Cristian Carrizales
Lab 15
11/30/2025
CS 362-Digital Forensics

Part 1 - Non Techinical Summary

For the following lab I was to observe and perform static malware analysis using both python and readpe(linux command line tool) Using these tools allowed me to observe the malware from the “outside” which allows me to understand its structure and intent without actually running it. I learned what I must look for in the headers, functions, and imported .dll files to determine whether or not a file is malware or not. Basically, what is more than likely the norm vs what is abnormal. In my case the presence of .guclvne and the jumbled mess displayed in some of my outputs confirms that the malware sample downloaded is legitimate.

Part 2- Technical Summary

Part 1 Analyze the DOS Header

Before starting the lab there was a decent amount of pre preparation for it. Some of it included downloading some sample malware into my linux vm, installing tools like pefile/readpe, and setting up an environment inside a linux terminal. After setting up the first step of the lab is to work with the DOS Header which is the first 64 bytes of a Portable Executable(pe) file that allows it to be recognized as a valid ms-dos executable. When running the given string import pefile pe = pefile.PE('malware.exe') print(pe.DOS_HEADER) the output pictured below is shown. The most important part of the output I am looking for is the “magic number". The magic number is the first two bytes (0x5A4D or the ASCII characters "MZ"). This identifies the file as a valid executable for ms-dos.

```
┌──(env)─(cristian㉿kali)-[~/pe_tree]
└─$ python3
Python 3.13.3 (main, Apr 10 2025, 21:38:51) [GCC 14.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pefile
>>> pe = pefile.PE('malware.exe')
>>> print(pe.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:                      0x120
>>>
```

Next command line input has me displaying the header as a dictionary
The output below is what I received, however my output does not look like the example in the lab and is pretty difficult to read.

```
>>> print(pe.DOS_HEADER.dump_dict())
{'Structure': 'IMAGE_DOS_HEADER', 'e_magic': {'FileOffset': 0, 'Offset': 0,
Value': 23117}, 'e_cblp': {'FileOffset': 2, 'Offset': 2, 'Value': 144}, 'e_cp
': {'FileOffset': 4, 'Offset': 4, 'Value': 3}, 'e_crlc': {'FileOffset': 6, 'O
ffset': 6, 'Value': 0}, 'e_cparhdr': {'FileOffset': 8, 'Offset': 8, 'Value':
4}, 'e_minalloc': {'FileOffset': 10, 'Offset': 10, 'Value': 0}, 'e_maxalloc':
 {'FileOffset': 12, 'Offset': 12, 'Value': 65535}, 'e_ss': {'FileOffset': 14,
 'Offset': 14, 'Value': 0}, 'e_sp': {'FileOffset': 16, 'Offset': 16, 'Value':
 184}, 'e_csum': {'FileOffset': 18, 'Offset': 18, 'Value': 0}, 'e_ip': {'File
Offset': 20, 'Offset': 20, 'Value': 0}, 'e_cs': {'FileOffset': 22, 'Offset':
22, 'Value': 0}, 'e_lfarlc': {'FileOffset': 24, 'Offset': 24, 'Value': 64},
e_ovno': {'FileOffset': 26, 'Offset': 26, 'Value': 0}, 'e_res': {'FileOffset'
: 28, 'Offset': 28, 'Value': '\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00'}, 'e_
oemid': {'FileOffset': 36, 'Offset': 36, 'Value': 0}, 'e_oeminfo': {'FileOffs
et': 38, 'Offset': 38, 'Value': 0}, 'e_res2': {'FileOffset': 40, 'Offset': 40
, 'Value': '\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00
\\x00\\x00\\x00\\x00\\x00\\x00\\x00'}, 'e_lfanew': {'FileOffset': 60, 'Offset
': 60, 'Value': 288}}
```

Asked to display the size of the header which I mentioned before should be 64

```
>>> print(pe.DOS_HEADER.sizeof())
64
```

