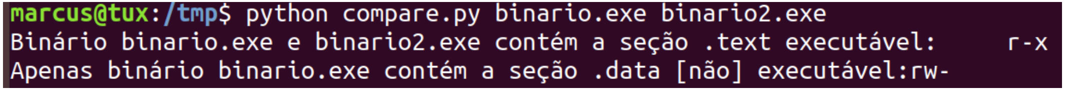
- Enumerate the executable sections of a given PE binary file.



```
marcus@tux:/tmp$ python enum.py binario.exe
Seção: .text é executável:      r-x
Seção: .data [não] é executável:rw-
```

Try Yourself (2/2)

- Compare the sections of two given PE binary files.



```
marcus@tux:/tmp$ python compare.py binario.exe binario2.exe
Binário binario.exe e binario2.exe contém a seção .text executável:     r-x
Apenas binário binario.exe contém a seção .data [não] executável:rw-
```