Static Analysis

The technique of analyzing the suspect file without executing it

Common static analysis goals:

- Identifying the malware's target architecture
- Fingerprinting the malware
- Scanning the suspect binary with anti-virus engines
- Extracting strings, functions, and metadata associated with the file
- Identifying the obfuscation techniques used to thwart analysis
- Classifying and comparing the malware samples

1. Determining the file Type

Determining the file type will help determine the target OS ans architecture. File extensions are not the sole indicator of a file type, File signature can be used instead.

A File signature is a unique sequence of bytes that is written to the file's header. Different files have different signatures which can be used to identify the type of file. The Windows executables have a file signature of `MZ` or hexadecimal characters `4D 5A` in the first two bytes of the file.

1.1 Identifying the file using manual method

Open a file with a hex editor, and look for the file signature. (`xxd` on Linux and `HxD` on Windows)

1.2 Identifying File Type using Tools

Use file identification tools (`file` on Linux and `CFF Explorer` on Windows)

1.3 Determining the File using Python

The `python-magic` module can be used to determine the file type.

```python
import magic

m = magic.open(magic.MAGIC_NONE)
m.load()
print(m.file(r'log.exe'))
```

2. Fingerprinting the Malware

Fingerprinting involves generating the cryptographic hash values for the suspect binary based on it's file content. (Using `MD5, SHA1, SHA256`)

Useful because:

- Calculated based on file content, which remains the same (Unique identifier)
- During dynamic analysis, the file might copy and spread, hash helps to know if the analysis has to be performed on one file or on many more
- Used as an indicator to share with other security analysts
- Used to check if it's new (in *virusTotal*)

2.1 Generating Cryptographic hash using tools

On Linux: `md5sum, sha256sum sha1sum` On Windows: `HashMyFiles`

2.2 Determining cryptographic Hash with Python

Using the `hashlib` module

```
import hashlib

content = open(r'log.exe', 'rb').read()
print(hashlib.md5(content).hexdigest()
```

3. Multiple Anti-virus scanning

Helps determine whether malicious code signatures exist for the suspect file. Allows you to gather more information on this file.

3.1 Scanning the Suspect Binary with VirusTotal

- Allows you to scan the virus sample with several anti-virus techniques.
- Allows you to search their database by *hash*, *URL*, *domain*, or *IP address*.
- VirusTotal Graph allows you to visualize relationships between files that you submitted and it's associated indicators (*URL*, *domain*, or *IP address*)

3.2 Querying hash values using VirusTotal Public API

It allows you to automate file submissions, retrieve file/URL scan reports, and retrieve domain/IP reports.

The alternatives to scripting is to use PE analysis tools, such as: *pestudio* or *PPEE*

There are a few risks to consider when scanning a binary with Anti-virus scanners:

- If the malware scanner does not detect the file as malware, this does not mean it is safe.
- The suspect malware may contain sensitive information specific to your organization.
- Attackers can use the scan feature, to search for their malware.

4. Extracting Strings

Extracting strings might give us an indication about the program functionality and indicators associated with a suspect binary.

4.1 String extraction using tools

- **Linux:** `Strings`
  - extracts strings from a given file
  - Can extract both *ASCII* and *Unicode* (`-el` option) strings from binary
- **Windows:** `pestudio`
  - Displays both *ASCII* and *Unicode* strings.

4.2 Decoding Obfuscated Strings Using FLOSS (FireEye Labs Obfuscated String Solver)

Designed to identify and extract obfuscated strings from malware automatically. It also extracts *stack strings*

5. Determining File Obfuscation

Malware authors obfuscate or armour their malware binary. Obfuscation is used by malware authors to protect the inner-workings of their program from security researchers, malware analysts and reverse engineers. They commonly use *Packers* and *Cryptors* to obfuscate their file.

5.1 Packers and Cryptors

- *Packer:* program that uses compression to obfuscate the executable's content. This obfuscated content is then stored within the structure of a new executable file; The result is a new executable file with obfuscated contents on the disk. Upon execution of the packed program, it executes a decompression routine, which extracts the original binary in memory during runtime and triggers the execution.
- *Cryptor:* Similar to *Packer* but instead of using compression, it uses encryption to obfuscate the executable's content.

5.2 Detecting File Obfuscation Using Exeinfo PE

- Use *Exeinfo PE* (Windows) to detect if the sample is packed.
- It uses more than 4500 signatures (Stored in `userdb.txt`) to detect various compilers, packers and cryptors utilized to build a program.
- It directs you on how to unpack the sample

6. Inspecting PE Header Information

- Windows executables must conform to the *PE/COFF* (Portable Executable / Common Object File Format).
- When an executable is compiled, it includes a header (PE header), which describes it's structure.
- When the binary is executed, the OS loader reads the information from the PE header and then loads the binary content from the file into memory.
- The PE header contains information such as:
  - Where the executable needs to be loaded into memory
  - Address where the execution starts
  - List of libraries/functions on which the application relies on
  - Resources used by the binary

6.1 Inspecting File Dependencies and Imports

Malware needs access to Windows API functions (Through DLL's) to interact with files, registry, network, and so on. The functions that are imported from other files, are called *Imported functions* or *Imports*.

The file dependencies in Windows executables are stored in the import table of the PE file structure.