Displaying more specific info with more commands

```
>>> print('e_magic in decimal:', pe.DOS_HEADER.e_magic)
e_magic in decimal: 23117
>>> print('e_magic in hexadecimal:', hex(pe.DOS_HEADER.e_magic))
e_magic in hexadecimal: 0x5a4d
```

Next is displaying the offset to the pe header e_lfanew = value is a pointer. It tells Windows loader to skip x number of bytes from the beginning of the file to basically find where the NT Header begins.

```
>>> print('offset to the PE Header:', pe.DOS_HEADER.e_lfanew)
offset to the PE Header: 288
```

Part 2: Analyzing NT HEADERS

Speaking of NT Headers that is what I end up analyzing next, per lab instructions.

```
>>> print("[*] Signature: " + hex(pe.NT_HEADERS.Signature))
[*] Signature: 0x4550
>>> print(pe.FILE_HEADER)
[IMAGE_FILE_HEADER]
0x124      0x0   Machine:                      0x14C
0x126      0x2   NumberOfSections:             0x7
0x128      0x4   TimeDateStamp:                0x640A9521 [Fri Mar 10 02:25:
37 2023 UTC]
0x12C      0x8   PointerToSymbolTable:          0x0
0x130      0xC   NumberOfSymbols:              0x0
0x134      0x10  SizeOfOptionalHeader:         0xE0
0x136      0x12  Characteristics:              0x102
```

The photo above is a screen shot of the output after running the commands print("[*] Signature: " + hex(pe.NT_HEADERS.Signature)) and print(pe.FILE_HEADER). Some of the important information in the picture includes the signature which operates like a handshake telling the operating system exactly what file type it is dealing with. In this case signature 0x4550 translates to ASCII Characters (PE) or Portable Executable. Next is machine (0x14C) telling me what kind of cpu the malware was built for. 0x14C = Intel 386 (32-bit) this means the malware can run on 32-bit Windows and 64-bit Windows. Timestamp Fri Mar 10 02:25:37 2023 the exact second the attacker compiled the code and NumberOfSections=0x7 code has 7 sections which is larger than the usual 3-5 for standard programs.

The next step was to analyze the optional header inside the NT header running the string print(pe.NT_HEADERS.OPTIONAL_HEADER) gives the following output with a ton of information which includes some more important information such as AddressOfEntryPoint (0xF5603) which is the exact offset where the code starts running. SizeOfImage (0x17B000) What this means is that when the file is loaded into RAM it will take up exactly this amount of space. If the file on disk is much smaller it hints that the malware might unpack.

