LAB 15: MALWARE ANALYSIS

Lab Requirements

1. One Linux VM
2. Python 3.9

Content

Part I: Dataset and Resources

STEP 1: The malware samples used in this lab are available in the Code and Data page of the website of the book "Malware Data Science: Attack Detection and Attribution" [https://www.malwaredatascience.com/code-and-data]. Download the .zip file and unzip it in a folder of your choice. I remind you to use a Linux virtual machine as Windows malware samples will be flagged by Windows OS and might be accordingly deleted.

Part II: Malicious Software

STEP 2: Malware stands for malicious software, which is any software created and used by threat actors to gain access or cause harm to your system. There are two types of malware analysis:

- **Static analysis:** Static analysis is performed by analyzing a program file's code, graphical images, strings, and other in-file stored information.
- **Dynamic analysis:** Dynamic analysis consist of the running the malware in a safe and isolate environment to analyze its behavior.

STEP 3: Most common types of malware:

- **VIRUS**

- A **virus is** a computer code that can replicate itself by modifying other files or programs. The inserted code is capable of further replication. Replication requires user assistance, such as clicking on an email attachment, sharing a USB drive, among others.
- A **virus** has the following **phases**: 1) dormant phase, propagation phase, 3) triggering phase, 4) action phase.
- **Types**: program virus (infects executables), macro or document virus (infects documents), boot sector virus, encrypted virus, polymorphic virus, and metamorphic virus.

- **TROJAN HORSES**
  - **A trojan is** a malware program that appears to be benign, but which is also doing some malicious actions.
- **COMPUTER WORMS**
  - **A worm is** a malware that spreads copies of itself without the need of infecting other programs. Worms, like viruses, spread by self-replication. However, viruses infect other files/programs and worms don't.
- **ROOTKITS**
  - They are software programs designed to gain administrator-level (Windows) or root-level (Linux) over an operating system without being detected.
  - A **rootkit** might work in one of the following **two modes**: **User-mode**: User-mode rootkits **alter** the system utilities or libraries. This type of malicious activity can be easily detected using hash functions. Other user-mode rootkits inject code in other running usermode process to alter its behavior. **Kernel-mode**: This type of rootkits works at the lowest level of the operating system. They are, therefore, difficult to detect. In Windows, Kernel rootkits are always loaded as device drivers. As such, they can upload arbitrary code to the kernel.
- **BOTNET**
  - A large-scale attack can be performed by a network of compromised computers, known as a botnet and controlled by the bot herder. A botnet can have a truly massive size of several millions of compromised computers, known as zombies. It is conjectured that one quarter of the computers connected to the Internet are part of some botnet. A bot software is installed on a zombie via a worm, a Trojan horse, or any type of malware software. The zombie contacts a central control server to request commands. Accordingly, the bot herder can issue commands to control the botnet.
- **ADWARE**
  - It **is a software** that displays advertisements on a user's screen against their consent. When a user visits an infected web page, opens an infected e-mail, installs a freeware that

has an adware embedded in a Trojan horse, an adware is installed on user's machine and periodically displays pop-up advertisements.

- **SPYWARE**
  - A **spyware is** a software installed on user's computer without his consent to gather information about the user, his communication, or his computer usage without his consent.
  - **Types of spyware: Keylogging**, **screen capturing**, **tracking cookies**: cookies are used to maintain state between user's multiple visits, **data harvesting**: searching on the infected computer for personal data.
- **RANSOMWARE**
  - It is a malware that encrypts a device's data. The keys used to encrypt the data are exchanged for money.

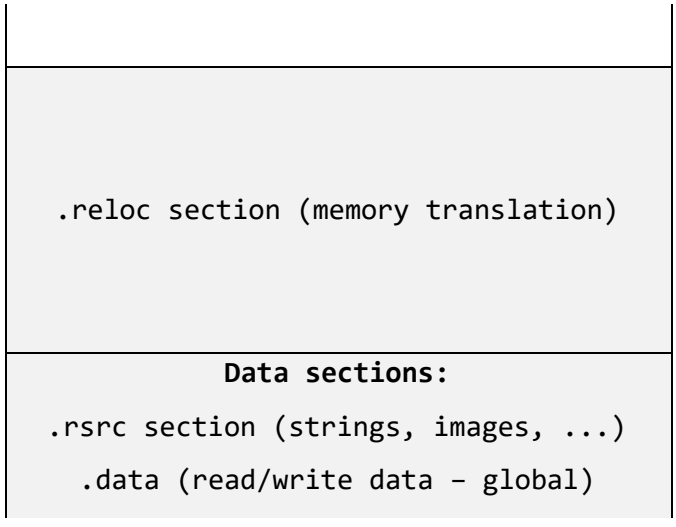
Part III: Portable Executable File Format (PE File Format)

