Static Malware Analysis - Binary Analysis & Symbolic Execution

Static analysis is a method of computer program debugging & quality assurance that involves examining the source code or binary code without the execution of the program.

This lab concerns with initial stages of static malware analysis. Its goal is to analyse a benign Windows PE file (Procmon.exe - tool for system monitoring & malware analysis) using pyhton programming language and it mimics the steps that a security analyst would take before moving towards the dynamic execution.

Hash Calculation (IOCs)

The below code computes cryptographic hashes directly from the raw bytes of Procmon.exe file to generate unique identifier for threat intelligence & correlation.

```
import hashlib
def compute_hash(path, algorithm):
  h = hashlib.new(algorithm)
  with open(path, "rb") as f:
    h.update(f.read())
  return h.hexdigest()
sample = r"Procmon.exe"
print("MD5: ", compute_hash(sample, "md5"))
print("SHA1: ", compute_hash(sample, "sha1"))
print("SHA256:", compute_hash(sample, "sha256"))
```

```
MD5:  c3e77b6959cc68baee9825c84dc41d9c
SHA1:  bc18a67ad4057dd36f896a4d411b8fc5b06e5b2f
SHA256: 3b7ea4318c3c1508701102cf966f650e04f28d29938f85d74ec0ec2528657b6e
```

The process involves loading file into binary mode, computing the multiple digests and observing how even modification of one byte can lead to change in all hashes. Thus, this replicates how SOCs (Security Operation Centers) track samples and check reputed databases. And SHA256 is the most reliable hash till today.

String Extraction

Here we are extracting printable ASCII strings using a regex match on byte sequences.

```
import re
def extract_strings(path):
  with open(path, "rb") as f:
    data = f.read()
  pattern = rb"[ -~]{4,}"
  return re.findall(pattern, data)
strings = extract_strings(sample)
for s in strings[:20]:
  print(s.decode(errors="ignore"))
```

```
!This program cannot be run in DOS mode.
V*0T
0RichU
.text
`.rdata
@.data
.rsrc
@.reloc
hpqQ
h`EN
h|nN
h\nN
hlnN
=UUU
h_rM
hDLN
h`GO
hDLN
h|GO
hDLN
```

The output contained expected benign artefacts such as DDL (Data Definition Language, a subset of SQL - used to define and manage the structure of databases) names, windows menu labels and standard system messages.

In this case there is no suspicious pattern due to the sample which is safe. But in real cases, this step would expose domains, embedded scripts, C2 URLs, registry keys, or obfuscation fragments. Usually, this becomes the first meaningful source of IOCs (Indicator of Compromise).

PE Header Inspection Using pefile

In this step, we are inspecting the PE (Portable Execution) metadata that includes entry points, image base, imported DLLs and imported APIs.

Procmon imported typical windows libraries including kernel32.dll, user32.dll, advapi32.dll. But no process injection or memory-manipulation APIs.

```
# Install pefile if not already installed
!pip install pefile

import pefile
pe = pefile.PE(sample)
print("Entry Point:", hex(pe.OPTIONAL_HEADER.AddressOfEntryPoint))
print("Image Base:", hex(pe.OPTIONAL_HEADER.ImageBase))
print("\nImported DLLs and functions:")
for entry in pe.DIRECTORY_ENTRY_IMPORT:
  print(" ", entry.dll.decode())
  for imp in entry.imports[:5]:
    print(" -", imp.name.decode() if imp.name else "None")
```

```
 - AcquireSRWLockShared
 - InitializeSRWLock
 - GetSystemInfo
 - VerSetConditionMask
  USER32.dll
 - LoadStringA
 - DrawEdge
 - GetMessageW
 - TranslateMessage
 - DispatchMessageW
  GDI32.dll
 - SaveDC
```

```
  - SafeArrayGetLBound
   SHLWAPI.dll
  - SHAutoComplete
   UxTheme.dll
  - IsThemeActive
  - SetWindowTheme
  - IsAppThemed
  - IsCompositionActive
   dwmapi.dll
  - DwmSetWindowAttribute
  - DwmDefWindowProc
   ntdll.dll
  - RtlGetVersion
```

Imports that are abnormal for example, CreatedRemoteThread, VirtualAllocEx, LoadLibraryA would indicate more aggressive capabilities. And minimal import table or unexpected entry points often suggests packing or obfuscation.