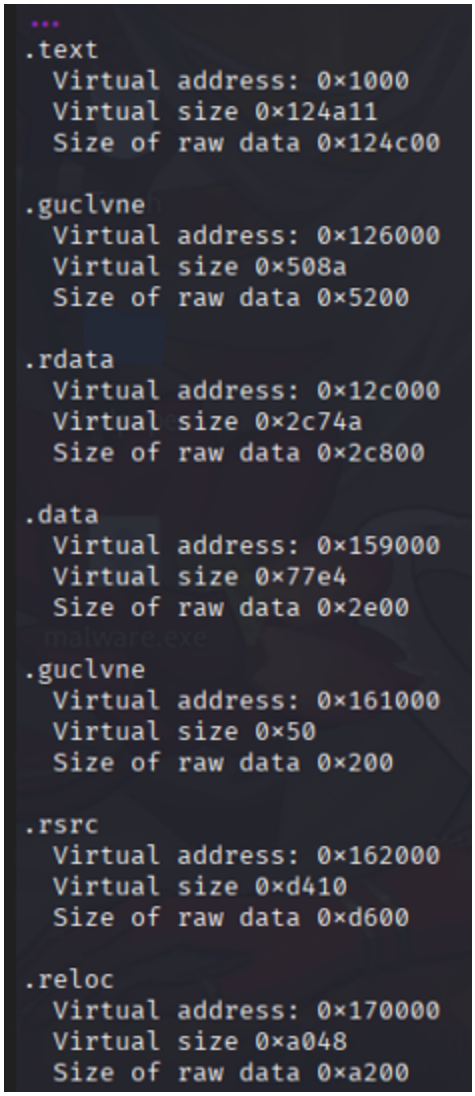
```
>>> print(pe.NT_HEADERS.OPTIONAL_HEADER)
[IMAGE_OPTIONAL_HEADER]
0x138       0x0   Magic:                       0x10B
0x13A       0x2   MajorLinkerVersion:          0xE
0x13B       0x3   MinorLinkerVersion:          0x23
0x13C       0x4   SizeOfCode:                  0x129E00
0x140       0x8   SizeOfInitializedData:       0x4BA00
0x144       0xC   SizeOfUninitializedData:     0x0
0x148       0x10  AddressOfEntryPoint:         0xF5603
0x14C       0x14  BaseOfCode:                  0x1000
0x150       0x18  BaseOfData:                  0x12C000
0x154       0x1C  ImageBase:                   0x400000
0x158       0x20  SectionAlignment:            0x1000
0x15C       0x24  FileAlignment:               0x200
0x160       0x28  MajorOperatingSystemVersion: 0x6
0x162       0x2A  MinorOperatingSystemVersion: 0x0
0x164       0x2C  MajorImageVersion:           0x0
0x166       0x2E  MinorImageVersion:           0x0
0x168       0x30  MajorSubsystemVersion:       0x6
0x16A       0x32  MinorSubsystemVersion:       0x0
0x16C       0x34  Reserved1:                   0x0
0x170       0x38  SizeOfImage:                 0x17B000
0x174       0x3C  SizeOfHeaders:               0x400
0x178       0x40  CheckSum:                    0x0
0x17C       0x44  Subsystem:                   0x2
0x17E       0x46  DllCharacteristics:          0x8140
0x180       0x48  SizeOfStackReserve:          0x100000
0x184       0x4C  SizeOfStackCommit:           0x1000
0x188       0x50  SizeOfHeapReserve:           0x100000
0x18C       0x54  SizeOfHeapCommit:            0x1000
0x190       0x58  LoaderFlags:                 0x0
0x194       0x5C  NumberOfRvaAndSizes:         0x10
>>>
```

Part 3 Analyzing Section Headers

The following part had me looking and analyzing section headers. When analyzing the picture and according to online sources sections like .text, .rdata and .rsrc are normal for sections the part that reads guclvne is not however. A random string of letters like that is a red flag. It can mean that the malware author used a custom tool to hide the virus so antivirus software wouldn't recognize it.

```
>>> for section in pe.sections:
...     print(section.Name.decode().rstrip('\x00'))
...     print("  Virtual address: " + hex(section.VirtualAddress))
...     print("  Virtual size " + hex(section.Misc_VirtualSize))
...     print("  Size of raw data " + hex(section.SizeOfRawData) + '\n'\
)
```

Output



Next I was to display information of the first section using print(pe.sections[0])

```
>>> print(pe.sections[0])
[IMAGE_SECTION_HEADER]
0×218      0×0   Name:                          .text
0×220      0×8   Misc:                          0×124A11
0×220      0×8   Misc_PhysicalAddress:          0×124A11
0×220      0×8   Misc_VirtualSize:              0×124A11
0×224      0×C   VirtualAddress:                0×1000
0×228      0×10  SizeOfRawData:                 0×124C00
0×22C      0×14  PointerToRawData:              0×400
0×230      0×18  PointerToRelocations:          0×0
0×234      0×1C  PointerToLinenumbers:          0×0
0×238      0×20  NumberOfRelocations:           0×0
0×23A      0×22  NumberOfLinenumbers:           0×0
0×23C      0×24  Characteristics:               0×60000020
```

Shown above(Output) we can see that the first section is .text. A takeaway from the output is that Misc_VirtualSize = 0x124A11 and SizeOfRawData = 0x124C00 are very close in size meaning that the .text section is not “packed”.

