

Malware Detection via EAT

CSC 471 - 02

Tyler Prehl

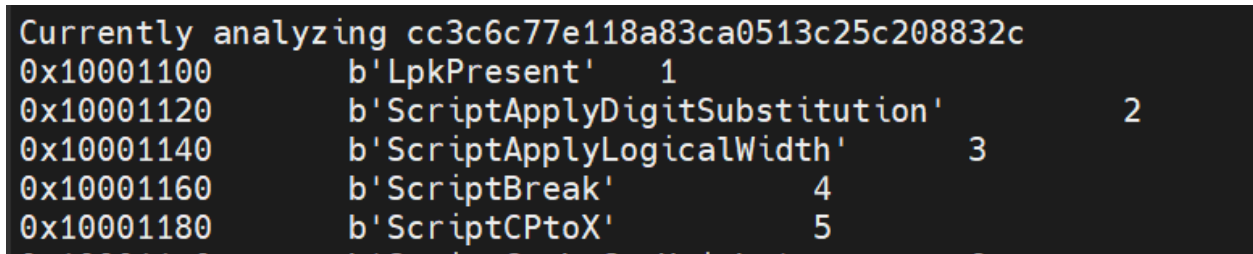
2/21/2022

Introduction

The purpose of this lab is to better understand how to use the IAT and/or EAT for static analysis of PE files. By using heuristic rules based on functions found in the IAT/EAT, we can identify what files may be malicious. In learning how to use the IAT/EAT for static analysis, I will also learn how to use the “pefile” library in Python to analyze PE files.

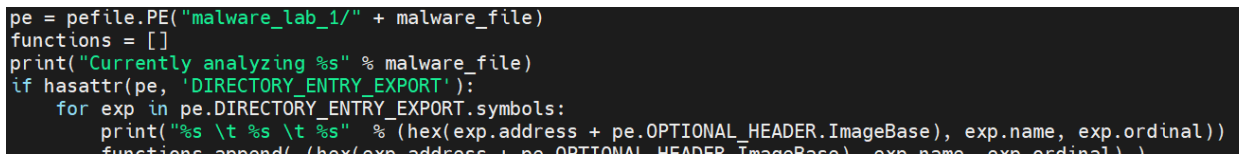
Analysis and Results

To detect whether a PE file is malicious, we can look for two different heuristic rules. Heuristic rule one occurs when three or more exported symbols have the same memory address. Heuristic rule two occurs when three or more exported symbols have the same *offset* from the previous exported symbol. To find out whether each of these rules apply to a PE file, we import the “pefile” library to break down each exported symbol to identify its starting address, name, and ordinal through the Export Address Table (EAT). An example output of this information looks like this:



```
Currently analyzing cc3c6c77e118a83ca0513c25c208832c
0x10001100      b'LpkPresent'      1
0x10001120      b'ScriptApplyDigitSubstitution'      2
0x10001140      b'ScriptApplyLogicalWidth'      3
0x10001160      b'ScriptBreak'      4
0x10001180      b'ScriptCPToX'      5
```

Here is the code used to produce this output:



```
pe = pefile.PE("malware_lab_1/" + malware_file)
functions = []
print("Currently analyzing %s" % malware_file)
if hasattr(pe, 'DIRECTORY_ENTRY_EXPORT'):
    for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
        print("%s \t %s \t %s" % (hex(exp.address + pe.OPTIONAL_HEADER.ImageBase), exp.name, exp.ordinal))
        functions.append((hex(exp.address + pe.OPTIONAL_HEADER.ImageBase), exp.name, exp.ordinal))
```

Now that we have access to the memory address information, we can analyze whether or not either of the malware rules apply. To do this, I logged each exported symbol to a list to be looped

through. To search for three or more of the same memory address, I used a dictionary to log occurrences of each memory address, and to search for three or more of the same memory offset, I once again used a dictionary to log the occurrences.

```
memory_offsets = {}
memory_addresses = {}
prev_address = 0
for function in functions:
    if function[0] not in memory_addresses:
        memory_addresses.update( {function[0]: 1} )
    else:
        repeats = memory_addresses.get(function[0]) + 1
        memory_addresses.update( {function[0]: repeats} )
    offset = hex( int(function[0], 16) - prev_address )
    if offset in memory_offsets:
        repeats = memory_offsets.get(offset) + 1
        memory_offsets.update( {offset: repeats} )
    else:
        memory_offsets.update( {offset: 1} )
    prev_address = int(function[0], 16)
```

With this complete, all that was left was to loop through each set, comparing the value of the number of occurrences (the location itself or the offset) to see if there were three or more. For each case, a proper print statement is issued to the user.

```
is_malware = False
for address in memory_addresses:
    if memory_addresses.get(address) > 2:
        print("Malware! More than 2 of the same memory address (%s) found!\n" % address)
        is_malware = True
        break
for offset in memory_offsets:
    if memory_offsets.get(offset) > 2:
        print("Malware! More than 2 of the same memory offset (%s) found!\n" % offset)
        is_malware = True
        break
if not is_malware:
    print("There was no malware found\n")
```

To make this code more usable, I added a section in the beginning that takes a whole folder of PE files and uses the rest of the code to loop through each file, identifying which have malware along the way. I imported the “os” library to do this.

```
1 import os
2 import pefile
3 import sys
4
5 files = os.listdir("/workdir/malware_lab_1")
6 print("PE files to analyze:")
7 print(files)
8 print("\n")
9
10 for malware_file in files:
11     pe = pefile.PE("malware_lab_1/" + malware_file)
12     functions = []
13     print("Currently analyzing %s" % malware_file)
14     if hasattr(pe, 'DIRECTORY_ENTRY_EXPORT'):
15         for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
```

An example output of a successful run of this program, enum_exports.py, looks like this:

```
root@5c40b4158d4c:/workdir # python3 enum_exports.py
PE files to analyze:
['6a764e4e6db461781d080034aab85aff', 'e0bed0b33e7b6183f654f0944b607618', '0fd6e3fb1cd5ec397ff3cdbaac39d80c', '1c1131112db91382b9d8b46115045097', '16d6b0e2c77da2776a88dd88c7cfc672', 'cc3c6c77e118a83ca0513c25c208832c']

Currently analyzing 6a764e4e6db461781d080034aab85aff
Malware! More than 2 of the same memory offset (0x10) found!

Currently analyzing e0bed0b33e7b6183f654f0944b607618
There was no malware found

Currently analyzing 0fd6e3fb1cd5ec397ff3cdbaac39d80c
Malware! More than 2 of the same memory offset (0xc) found!

Currently analyzing 1c1131112db91382b9d8b46115045097
There was no malware found

Currently analyzing 16d6b0e2c77da2776a88dd88c7cfc672
Malware! More than 2 of the same memory address (0x100011e0) found!

Currently analyzing cc3c6c77e118a83ca0513c25c208832c
Malware! More than 2 of the same memory offset (0x20) found!

root@5c40b4158d4c:/workdir #
```

As you can see, each PE file was properly analyzed for malware based on the heuristic rules described above.

Discussion and Conclusion

In conclusion, through this lab I successfully detected whether PE files were malicious or not using a Python script and the “pefile” library. By having a thorough understanding of the IAT and EAT - what information they hold and the service they provide - I could identify pieces of information that helped me determine whether the PE file was malicious or not. This will help me in the future as we continue to analyze PE files and dynamically linked libraries.