

# Basic Static Analysis

## Introduction

In the previous rooms of this module, we learned about the basics of computer architecture and Assembly language. While those topics are essential building blocks to learning malware analysis, we will start analyzing malware starting from this room.

## Pre-requisites

Before starting this room, it is recommended that you complete the following rooms.

- [Intro to Malware Analysis](#)
- [x86 Assembly Crash Course](#)
- [x86 Architecture Overview](#)

## Learning Objectives

The first step in analyzing malware is generally to look at its properties without running it. This type of analysis is called static analysis because the malware is static and is not running. We will cover basic static analysis in this room. In particular, we will cover the following topics.

- Lab setup for malware analysis
- Searching for strings in a malware
- Fingerprinting malware through hashes
- Signature-based detection mechanisms
- Extracting useful information from the PE header

So without further ado, let's move on to the next task to learn about setting up a malware analysis lab.

## Lab Setup

### Basic precautions for malware analysis:

Before analyzing malware, one must understand that malware is often destructive. This means that when malware is being analyzed, there is a high chance of damaging the environment in which it is being analyzed. This damage can be permanent, and it might take more effort to get rid of this damage than the effort made to analyze the malware. Therefore, creating a lab setup that can withstand the destructive nature of malware is necessary.

### Virtual Machines:

A lab setup for malware analysis requires the ability to save the state of a machine (snapshot) and revert to that state whenever required. The machine is thus prepared with all the required tools installed, and its state is saved. After analyzing the malware in that machine, it is restored to its clean state with all the tools installed. This activity ensures that each malware is analyzed in an otherwise clean environment, and after analysis, the machine can be reverted without any sustained damage.

Virtual Machines provide an ideal medium for malware analysis. Some famous software used for creating and using Virtual Machines includes [Oracle VirtualBox](#) and [VMWare Workstation](#). These applications can create snapshots and revert to them whenever required, making them well-suited for our malware analysis pursuit. In short, the following steps portray the usage of Virtual Machines for malware analysis.

1. Created a fresh Virtual Machine with a new OS install
2. Set up the machine by installing all the required analysis tools in it
3. Take a snapshot of the machine
4. Copy/Download malware samples inside the VM and analyze it
5. Revert the machine to the snapshot after the analysis completes

Following these steps ensures that your VM is not contaminated with remnants of previous malware samples when analyzing new malware. It also ensures that you don't have to install your tools again and again for each analysis. Selecting the tools to install in your malware analysis VM can also be hectic. One can use one of the freely available malware analysis VMs with pre-installed tools to ease this task. Let's review some common malware analysis VMs most popular among security researchers.

### **FLARE VM:**

The FLARE VM is a Windows-based VM well-suited for malware analysis created by Mandiant (Previously FireEye). It contains some of the community's favorite malware analysis tools. Furthermore, it is also customizable, i.e., you can install any of your own tools to the VM. FLARE VM is compatible with Windows 7 and Windows 10. For a list of tools already installed in the VM and installation steps, you can visit the [GitHub page](#) or the [Mandiant blog](#) for the VM. Since it is a Windows-based VM, it can perform dynamic analysis of Windows-based malware.

An instance of FLARE VM is attached to this room for performing practical tasks. Please click the Start Machine button on the top-right corner of this task to start the machine before proceeding to the next task. The attached VM has a directory named mal on the Desktop, which contains malware samples that we would be analyzing for this room.

### **REMnux:**

REMnux stands for Reverse Engineering Malware Linux. It is a Linux-based malware analysis distribution created by Lenny Zeltser in 2010. Later on, more people joined the team to improve upon the distribution. Like the FLARE VM, it includes some of the most popular reverse engineering and malware analysis tools pre-installed. It helps the analysts save time that would otherwise be spent in identifying, searching for, and installing the required tools. Details like installation and documentation can be found on [GitHub](#) or the [website](#) for distribution. Being a Linux-based distribution, it cannot be used to perform dynamic analysis of Windows-based malware. REMnux was previously used in the Intro to Malware Analysis room and will also be used in the upcoming rooms.

Please click the Start Machine button on the top-right corner of this task to start the machine before proceeding to the next task. The machine will start in a split-screen view. If the VM is not visible, use the blue Show Split View button at the top-right of the page. The attached VM has a directory named mal on the Desktop, which contains malware samples that we would be analyzing for this room. Alternatively, you might use the following information to log into the machine:

- Machine IP: MACHINE\_IP
- Username: Administrator
- Password: letmein123!