Then I was asked to display imported dll files and the used functions since my output did not match or contain similar output:

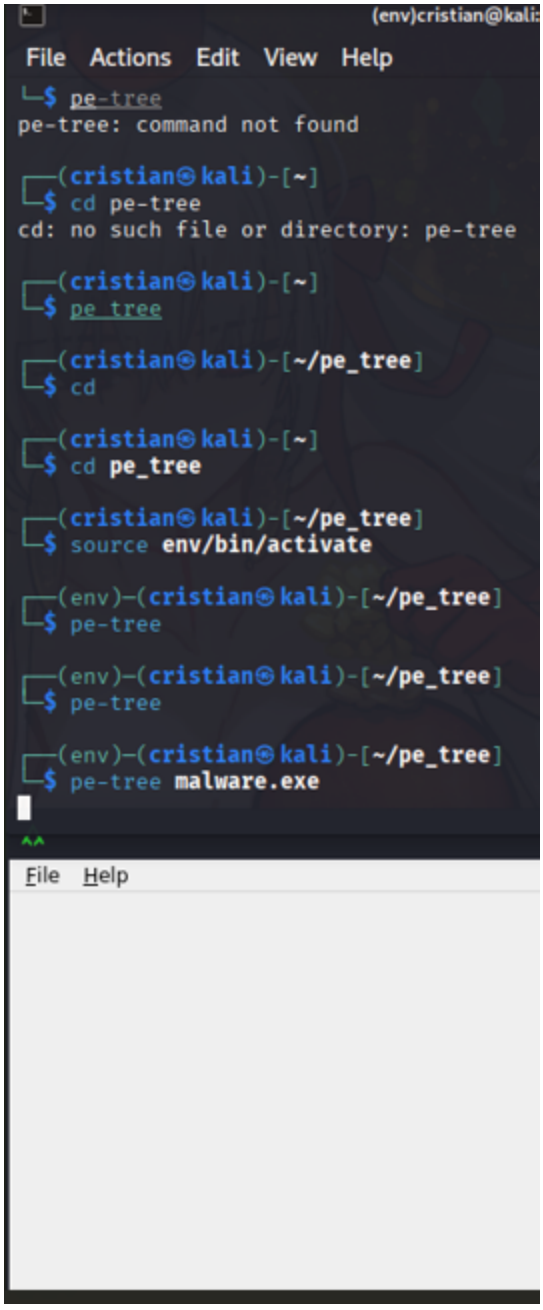
mscoree.dll
| _CorExeMain

I do not know if I should pursue this step further. Did the code run correctly? I assume everything should be okay? My Virtual Machine did crash shortly after the command was ran but I am unsure of it is coincidence or because of the block of codes output

```
>>> 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
|  GetComputerNameW
|  GetModuleFileNameA
|  GetCurrentProcessId
```

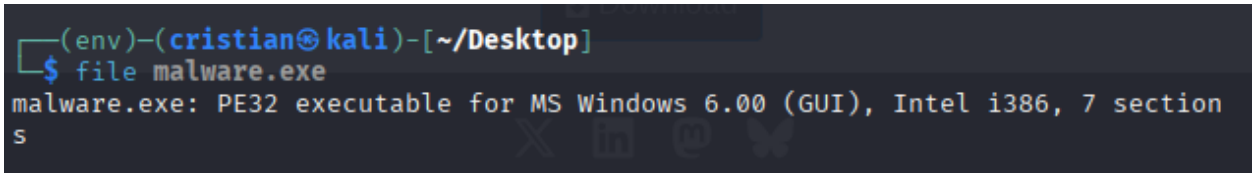
Part 4: Display an Executable Information using PE TREE

Unfortunately I could not get the PE-Tree tool working on my VM. I ensured the files were where they needed to be and ran some previously successful commands to see if anything had changed or if my environment was incorrect for whatever reason. None of that ended up being the case and despite my troubleshooting efforts my screen remained blank on the pe-tree GUI (Shown Below)