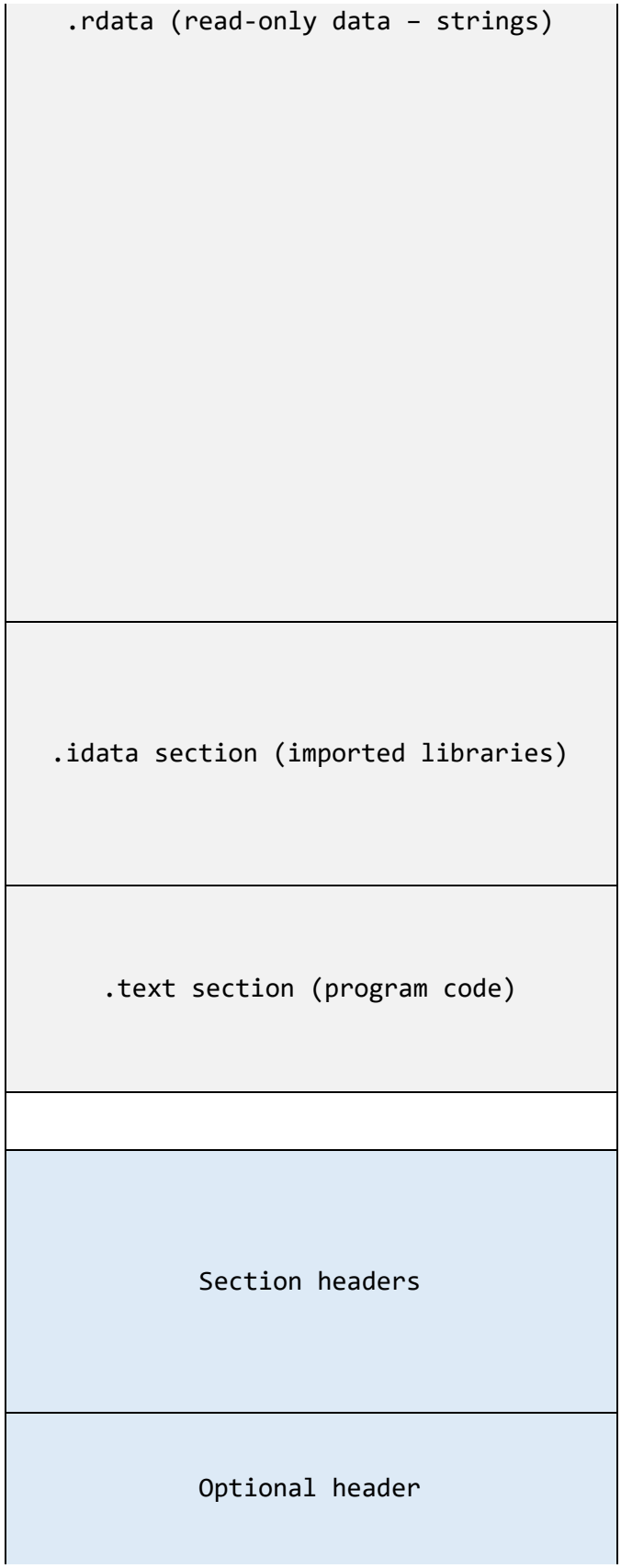
STEP 3: PE file format is used by Windows operating systems for several file types, including:

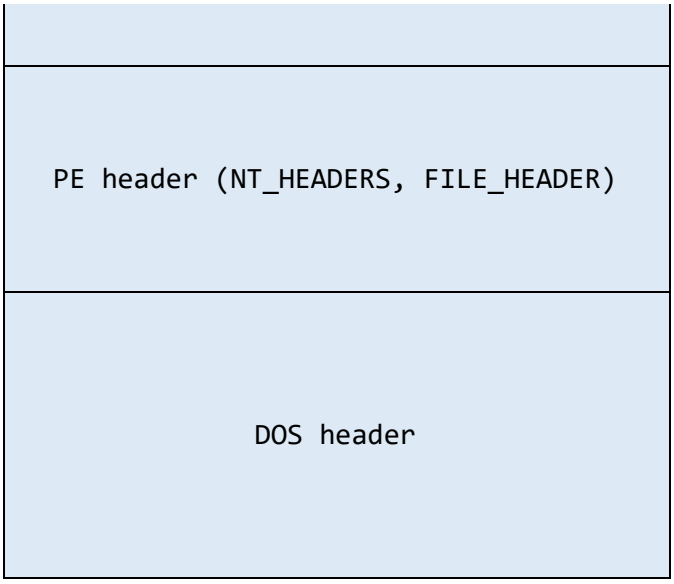
- Executable files (.exe)
- Dynamic link library (DLL),
- ActiveX control (.ocx),
- System files (.sys) and kernel drivers (.drv)

A good understanding of the PE file structure is essential for static malware analysis. The PE format includes information to instruct the operating system how to load the program into memory, in addition to several sections that contain the executable's actual data.

STEP 4: PE file format is used by Windows




.rdata (read-only data – strings)
.idata section (imported libraries)
.text section (program code)
Section headers
Optional header



1. **DOS HEADER:** It is present for compatibility reasons. This header contains the values: `e_magic`, `e_cblp`, `e_cp`, `e_crlc`, `e_cparhdr`, `e_minalloc`, `e_maxalloc`, `e_ss`, `e_sp`, `e_csum`, `e_ip`, `e_cs`, `e_lfarlc`, `e_ovno`, `e_res`, `e_oemid`, `e_oeminfo`, `e_res2`, `e_lfanew`. The two important values are `e_magic` and `e_lfanew`: `e_magic` is set to the value 0x5A4D (MZ, the initials of Mark Zbikowski, one of the early architects of MS-DOS) and `e_lfanew` contains the offset of the PE header.

```
import pefile

# Load the malware sample
pe = pefile.PE('./ch1/ircbot.exe')

# Print all the information of the PE file
# In the following, we will analyze each of these values pe.print_info()

    ----------DOS_HEADER----------
    [IMAGE_DOS_HEADER]
    0x0        0x0   e_magic:                       0x5A4D
    0x2        0x2   e_cblp:                        0x90
    0x4        0x4   e_cp:                          0x3
    0x6        0x6   e_crlc:                        0x0
    0x8        0x8   e_cparhdr:                     0x4
    0xA        0xA   e_minalloc:                    0x0
    0xC        0xC   e_maxalloc:                    0xFFFF
    0xE        0xE   e_ss:                          0x0
    0x10       0x10  e_sp:                          0xB8
    0x12       0x12  e_csum:                        0x0
    0x14       0x14  e_ip:                          0x0
    0x16       0x16  e_cs:                          0x0
    0x18       0x18  e_lfarlc:                      0x40
    0x1A       0x1A  e_ovno:                        0x0
    0x1C       0x1C  e_res:
    0x24       0x24  e_oemid:                       0x0
    0x26       0x26  e_oeminfo:                     0x0
    0x28       0x28  e_res2:
    0x3C       0x3C  e_lfanew:                      0xE0 ...

```

2. **NT_HEADERS:** The most important value in this header is `Signature`, which is set to `0x4550` (PE in ASCII)

```
print("[*] Signature: " + hex(pe.NT_HEADERS.Signature))
    [*] Signature: 0x4550
```

3. **FILE_HEADER:** This structure contains information about number of sections, whether the file is for 32- or 64-bit systems, and the time at which the file author has compiled it.

```
for field in pe.FILE_HEADER.dump():
    print(field)

    [IMAGE_FILE_HEADER]
    0xE4    0x0   Machine:                0x14C
    E6      0x2   NumberOfSections:       0x5
    0xE8    0x4   TimeDateStamp:       0x4F79D506 [Mon Apr 2 16:34:14 2012 UTC]
    0xEC    0x8   PointerToSymbolTable:   0x0
    0xF0    0xC   NumberOfSymbols:        0x0
    0xF4    0x10  SizeOfOptionalHeader:   0xE0
    0xF6    0x12  Characteristics:        0x10F
```