## String Search

In the [Intro to Malware Analysis room](#), we identified that searching for strings is one of the first steps in malware analysis. A string search provides useful information to a malware analyst by identifying important pieces of strings present in a suspected malware sample. To learn a little more about strings, we can look at [this room](#) dedicated to strings.

### How a string search works:

A string search looks at the binary data in a malware sample regardless of its file type and identifies sequences of ASCII or Unicode characters followed by a null character. Wherever it finds such a sequence, it reports that as a string. This might raise the question that not all sequences of binary data that looks like ASCII or Unicode characters will be actual strings, which is right. Many sequences of bytes can fulfill the criteria mentioned above but are not strings of useful value; rather, they might include memory addresses, assembly instructions, etc. Therefore, a string search leads to many False Positives (FPs). These FPs show up as garbage in the output of our string search and should be ignored. It is up to the analyst to identify the useful strings and ignore the rest.

## What to look for?

Since an analyst has to identify actual strings of interest and differentiate them from the garbage, it is good to know what to look for when performing a string search. Although a lot of useful information can be unearthed in a string search, the following artifacts can be used as Indicators of Compromise (IOCs) and prove more useful.

- Windows Functions and APIs, like SetWindowsHook, CreateProcess, InternetOpen, etc. They provide information about the possible functionality of the malware
- IP Addresses, URLs, or Domains can provide information about possible C2 communication. The Wannacry malware's killswitch domain was found using a string search
- Miscellaneous strings such as Bitcoin addresses, text used for Message Boxes, etc. This information helps set the context for further malware analysis

## Basic String Search:

In the Intro to Malware Analysis room, we learned about the strings utility, which is pre-installed in Linux machines and can be used for a basic string search. Similarly, the FLARE VM comes with a Windows utility, strings.exe, that performs the same task. This Windows strings utility is part of the Sysinternals suite, a set of tools published by Microsoft to analyze different aspects of a Windows machine. Details about the strings utility can be found in Microsoft Documentation. The strings utility comes pre-installed in the FLARE VM attached to this room. The good thing about the command line strings utility is that it can dump strings to a file for further analysis. In the attached VM, executing the following command will perform a basic string search in a binary.

```
C:\Users\Administrator\Desktop>strings <path to binary>
```

Several other tools included in the FLARE VM can be used for string search. For example, CyberChef (Desktop>FLARE>Utilities>Cyberchef) has a recipe for basic string search as well. PEstudio (Desktop>FLARE>Utilities>pestudio) also provides a string search utility. PEstudio also provides some additional information about the strings, like, the encoding, size of the string, offset in the binary where the string was found, and a hint to guess what the string is related to. It also has a column for a blacklist, which matches the strings against some signatures.

pestudio 9.22 - Malware Initial Assessment - www.winitor.com

file settings about

encoding (2)	size (bytes)	file-offset	blacklist (25)	hint (9)	value (8556)
ascii	25	0x00901A2	x	-	QueryPerformanceFrequency
ascii	14	0x009025A	x	-	VirtualProtect
ascii	19	0x00902A2	x	-	GetCurrentProcessId
ascii	15	0x00902F8	x	-	DeviceIoControl
ascii	16	0x00903A8	x	-	TerminateProcess
ascii	16	0x00903BC	x	-	GetCurrentThreadId
ascii	11	0x00903D0	x	-	OpenProcess
ascii	10	0x0090410	x	-	DeleteFile
ascii	13	0x009042A	x	-	FindFirstFile
ascii	12	0x009043C	x	-	FindNextFile
ascii	9	0x009047E	x	-	WriteFile
ascii	13	0x00904EC	x	-	CreateProcess
ascii	24	0x00905D2	x	-	CreateToolhelp32Snapshot
ascii	14	0x00905EE	x	-	Process32First
ascii	13	0x0090600	x	-	Process32Next
ascii	16	0x009064C	x	-	SetPriorityClass
ascii	25	0x0090670	x	-	QueryFullProcessImageName
ascii	19	0x00906BA	x	-	GetForegroundWindow
ascii	13	0x00907C0	x	-	OpenClipboard
ascii	14	0x00907D0	x	-	CloseClipboard
ascii	16	0x00907E2	x	-	SetClipboardData
ascii	14	0x00907F6	x	-	EmptyClipboard
ascii	9	0x0090808	x	-	SendInput
ascii	13	0x0090814	x	-	MapVirtualKey
ascii	16	0x009086A	x	-	GetClipboardData
ascii	5	0x0038399	-	format-string	D8%g
ascii	4	0x00586E3	-	file	ds.h
ascii	9	0x009015E	-	file	ntdll.dll
ascii	12	0x00906AA	-	file	KERNEL32.dll

sha256: 2606E785097F8C4D22E7C28CCAEE1FEDD98DC91B307EA56AD9C69B5EED7BC363    cpu: 64-bit    file-type: executable    subsystem: GUI    entry-point: 0x0005A48C    signature: n/a