Part 6: Display an Executable Information using readpe tool

Fortunately the info displayed on pe-tree can be grabbed using readpe. First I run a command to verify/obtain more information about malware.exe



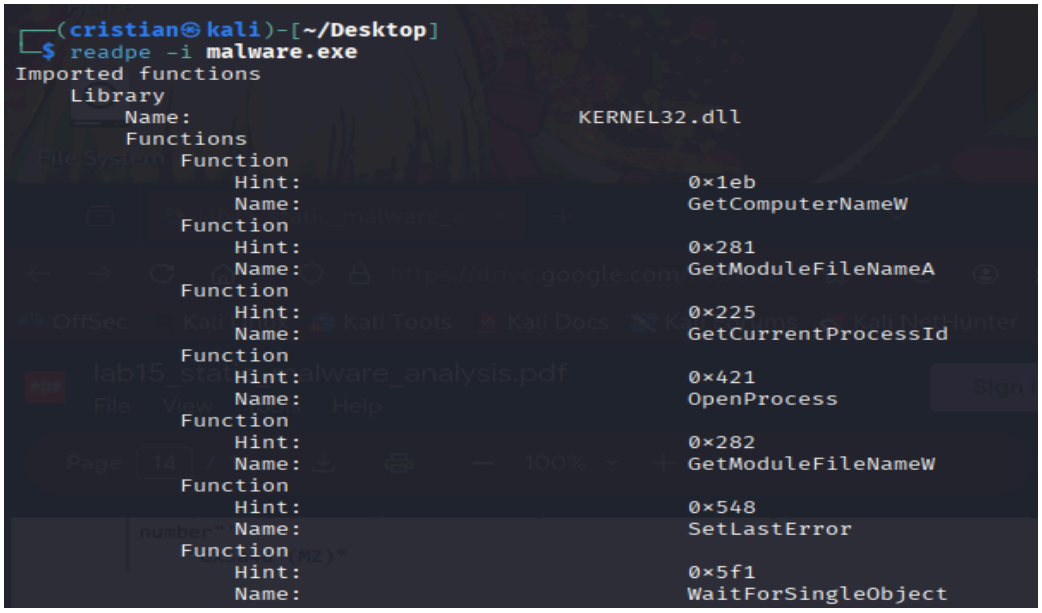
Then for this readpe portion of the lab I am essentially doing the same thing done in the python environment portion of the lab except now I am only using the linux command line to analyze the malware. readpe -h dos -f json malware.exe is showing me dos _ header again. The output displayed in the subsequent photo should match the output of an earlier block of python code for the same file.
Note* Could be thought of as the files “Birth Certificate”

```
┌──(cristian㉿kali)-[~/Desktop]
└─$ readpe -h dos -f json malware.exe
{
    "DOS Header": {
        "Magic number": "0x5a4d (MZ)",
        "Bytes in last page": "144",
        "Pages in file": "3",
        "Relocations": "0",
        "Size of header in paragraphs": "4",
        "Minimum extra paragraphs": "0",
        "Maximum extra paragraphs": "65535",
        "Initial (relative) SS value": "0",
        "Initial SP value": "0xb8",
        "Initial IP value": "0",
        "Initial (relative) CS value": "0",
        "Address of relocation table": "0x40",
        "Overlay number": "0",
        "OEM identifier": "0",
        "OEM information": "0",
        "PE header offset": "0x120"
    }
}
```