4. **OPTIONAL_HEADER:** Contrary to what its names suggests, it contains very important information including program's entry point in the PE file.
    a. **Magic**: `0x10B` for 32-bit and `0x20B` for 64-bit binaries.
    b. **AddressOfENtryPoint**: It specifies the relative virtual address of the entry point.
    c. **SectionAlignment**: Sections starts at multiples of this value.
    d. **SizeOfImage**: The amount of memory required to load this malware image
    e. **DATA_DIRECTORY**: An array of values.

```
for field in pe.OPTIONAL_HEADER.dump():
    print(field)

    [IMAGE_OPTIONAL_HEADER]
    0xF8       0x0   Magic:                         0x10B
    0xFA       0x2   MajorLinkerVersion:            0x6
    0xFB       0x3   MinorLinkerVersion:            0x0
    0xFC       0x4   SizeOfCode:                    0x32A00
    0x100      0x8   SizeOfInitializedData:         0x64200
    0x104      0xC   SizeOfUninitializedData:       0x0
    0x108      0x10  AddressOfEntryPoint:           0xCC00FFEE
```

```
    0x10C      0x14  BaseOfCode:                    0x1000
    0x110      0x18  BaseOfData:                    0x1000
    0x114      0x1C  ImageBase:                     0x400000
    0x118      0x20  SectionAlignment:              0x1000
    0x11C      0x24  FileAlignment:                 0x200
    0x120      0x28  MajorOperatingSystemVersion:   0x4
    0x122      0x2A  MinorOperatingSystemVersion:   0x0
    0x124      0x2C  MajorImageVersion:             0x0
    0x126      0x2E  MinorImageVersion:             0x0
    0x128      0x30  MajorSubsystemVersion:         0x4
    0x12A      0x32  MinorSubsystemVersion:         0x0
    0x12C      0x34  Reserved1:                     0x0
    0x130      0x38  SizeOfImage:                   0x9A000
    0x134      0x3C  SizeOfHeaders:                 0x1000
    0x138      0x40  CheckSum:                      0x0
    0x13C      0x44  Subsystem:                     0x2
    0x13E      0x46  DllCharacteristics:            0x0
    0x140      0x48  SizeOfStackReserve:            0x100000
    0x144      0x4C  SizeOfStackCommit:             0x1000
    0x148      0x50  SizeOfHeapReserve:             0x100000
    0x14C      0x54  SizeOfHeapCommit:              0x1000
    0x150      0x58  LoaderFlags:                   0x0
    0x154      0x5C  NumberOfRvaAndSizes:           0x10
 for val in
pe.OPTIONAL_HEADER.DATA_DIRECTORY:
    print("Name: " + val.name + " | Size : " + str(val.Size) + " | Address : "
+ hex(val.VirtualAddress))
```

5. **SECTIONS HEADER:** Contains information about the sections.

```


for section in pe.sections:
    print(section.Name.decode().rstrip('\x00'))
    print("   Virtual addresss: " + hex(section.VirtualAddress))
print("   Virtual size " + hex(section.Misc_VirtualSize))
    print("   Size of raw data " + hex(section.SizeOfRawData) + '\n')
    .text
       Virtual addresss: 0x1000
    Virtual size 0x32830
       Size of raw data 0x32a00

    .rdata
       Virtual addresss: 0x34000
       Virtual size 0x427a
       Size of raw data 0x4400
```

```
    .data
       Virtual addresss: 0x39000
       Virtual size 0x5cff8
       Size of raw data 0x2a00

    .idata
       Virtual addresss: 0x96000
       Virtual size 0xbb0
       Size of raw data 0xc00

    .reloc
       Virtual addresss: 0x97000
       Virtual size 0x211d
       Size of raw data 0x2200

# You can display the content of any of the sections by providing its index.
print(pe.sections[2])
    [IMAGE_SECTION_HEADER]
    0x228      0x0   Name:                          .data
    0x230      0x8   Misc:                          0x5CFF8
    0x230      0x8   Misc_PhysicalAddress:          0x5CFF8
    0x230      0x8   Misc_VirtualSize:             0x5CFF8
    0x234      0xC   VirtualAddress:                0x39000
    0x238      0x10  SizeOfRawData:                 0x2A00
    0x23C      0x14  PointerToRawData:              0x37200
    0x240      0x18  PointerToRelocations:          0x0
    0x244      0x1C  PointerToLinenumbers:          0x0
    0x248      0x20  NumberOfRelocations:           0x0
    0x24A      0x22  NumberOfLinenumbers:           0x0
    0x24C      0x24  Characteristics:               0xC0000040
```