The above screenshot from PEstudio shows strings found by PEstudio in a malware sample. This can be done by selecting strings in the left pane after loading the PE file in PEstudio. The blacklist here shows a bunch of Windows API calls, which PEstudio flags as potentially used in malicious processes. You can learn about these APIs using resources like [MalAPI](#) or [MSDN](#).

## Obfuscated strings:

Searching for strings often proves one of the most effective first steps in malware analysis. As seen in [the case of Wannacry](#), effective use of string search can often disrupt malware propagation and infection. The malware authors know this and don't want a simple string search to thwart their malicious activities. Therefore, they deploy techniques to obfuscate strings in their malware. Malware authors use several techniques to obfuscate the key parts of their code. These techniques often render a string search ineffective, i.e., we won't find much information when we search for strings.

Mandiant (then FireEye) launched FLOSS to solve this problem, short for FireEye Labs Obfuscated String Solver. FLOSS uses several techniques to deobfuscate and extract strings that would not be otherwise found using a string search. The type of strings that FLOSS can extract and how it works can be found in [Mandiant's blog post](#).

To execute FLOSS, open a command prompt and navigate to the Desktop directory. From there, use the following command.

```
C:\Users\Administrator\Desktop>floss -h
```

This command will open the help page for FLOSS. We can use the following command to use FLOSS to search for obfuscated strings in a binary.

```
C:\Users\Administrator\Desktop>floss --no-static-strings <path to binary>
```

Please remember that the command might take some time to execute, and you might see what appear to be some error messages before the results are generated.

\*\*\*\*\*

**Answer the questions below:**

**On the Desktop in the attached VM, there is a directory named 'mal' with malware samples 1 to 6. Use floss to identify obfuscated strings found in the samples named 2, 5, and 6. Which of these samples contains the string 'DbgView.exe'?**

Sample 2:

```
C:\Users\Administrator\Desktop>floss --no-static-strings C:\Users\Administrator\Desktop\mal\2
FLOSS decoded 0 strings
FLOSS extracted 0 stackstrings
Finished execution after 0.016000 seconds
FLARE Thu 05/22/2025 21:37:02.39
```

Sample 5:

```
C:\Users\Administrator\Desktop>floss --no-static-strings C:\Users\Administrator\Desktop\mal\5
FLOSS decoded 0 strings
FLOSS extracted 0 stackstrings
Finished execution after 0.016000 seconds
FLARE Thu 05/22/2025 21:37:35.18
```

Sample 6:

```

C:\Users\Administrator\Desktop>floss --no-static-strings C:\Users\Administrator\Desktop\mal\6
WARNING:envi.codeflow:parseOpcode error at 0x1400596c8 (addCodeFlow(0x140058558)): InvalidInstruction("'c5eeffff440fb70233c041baffff0000' at 0x1400596c8L",)
WARNING:envi.codeflow:parseOpcode error at 0x14004edc2 (addCodeFlow(0x14004b0d4)): InvalidInstruction("'db75178b1509f30400488d0d2ca50300' at 0x14004edc2L",)
WARNING:envi.codeflow:parseOpcode error at 0x14004e645 (addCodeFlow(0x14004e5b0)): InvalidInstruction("'db75178b1586fa0400488d0df9ab0300' at 0x14004e645L",)
FLOSS decoded 1 strings
@@AD
FLOSS extracted 45 stackstrings
Dbgview.exe
Dbgview.exe
AV0ZAWI
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
D$ H
Dbgview.exe
Dbgview.exe
L$ H
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
A AC
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe

```

```

Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
112222I
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
D$ H
@@AD
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Dbgview.exe
Finished execution after 36.482000 seconds
FLARE Thu 05/22/2025 21:39:06.14

```

Answer: **6**

## Fingerprinting Malware

When analyzing malware, it is often required to identify unique malware and differentiate them from each other. File names can't be used for this purpose as they can be duplicated easily and might be confusing. Also, a file name can be changed easily as well. Hence, a hash function is used to identify a malware sample uniquely. A hash function takes a file/data of arbitrary length as input and creates a fixed-length unique output based on file contents. This process is irreversible, as you can't recreate

the file's contents using the hash. Hash functions have a very low probability (practically zero) of two files having different content but the same hash. A hash remains the same as long as the file's content remains the same. However, even a slight change in content will result in a different hash. It might be noted that the file name is not a part of the content; therefore, changing the file name does not affect the hash of a file.