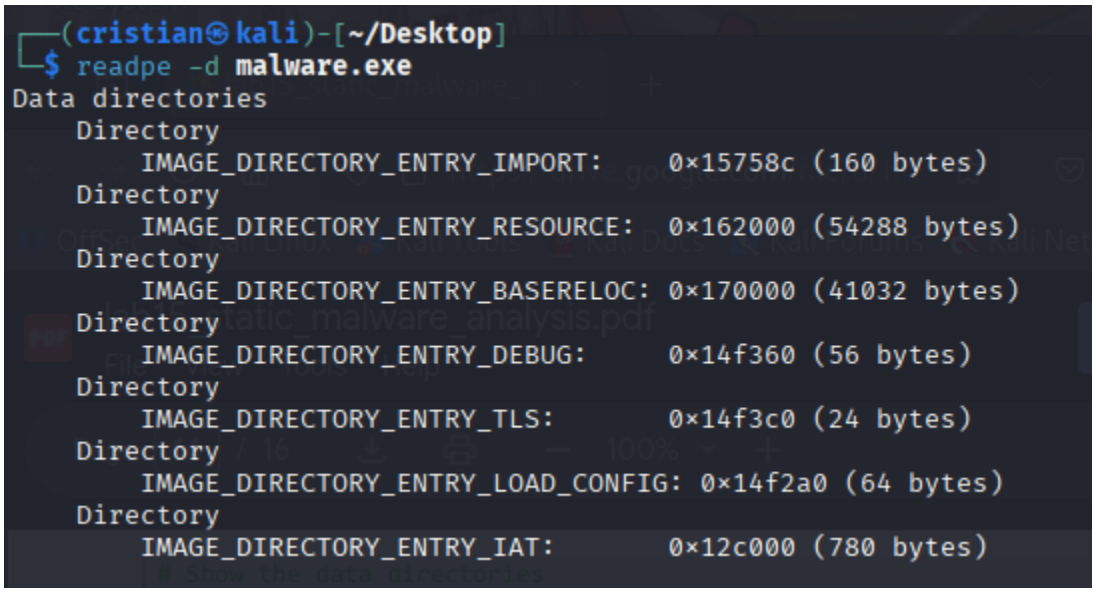
I was to then run a command to observe the magic number of the file using jq however after a couple of attempts I could not get an output that wasn't “null” as displayed here. Not entirely sure why it didn't work since I already know what the magic number is it shouldn't be an issue but it is odd to me that some of my outputs do not match what is shown in the lab

```
┌──(cristian㉿kali)-[~/Desktop]
└─$ readpe -h dos -f json malware.exe | jq '."DOS Header"."Magic
number"'
null
```

Following that I was to use readpe to once again display imported .dll files and functions the output matched the output from the earlier step that was used with python. Once again there is still no mscoree.dll that is seen on the lab example. I would assume now that my file is still packed and hiding its true nature.



Using readpe -d malware.exe to look at data directories output also does not match the lab output probably cause the file is still packed and hiding though im not exactly sure where or how to get it out of hiding.



Now I analyze sections of malware.exe again using readpe and the random string of letters .gucvlne once again highlights auspicious details present in the file.

```
┌──(cristian㉿kali)-[~/Desktop]
└─$ readpe -S malware.exe
Sections
    Section
        Name:                           .text
        Virtual Size:                   0×124a11 (1198609 bytes)
        Virtual Address:                0×1000
        Size Of Raw Data:               0×124c00 (1199104 bytes)
        Pointer To Raw Data:            0×400
        Number Of Relocations:          0
        Characteristics:                0×60000020
        Characteristic Names
                                            IMAGE_SCN_CNT_CODE
                                            IMAGE_SCN_MEM_EXECUTE
                                            IMAGE_SCN_MEM_READ
    Section
        Name:                           .guclvne
        Virtual Size:                   0×508a (20618 bytes)
        Virtual Address:                0×126000
        Size Of Raw Data:               0×5200 (20992 bytes)
        Pointer To Raw Data:            0×125000
        Number Of Relocations:          0
        Characteristics:                0×60000020
        Characteristic Names
                                            IMAGE_SCN_CNT_CODE
                                            IMAGE_SCN_MEM_EXECUTE
                                            IMAGE_SCN_MEM_READ
```