Thus, all of this is crucial for identifying malware families & and their behavioural intent before running them.

YARA Analysis

This below code is a simple YARA rule that detects the presence of the string 'http'. Its purpose is to demonstrate signature-based classification and show how SOC piplines flag files with embeded network indicators.

```
# Install yara if not already installed
!pip install yara-python

import yara

rule_source = """
rule ContainsHTTP {
strings:
$s = "http"
condition:
$s
}
"""
rules = yara.compile(source=rule_source)
matches = rules.match(sample)
print(matches)
```

```
Collecting yara-python
  Downloading yara_python-4.5.4-cp312-cp312-manylinux_2_17_x86_64.manylinux20
Downloading yara_python-4.5.4-cp312-cp312-manylinux_2_17_x86_64.manylinux2014
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.3/2.3 MB 47.4 MB/s eta 0:00:00
Installing collected packages: yara-python
Successfully installed yara-python-4.5.4
[ContainsHTTP]
```

As procmon contains HTPP-related content, so the rule matched successfully.

Complete Workflow Of Static Analysis

This final step is basically to combine all the scripts, here we are doing hash calculation, string extraction, import enumeration, IOC extraction (URLs, IPs) and YARA matching.

```
import hashlib, pefile, re, yara
# sample = "samples/procmon.exe"
def compute_hashes(path):
  algos = ["md5", "sha1", "sha256"]
  output = {}
  for a in algos:
    h = hashlib.new(a)
    with open(path, "rb") as f:
      h.update(f.read())
    output[a] = h.hexdigest()
  return output

def extract_strings(path):
  with open(path, "rb") as f:
    data = f.read()
  pattern = rb"[ -~]{4,}"
  return re.findall(pattern, data)

print("Hashes:", compute_hashes(sample))
print("\nStrings:")
print(extract_strings(sample)[:10])
print("\nImports:")
pe = pefile.PE(sample)
for entry in pe.DIRECTORY_ENTRY_IMPORT:
  print(entry.dll.decode())

print("\nIOCs:")
decoded = open(sample, "rb").read().decode(errors="ignore")
print("URLs:", re.findall(r"https?://[^\s\"']+", decoded))
print("IPs:", re.findall(r"\b\d{1,3}(?:\.\d{1,3}){3}\b", decoded))

print("\nYARA:")
rule = yara.compile(source="""
rule Simple {
strings: $s = "http"
condition: $s
}
""")
print(rule.match(sample))
```

```
Hashes: {'md5': 'c3e77b6959cc68baee9825c84dc41d9c', 'sha1': 'bc18a67ad4057dd3
```

```
Strings:
[b'!This program cannot be run in DOS mode.', b'V*0T', b'0RichU', b'.text', b

Imports:
WS2_32.dll
VERSION.dll
COMCTL32.dll
FLTLIB.DLL
KERNEL32.dll
USER32.dll
GDI32.dll
COMDLG32.dll
ADVAPI32.dll
SHELL32.dll
ole32.dll
OLEAUT32.dll
SHLWAPI.dll
UxTheme.dll
dwmapi.dll
ntdll.dll

IOCs:
URLs: ['https://go.microsoft.com/fwlink/?LinkId=521839', 'https://go.microsof
IPs: ['2.0.0.0', '6.0.0.0', '2.0.0.0', '6.0.0.0']

YARA:
[Simple]
```

The above step is the exact workflow which an analyst will use during initial investigation to decide whether a file requires a deeper reverse engineering or dynamic execution or not.

Reflection

This lab was focused on performing systematic static analysis on Windows executable without relying on running the program. It is an essential first-stage assessment in malware analysis.

Key Points:

- Hashes basically identify a file uniquely, and security tools check whether the file has been detected as a malware in threat intelligence databases or not.
- Strings, immediately indicate malicious intent of a file without deeper analysis. As strings inspection often reveal first useful clue like URLs, IPs, file path or commands.
- PE metadata describes the structure of the executable file. And it shows the imports and entry points of the program to indicate what the program is designed to do and what capabilities it might possess.

- YARA, rules automation detection, it basically allows the analysts and SOC systems to flag files that contains known malicious patterns, strings, or code fragments.