Besides identifying files, hashes are also used to store passwords to authenticate users. In malware analysis, hash files can be used to identify unique malware, search for this malware in different malware repositories and databases, and as an Indicator of Compromise (IOC).

### **Commonly used methods of calculating File hashes:**

For identification of files, a hash of the complete file is taken. There are various methods to take the hash. The most commonly used methods are:

- Md5sum
- Sha1sum
- Sha256sum

The first two types of hashes are now considered insecure or prone to collision attacks (when two or more inputs result in the same hash). Although a collision attack for these hash functions is not very probable, it is still possible. Therefore, sha256sum is currently considered the most secure method of calculating a file hash. In the attached VM, we can see that multiple utilities calculate file hashes for us.

### **Finding Similar files using hashes:**

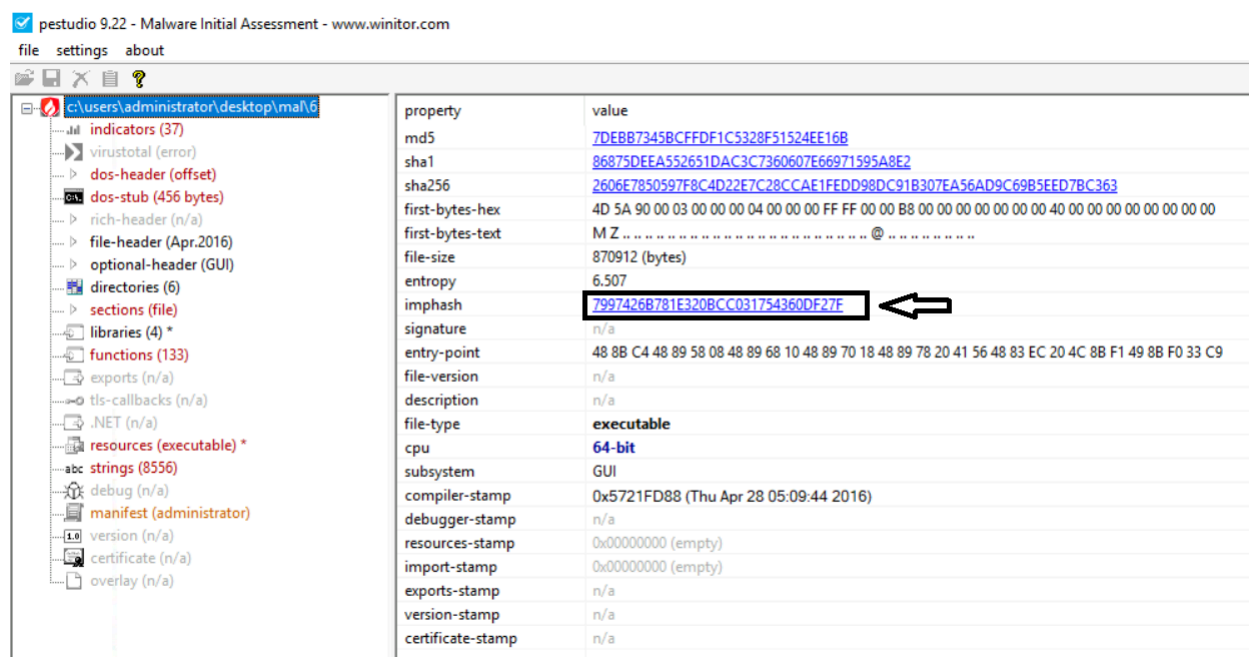
Another scenario in which hash functions help a malware analyst is identifying similar files using hashes. We already established that even a slight change in the contents of a file would result in a different hash. However, some types of hashes can help identify the similarity among different files. Let's learn about some of these.

### **Imphash:**

The imphash stands for "import hash". Imports are functions that an executable file imports from other files or Dynamically Linked Libraries (DLLs). The imphash is a hash of the function calls/libraries that a malware sample imports and the order in which these libraries are present in the sample. This helps identify samples from the same threat groups or perform similar activities. More details on the Imphash can be found on Mandiant's blog [here](#).