Part 7: Extract Strings from an Executable

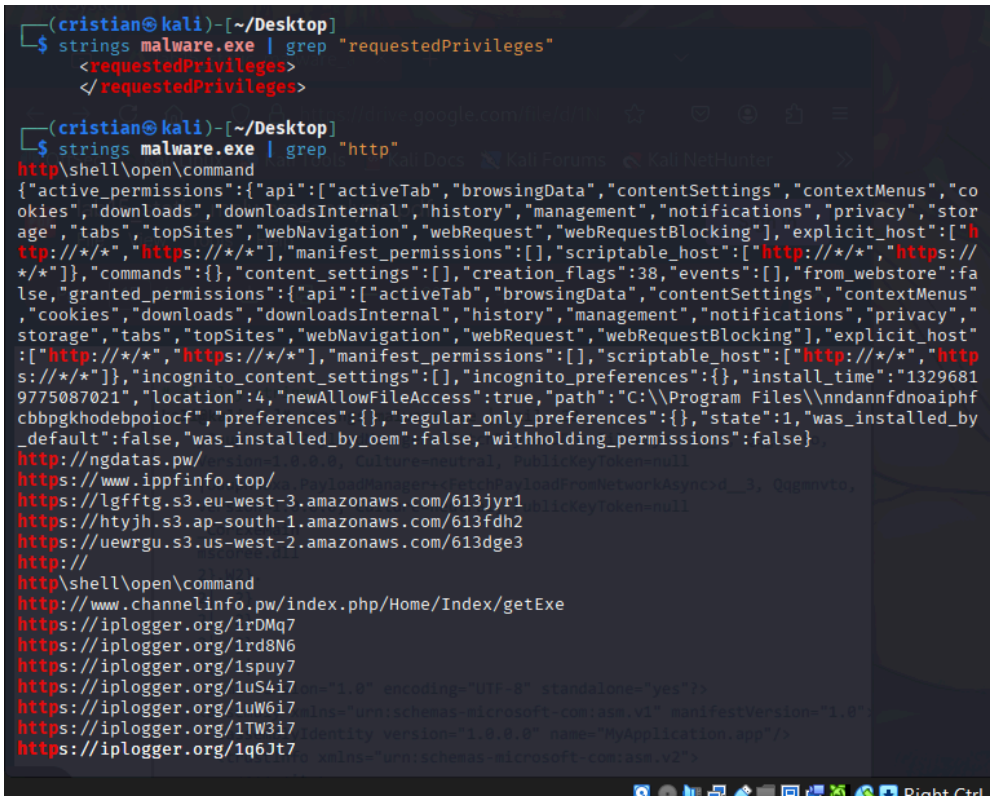
Final part of the lab has me using strings malware.exe | tail -20 to extract strings from the malware file. The output is something I have never seen during my time in this course

```
┌──(cristian㉿kali)-[~/Desktop]
└─$ strings malware.exe | tail -20
7 74787<7P7X7l7t7
80888L8T8h8p8
9,949H9P9d9l9
:(:0:D:H:L:`:d:h:|:
;$;,;8;@;H;d;
< <(<0<<<D<L<X<`<h<t◁<
= =$=(=,=0=4=H=L=P=T=X=\=`=d=h=l=p=t=x=╪═
> >,>0>8>◇D>H>P>T>\>`>h>l>t>x>
? ?(?4?@?D?L?P?X?\?d?h?p?t?|?
0$0(00040<0@0H0L0T0X0`0d0l0p0×0
1 1,181D1H1\1d1×1
2 2$2(2,2024282<2@2D2H2p2
3$3H3h3
444X4t4
5@5|5
6X6|6
7,7H7
909P9p9
:0:L:l:
;@;|;
```

The output appearing like this probably means that the file is still obfuscated as the output is

more than likely showing a camouflaged layer seeing as my output does not resemble anything close to the example output from the lab instructions.

Bonus Slide as an attempt to find out some information about URLs in the file



Part 3-Works Cited
PDF File Lab 15(2)