Networks and Systems Security
Week 06
Binary Analysis & Symbolic Execution

Aims of the Seminar

This laboratory exercise introduces students to the core concepts and practices of static malware analysis using Python. While traditional malware analysis often involves virtualised environments and controlled execution of suspicious samples, this lab focuses exclusively on safe, static techniques applied to benign Windows executables from trusted sources. The objective is to develop the foundational skills analysts rely on when triaging files, assessing potential risks, and extracting early indicators of compromise.

Students will work directly with Windows Portable Executable (PE) files, including a Sysinternals utility, which are safe to analyse. Through a series of structured tasks, students will learn how to extract cryptographic hashes, identify meaningful strings, read PE header information, evaluate imported functions, and apply basic YARA rules. These steps collectively mirror the initial stages of a real-world malware analysis workflow, allowing students to build confidence and familiarity before advancing to more complex dynamic or reverse-engineering scenarios.
The lab operates entirely through Python-based tooling, enabling students to understand not only how malware analysts interpret file structures, but also how automated pipelines are developed within security operations environments. By the end of the session, students will have a practical understanding of static triage methods and an appreciation of how these techniques contribute to wider incident response, threat intelligence, and detection engineering efforts.

Learning Outcomes
By the end of this lab, you will be able to:

1. Explain the role of static analysis in malware triage, including its purpose, strengths, and limitations within the broader malware analysis lifecycle.
2. Compute and interpret cryptographic hashes (MD5, SHA1, SHA256) as indicators of compromise, and explain how these identifiers are used in threat intelligence and incident response.
3. Extract and analyse human-readable strings from executable files, identifying potential artefacts such as file paths, registry references, and network indicators.

4. Inspect the structure of Windows PE files using Python, including reading header information, locating entry points, and examining imported libraries and API calls.
5. Detect simple behavioural characteristics using YARA rules, demonstrating how signature-based detection can be applied to classify files or identify suspicious patterns.
6. Conduct a complete static triage workflow, integrating hashing, string analysis, PE inspection, IOC extraction, and YARA matching into a coherent analytical process.
7. Interpret findings and articulate their significance, showing how extracted artefacts contribute to threat assessment, detection opportunities, and future investigation steps.

Feel free to discuss your work with peers, or with any member of the teaching staff.

Reminder

We encourage you to discuss the content of the workshop with the delivery team and any findings you gather from the session.

Workshops are not isolated, if you have questions from previous weeks, or lecture content, please come and talk to us.

Exercises herein represent an example of what to do; feel free to expand upon this.

Helpful Resources

Yara Library

https://github.com/VirusTotal/yara-python

Setting up

1. Instal the following libraries:

```
pip install pefile yara-python
```

2. Download Process Monitor (Procmon.exe) from the official Microsoft Sysinternals website

   https://learn.microsoft.com/sysinternals/downloads/procmon

Understanding the Role of a Malware Analyst

Before analysing any file, a professional malware analyst typically:

1. Reviews file metadata (hashes, timestamps, size).
2. Performs **static analysis** to understand structure and imports.
3. Searches for suspicious strings, URLs, IP addresses, or encoded data.
4. Applies custom or organisational YARA rules.
5. Produces **IOCs** (Indicators of Compromise) for incident response.
6. Writes reports for SOC teams or automated detection pipelines.

Hash Calculation (IOCs)

In malware analysis, cryptographic hashes are one of the most fundamental *Indicators of Compromise (IOCs)*.
They serve as unique identifiers for a file, allowing:

- Threat intelligence sharing
- Duplicate sample detection
- Quick reputation checks
- SOC correlation and automated triage

MD5, SHA1, and SHA256 all generate a fixed-size fingerprint of the file, but **SHA256 is the industry standard** for reliability and collision resistance.

What you are doing

You copy the full path of a benign executable (in this case, Procmon.exe from Microsoft Sysinternals), and feed it into a Python script. Python reads the file **as raw bytes**, computes the hash, and prints the results.

Python code and explanation

1. Copy the path of the Procmon file
2. Run the following script to print the hashes

```python
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"C:\Users\Basel\Downloads\ProcessMonitor\Procmon.exe"

print("MD5:    ", compute_hash(sample, "md5"))
print("SHA1:   ", compute_hash(sample, "sha1"))
print("SHA256:", compute_hash(sample, "sha256"))
```

Compare between the different hashes

3. Changing even a single byte in the file [you can use text editor] causes the hash to change completely (the avalanche effect).

String Extraction

Binary files, including malware, often contain human-readable text. Analysts review these strings to identify:

- Hardcoded paths
- Registry keys
- Network infrastructure (domains, URLs, IPs)
- Encryption keys or markers
- Persistence mechanisms

Strings provide valuable hints early in the analysis—often before deeper reverse engineering.

What the script does?

It scans the file for printable ASCII sequences of at least four characters. These correspond to readable strings embedded in the executable.

Python code and explanation

```
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"))
```

For benign tools like Procmon, strings look legitimate:

- DLL names
- Menu labels
- File paths
- Standard Windows messages

In actual malware, this step often reveals:

- Command-and-control domains
- Suspicious temp file names
- Embedded scripts
- Obfuscation artefacts

PE Header Inspection Using pefile

Why this matters?

Most Windows malware is delivered as a **Portable Executable (PE)** file. Learning to read PE headers reveals:

- How the program is structured
- Which libraries it relies on
- Whether the file shows signs of packing or obfuscation
- Possible capabilities (e.g., networking, registry manipulation)

What the script does?

The script loads the PE file and reads:

- **Entry Point**: where execution begins
- **Image Base**: preferred loading address in memory
- **Import Table**: all external functions (APIs) the binary relies on

These details form part of an analyst's early triage.

Python code and explanation

```
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")
```

Procmon imports many legitimate Windows APIs, such as:

- kernel32.dll
- user32.dll
- advapi32.dll

If this were malware, suspicious API imports might include:

- CreateRemoteThread (process injection)
- VirtualAllocEx (shellcode allocation)
- GetProcAddress and LoadLibraryA (dynamic API resolving)
- WinExec or ShellExecuteA (execution of child processes)

YARA Analysis

Why this matters?

YARA is the primary tool for:

- Writing detection rules
- Identifying malware families
- Matching file characteristics in SOC pipelines

Analysts use YARA to express signatures based on strings, binary patterns, and structural features.

What the script does

- Defines a rule that triggers whenever the string "http" is found.
- Compiles the rule using yara-python.
- Runs it against the sample file.

Python code and explanation

```
import yara

rule_source = """
rule ContainsHTTP {
    strings:
        $s = "http"
    condition:
        $s
}
"""

rules = yara.compile(source=rule_source)
matches = rules.match(sample)
print(matches)
```

Complete Static Triage Workflow

This section shows how different analysis techniques come together to form a **coherent triage workflow**, similar to what an analyst would do during a real investigation—except here all files are safe.

You should learn how to:

1. Compute hashes
2. Extract readable strings
3. Enumerate imports
4. Identify potential IOCs
5. Apply a YARA rule

Full integrated script with explanation

```
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()
    return re.findall(rb"[ -~]{4,}", 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))
```

The end ☺