STEP 5: List the DLLs that a binary will load and the function calls performed in each of those DLLs.

```
for entry in pe.DIRECTORY_ENTRY_IMPORT:
print(entry.dll.decode('utf-8'))     for
fnc in entry.imports:
        print("| ", fnc.name.decode('utf-8'))

    KERNEL32.DLL
       |  GetLocalTime
       |  ExitThread
       |  CloseHandle |
       WriteFile
       |  CreateFileA
       |  ExitProcess
```

```
        |  CreateProcessA |
        Sleep
        ...
    USER32.dll
        |  MessageBoxA
```

STEP 6: List the DLLs which another binary will load.

```
import pefile

pe = pefile.PE('./ch1/fakepdfmalware.exe')
for entry in pe.DIRECTORY_ENTRY_IMPORT:
print(entry.dll.decode('utf-8'))     for
fnc in entry.imports:
        print("| ", fnc.name.decode('utf-8'))
```

```
    KERNEL32.dll
    |  CreateDirectoryA
    |  ExpandEnvironmentStringsA
    |  WaitForSingleObject
    |  GetStartupInfoA
    |  SetCurrentDirectoryA
    |  WriteFile
    |  FreeResource
    |  GetTickCount
    …
    ADVAPI32.dll
    |  RegQueryValueExA
    |  RegCloseKey
    |  CryptEncrypt
    |  CryptAcquireContextA
    |  CryptCreateHash
    |  CryptHashData
    |  CryptDeriveKey
    |  CryptDestroyHash
    |  CryptDecrypt
    |  RegOpenKeyA
    SHELL32.dll
    |  ShellExecuteA
    LZ32.dll
    |  LZOpenFileA
    |  LZClose |
    LZCopy
    MSVCRT.dll
    |  strcmp
    |  free
    |  fclose
    |  fwrite |
    fread
    |  malloc
    …
```

STEP 7:

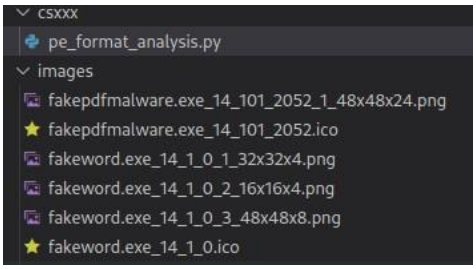
Malware always trick users by masquerading themselves as Word or PDF documents. In this task, we

will retrieve the executable images (icon of the file). We will use two commands from the `icoutils` library that contains the commands `wrestool` and `icotool`.

```
kali@kali [~/mds] sudo apt-get install icoutils

# Create a folder to store images kali@kali
[~/mds] mkdir images

# Extract the executable icon of several files (irbot.exe does not have icon)
kali@kali [~/mds] sudo wrestool -x ./ch1/*.exe --output=images
# Convert the .ico file to a .png image file kali@kali
[~/mds] icotool -x -o images images/*.ico
```



Part IV: Examining Malware Strings

STEP 7: Strings are printable characters within a program binary. It is important to analyze the strings of a suspicious software to extract important information such HTTP connections, FTP connections, IP addresses, servers' and hosts' names, embedded scripts or HTML requests, the programming language used to create the malware, etc.

```

kali@kali [~/mds] strings ircbot.exe > ircbot_strings.txt
# Search for lines that contain the word server OR http OR ftp kali@kali
[~/mds] strings ./ircbot.exe | grep -i 'server\|http\|ftp'
    [HTTPD]: Error: server failed, returned: <%d>.
```

```
    HTTP/1.0 200 OK Server:
    myBot
    HTTP/1.0 200 OK
    Server: myBot
    [HTTPD]: Failed to start worker thread, error: <%d>.
    [HTTPD]: Worker thread of server thread: %d.
    %s %s HTTP/1.1
    HttpSendRequestA
    HttpOpenRequestA
    server httpcon
    [HTTPD]: Failed to start server thread, error: <%d>. [HTTPD]:
    Server listening on IP: %s:%d, Directory: %s\.
    http httpserver
    irc.server2.net
```

Part V: Dynamic Malware Analysis

STEP 8: Discover www.virustotal.com [Upload a suspicious program and analyze the obtained report]

STEP 9: Discover https://hybrid-analysis.com/ [Upload a suspicious program, select the sandbox environment, and analyze the obtained report]