We can use PEstudio to calculate the Imphash of a sample.





Any malware samples with the same imports in the same order will have the same imphash. This helps in identifying similar samples.

https://bazaar.abuse.ch/browse.php?search=imphash%3A756fdea446bc618b4804509775306c0d

place your favorites here on the favorites bar. [Manage favorites now](#)

**MALWARE bazaar** by ABUSE|

Q Browse Upload Hunting API Export Statistics FAQ About Login

imphash:756fdea446bc618b4804509775306c0d

Search

Search Syntax ?

Search:

Date (UTC)	SHA256 hash	Type	Signature	Tags	Reporter	DL
2022-03-22 12:57	b74f292f11f9d17b83065...	exe	Heodo	Emotet exe Heodo	@JAMESWT_MHT	DL
2020-10-26 08:11	11ee502a6c858e30ccf9b...	exe	Heodo	Emotet Heodo	@Seifreed	DL
2020-10-26 06:54	3116a3f3e2998a57636a7...	exe	Heodo	Emotet Heodo	@Seifreed	DL
2020-10-25 18:46	003a6be25aed1e04592c...	exe	Heodo	Emotet Heodo	@Seifreed	DL
2020-09-13 08:14	ffff844fd04124948412b2...	exe	Heodo	doc Emotet epoch1 Heodo	@Cryptolaemus1	DL
2020-09-13 06:14	d9f666f9ae522c14e1755...	exe	Heodo	Emotet epoch1 exe Heodo	@Cryptolaemus1	DL
2020-09-05 06:25	5d6d506c820f0e461eb91...	exe	Heodo	Emotet Heodo	@FORMALITYDE	DL
2020-09-05 06:21	e177d832488ba0b525d2...	exe	Heodo	Emotet Heodo	@FORMALITYDE	DL
2020-09-05 06:20	d3af1ac985f91eaf735617...	exe	Heodo	Emotet Heodo	@FORMALITYDE	DL
2020-09-05 06:16	b9b640c7fabacd375d819...	exe	Heodo	Emotet Heodo	@FORMALITYDE	DL
2020-09-05 06:15	6b617beb0411f659d3f3a...	exe	Heodo	Emotet Heodo	@FORMALITYDE	DL

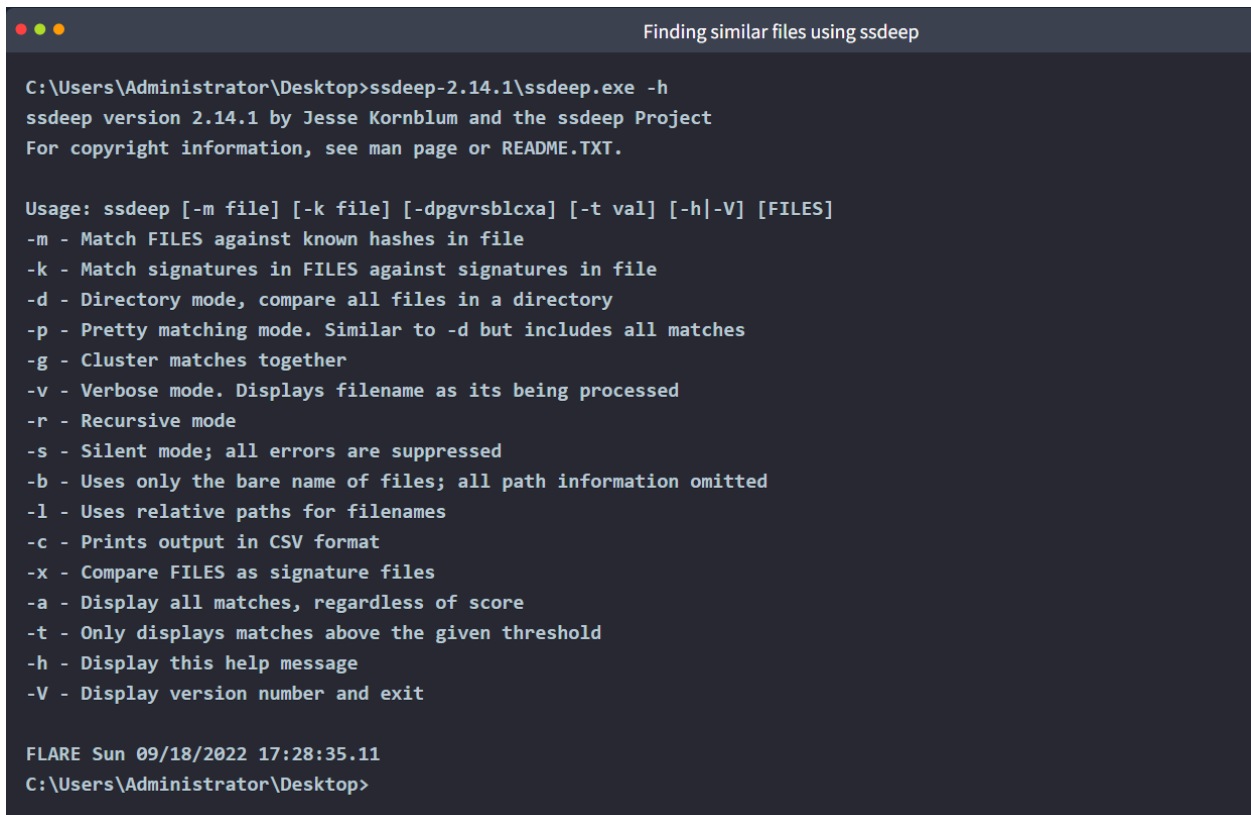
In the above screenshot from [Malware Bazaar](#), all these samples have the same imphash. We can see that all of these samples are classified as the same malware family. We can see that their sha256 hash is vastly different and doesn't provide any