Malware can also load DLL explicitly during runtime using API calls such as: `LoadLibrary()` or `LdrLoadDLL()` and it can resolve the function address using the `GetProcessAddress()` API. Information about the DLL's loaded during runtime will not be present in the PE file. PE imports can also help detect if a malware has been packed, as if true, it will display much less imports.

6.2 Inspecting Exports

Inspecting the DLL exports in `pestudio` can give you a general idea of the DLL's capabilities.

6.3 Examining PE Section Table And Sections

The content of the PE file is divided into sections. The sections are immediately followed by the PE header. These sections represent either *code* or *data* and they have in-memory attributes such as read/write. The sections representing code, contains instructions that will be executed by the processor. The sections containing data can represent different types of data: Read/write program variables, import/export tables, resources, and so on.

Section Name	Description
`.text` or `CODE`	Contains executable code
`.data` or `DATA`	Contains read/write data and global variables

Section Name	Description
`.rdata`	Contains read-only data. Sometimes it also contains import and export information
`.idata`	If present, contains the import table. If not present, the import information is stored in `.rdata` section
`.edata`	If present, contains export information. If not present, the export information is found in `rdata` section
`.rsrc`	This section contains the resources used by the executable such as icons, dialogs, menus, strings, and so on

These section names are for humans, so an attacker might change them to obfuscate them.

Packed samples might display certain anomalies, such as:

- Section names no not contain common sections added by the compiler (`.text, .data`), but contain section names like: `UPX0, UPX1`.
- Entry point is the `UPX1` section (Indicating that execution might start in this section (Decompression routine))
- Typically, `raw-size` and `virtual-size` should be almost equal.

6.4 Examining the Compilation Timestamp

The PE header contains information on when this binary was compiled.

6.5 Examining PE Resources

Resources required by the binary are stored under `.rsrc`. Often, attackers store information (Such as additional binary, decoy documents or configuration data) in this section. *ResourceHacker* is a great tool to examine, view and extract the resources from a suspect's binary.

7 Comparing And Classifying the Malware

Comparing the analysed sample with samples stored in a public or private repository can give an understanding of the malware family, it's characteristics, and the similarity with the previous analyzed samples.

7.1 Classifying Malware Using Fuzzy Hashing

ssdeep is a useful tool to generate the fuzzy hash for a sample, and it also helps in determining percentage similarity between the samples.

You can also compare a given malware sample with a dictionary, to obtain familiarity

```
ssdeep * > all_hashes.txt

ssdeep -m all_hashes.txt blab.exe
```

This will compute the fuzzy hashes of every sample and then cross-compare with `blab.exe`.

7.2 Classifying Malware Using Import Hash

Is a technique used to identify related samples by calculating hash values based on the library/imported function (API) names and their particular order within the executable. If the files where compiled from the same source and in the same manner, those files would tend to have the same *imphash* value.

7.3 Classifying Malware Using Section Hash

Similar to imphash, section hashing calculates the MD5 of each section (`.text, ,data, .rdata, ...`)

7.4 Classifying Malware Using YARA

A form of string-based malware classification.

YARA is a powerful malware identification and classification tool. Malware researchers can create YARA rules based in textual or binary information contained within the malware specimen. These YARA rules consist of a set of strings and a boolean expression, which determines it's logic. Once the rule is written, you can use those rules to scan files using the YARA utility or `yara-python` to integrate with your tools.

7.4.1 Installing YARA

7.4.2 YARA rule basics

A YARA rule consists of the following components:

- *Rule identifier:* Name that describes the rule
- *String Definition:* Section where the strings (text, hex, or regex) that will be part of the rule are defined
- *Condition Section:* Not an optional section, this is where the logic of the rule resides. Must contain a Boolean expression

```
rule suspicious_strings
{
strings:
  $a = "Synflooding"
  $b = "Portscanner"
  $c = "Keylogger" ascii wide nocase

condition:
  ($a or $b or $c)
}
```

7.4.3 Running YARA

Using the previous rule, we can analyse the strings of our samples. The rule will turn out true if any of these strings is found in a file.

```
    yara -r suspicious.yara samples/
```

The previous example will search any file, this one will match only executables. The `$mz at 0` string will tell YARA to look for the PE file signature. The `ascii wide nocase` are parameters, which tell YARA to look for the preceding string in ASCII, Unicode and case insensitive.

```
rule suspicious_strings
{
strings:
  $mz = {4D 5A}
  $a = "Synflooding" ascii wide nocase
  $b = "Portscanner" ascii wide nocase
  $c = "Keylogger" ascii wide nocase

condition:
  ($mz at 0) and ($a or $b or $c)
}
```

7.4.4 Applications of YARA

One example of a real use-case scenario of a YARA rule:

Detect a executable file, that contains an embedded Microsoft Office document in it. The rule will trigger if the hex string is found at an offset greater than `1024` bytes in the file (This skips the PE header), and the `filesize` specifies the end of the file.

```
rule embedded_office_document
{
meta:
  description = "Detects embedded office document"
strings:
  $mz = {4D 5A}
  $a = {D0 CF 11 E0 A1 B1 1A E1}

condition:
  ($mz at 0) and $a in (1024..filesize)
}
```