information as to their similarity. However, the same imphash helps us identify that they might belong to the same family.

### Fuzzy hashes/SSDEEP:

Another way to identify similar malware is through fuzzy hashes. A fuzzy hash is a Context Triggered Piecewise Hash (CTPH). This hash is calculated by dividing a file into pieces and calculating the hashes of the different pieces. This method creates multiple inputs with similar sequences of bytes, even though the whole file might be different. More information on SSDEEP can be found on this [link](#).

Multiple utilities can be used in the attached VM to calculate ssdeep, like CyberChef. However, the ssdeep utility has been placed on the Desktop to make it easier. The following command shows the help menu of the utility.



```
C:\Users\Administrator\Desktop>ssdeep-2.14.1\ssdeep.exe -h
ssdeep version 2.14.1 by Jesse Kornblum and the ssdeep Project
For copyright information, see man page or README.TXT.

Usage: ssdeep [-m file] [-k file] [-dpgvrsblcxa] [-t val] [-h|-V] [FILES]
-m - Match FILES against known hashes in file
-k - Match signatures in FILES against signatures in file
-d - Directory mode, compare all files in a directory
-p - Pretty matching mode. Similar to -d but includes all matches
-g - Cluster matches together
-v - Verbose mode. Displays filename as its being processed
-r - Recursive mode
-s - Silent mode; all errors are suppressed
-b - Uses only the bare name of files; all path information omitted
-l - Uses relative paths for filenames
-c - Prints output in CSV format
-x - Compare FILES as signature files
-a - Display all matches, regardless of score
-t - Only displays matches above the given threshold
-h - Display this help message
-V - Display version number and exit

FLARE Sun 09/18/2022 17:28:35.11
C:\Users\Administrator\Desktop>
```

Let's calculate the hashes of all the samples in the mal directory in the attached VM.

```
Calculating ssdeep

C:\Users\Administrator\Desktop>ssdeep-2.14.1\ssdeep.exe mal\*
ssdeep,1,1--blocksize:hash:hash,filename
3072:C3twbyJdvGwRCf/swDQhe0AmN4hMRl37G:8Eac0AmN6C,"C:\Users\Administrator\Desktop\mal\1"
768:fMjB/JpMfHDWqpuXDvod3UmQmv4acY2GS2C9xjwhU:UFQlPSDvoJrbvUfGS2q,"C:\Users\Administrator\Desktop\mal\2"
1536:C3tvICAqw8IKVn2wJk0c8PoYJvGwRCwAL6pILg17vBIQtCnDkbZ3e0AmV2u4hnnM:C3twbyJdvGwRCf/swDQhe0AmN4hM,"C:\Users\Administrator\Desktop\mal\3"
24576:u7Dt1SDAlZvEFZhb57buPPcedeHP5Xlnk03hGL85iw9zVZprY8fWg5r110:80KVizL3cvtk03hmVVZBYu5r1M,"C:\Users\Administrator\Desktop\mal\4"
12288:z+IIs67xrXWxgxmDp1NGvIcGZwWDVHJXuDzKYzIE5P/XiS61YSz8uahDtbNL6WTW:z+PsGlsFGgcQDJJQ,"C:\Users\Administrator\Desktop\mal\5"
24576:UCsTPcqE9S7td0DN+6ybJVCy2pvZHGOzPBjRj4AFsb:UC07tsp+6ybJVChpRjvs,"C:\Users\Administrator\Desktop\mal\6"

FLARE Sun 09/18/2022 17:41:12.38
C:\Users\Administrator\Desktop>
```

We can try the other options shown in the help file per the requirement. When we have the ssdeep hashes, we can match these hashes together to identify similar files. This helps us identify similar files if we have a bulk of data. The documentation link provided above has very good examples of usage. The following terminal window shows one of the examples relevant to our use case to match files. For this, we can use the -d operator. The -r operator runs the ssdeep utility recursively, and the -l operator outputs relative paths of the files.

```
Finding matching files using ssdeep

C:\Users\Administrator\Desktop>ssdeep-2.14.1\ssdeep -l -r -d Incoming Outgoing Trash
Outgoing/Corporate Espionage/Our Budget.doc matches Incoming/Budget 2007.doc (99)
Outgoing/Personnel Mayhem/Your Buddy Makes More Than You.doc matches Incoming/Salaries.doc (45)
Trash/DO NOT DISTRIBUTE.doc matches Outgoing/Plan for Hostile Takeover.doc (88)

FLARE Sun 09/18/2022 17:41:12.38
C:\Users\Administrator\Desktop>
```

The results show files that match each other. The number in the bracket at the end is the percentage of matches among the files.

\*\*\*\*\*

**Answer the questions below:**

**In the samples located at Desktop\mal\ directory in the attached VM, which of the samples has the same imphash as file 3?**

Sample 1:

property	value
md5	<a href="#">6548EEC09F4D8BC65148EE3E5452541C</a>
sha1	<a href="#">7BF46C62D975949FDD6777530940CF6435E8CB90</a>
sha256	<a href="#">6EC74CC0A9B5697EFD3F4CC4D3A21D9FFE6E0187B770990DF8743F8F4F3B2518</a>
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 40 00 00 00 00 00 00
first-bytes-text	M Z .....
file-size	296448 (bytes)
entropy	4.687
imphash	<a href="#">F397831B8900AFF7FBEF2FFDE97C2603</a>
signature	<a href="#">Microsoft Visual C++ 8</a>
entry-point	E8 22 42 00 00 E9 78 FE FF FF 8B FF 55 8B EC 83 EC 28 33 C0 53 8B 5D 0C 56 8B 75 10 57 8B 7D 08 88
file-version	n/a
description	n/a
file-type	<b>executable</b>
cpu	<b>32-bit</b>
subsystem	GUI
compiler-stamp	0x6160F5D9 (Fri Oct 08 18:52:25 2021)
debugger-stamp	<a href="#">0x630E70C4 (Tue Aug 30 13:19:16 2022)</a>
resources-stamp	0x00000000 (empty)
import-stamp	0x00000000 (empty)

## Sample 2:

property	value
md5	<a href="#">19116E822E8178FC103E51FE18C825A4</a>
sha1	<a href="#">F590A8F1B2F337864B166D8CE53A53E77089135B</a>
sha256	<a href="#">B9CB59244AE380B87C41822802FE472BBAB263E701339CE83A3D3896FBBDA8D2</a>
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 40 00 00 00 00 00 00
first-bytes-text	M Z .....
file-size	35408 (bytes)
entropy	5.482
imphash	<a href="#">FDF028EC701B2D6858C7AD4B1EC13C1E</a>
signature	<a href="#">Microsoft Visual C# v7.0 / Basic .NET</a>
entry-point	FF 25 00 20 40 00
file-version	7.411.116.376
description	DCHeV52Me63PW74m606Bobh3Re8
file-type	<b>executable</b>
cpu	<b>32-bit</b>
subsystem	GUI
compiler-stamp	<a href="#">0x8671CB49 (Sun Jun 23 05:49:13 2041)</a>
debugger-stamp	n/a
resources-stamp	0x00000000 (empty)
import-stamp	0x00000000 (empty)

## Sample 3:

property	value
md5	<a href="#">3F18BCD91424D85ED863D921B3E75BDF</a>
sha1	<a href="#">6AD1D6B7EB5E639E2B4DD97A1B0773D1DEB7116A</a>
sha256	<a href="#">04B9949B086249B895FDF4DE1A23E951F17D1F776D746DEA6A2EAA7A370935E9</a>
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
first-bytes-text	M Z .....
file-size	267297 (bytes)
entropy	4.439
imphash	<a href="#">F397831B8900AFF7FBEF2FFDE97C2603</a>
signature	<a href="#">Microsoft Visual C++ 8</a>
entry-point	E8 22 42 00 00 E9 78 FE FF FF 8B FF 55 8B EC 83 EC 28 33 C0 53 8B 5D 0C 56 8B 75 10 57 8B 7D 08 88
file-version	n/a
description	n/a
file-type	<b>executable</b>
cpu	<b>32-bit</b>
subsystem	GUI
compiler-stamp	0x6160F5D9 (Fri Oct 08 18:52:25 2021)
debugger-stamp	<a href="#">0x630E70C4 (Tue Aug 30 13:19:16 2022)</a>
resources-stamp	0x00000000 (empty)
import-stamp	0x00000000 (empty)

#### Sample 4:

property	value
md5	<a href="#">6B9FED8D4830B8E6553BEE3E79BC6181</a>
sha1	<a href="#">5100440948AF36C5FCFE56234B54701298F10FDE</a>
sha256	<a href="#">7A25F3FB62C078CC6B3F5AC524924A0693A858508B55952D2BE3DF5AEFB6650D</a>
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
first-bytes-text	M Z .....
file-size	1318986 (bytes)
entropy	6.428
imphash	n/a
signature	<a href="#">Microsoft Visual C++ v6.0</a>
entry-point	55 8B EC 6A FF 68 E8 66 51 00 68 E0 8C 4D 00 64 A1 00 00 00 00 50 64 89 25 00 00 00 00 83 EC 58 53
file-version	2.4.0.6850
description	mediamonitor gui
file-type	<b>executable</b>
cpu	<b>32-bit</b>
subsystem	GUI
compiler-stamp	0x539670DC (Mon Jun 09 19:43:40 2014)
debugger-stamp	0x4DDD7A2F (Wed May 25 14:52:47 2011)
resources-stamp	0x00000000 (empty)
import-stamp	0x00000000 (empty)

#### Sample 5:

property	value	
md5	<a href="#">1559EB5515EB732DE889DCDFF24662C9</a>	
sha1	<a href="#">69ABF00E7E4AB89A0592380413D3D12CFC714CB9</a>	
sha256	<a href="#">3984EB9BBB5210EAF04A4BCDFCC1512A58DF9D264CF2E8A19377F59D4FD8E55B</a>	
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	
first-bytes-text	MZ ..... @ .....	
file-size	645632 (bytes)	
entropy	7.441	
imphash	<a href="#">2916DDA3C80B39A540B60C072A91A915</a>	
signature	<a href="#">Microsoft Visual Studio .NET</a>	
entry-point	FF 25 00 20 40 00	
file-version	1.0.0.0	
description	Uno	
file-type	<b>executable</b>	
cpu	<b>32-bit</b>	
subsystem	GUI	
compiler-stamp	0x60FF5B80 (Mon Jul 26 18:04:00 2021)	
debugger-stamp	n/a	
resources-stamp	0x00000000 (empty)	
import-stamp	0x00000000 (empty)	

Sample 6:

property	value	
md5	<a href="#">7DEBB7345BCFFDF1C5328F51524EE16B</a>	
sha1	<a href="#">86875DEEA552651DAC3C7360607E66971595A8E2</a>	
sha256	<a href="#">2606E7850597F8C4D22E7C28CCAE1FEDD98DC91B307EA56AD9C69B5EED7BC363</a>	
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	
first-bytes-text	MZ ..... @ .....	
file-size	870912 (bytes)	
entropy	6.507	
imphash	<a href="#">7997426B781E320BCC031754360DF27F</a>	
signature	n/a	
entry-point	48 8B C4 48 89 58 08 48 89 68 10 48 89 70 18 48 89 78 20 41 56 48 83 EC 20 4C 8B F1 49 8B F0 33 C9	
file-version	n/a	
description	n/a	
file-type	<b>executable</b>	
cpu	<b>64-bit</b>	
subsystem	GUI	
compiler-stamp	0x5721FD88 (Thu Apr 28 05:09:44 2016)	
debugger-stamp	n/a	
resources-stamp	0x00000000 (empty)	
import-stamp	0x00000000 (empty)	

Answer: **1**

Using the ssdeep utility, what is the percentage match of the above-mentioned files?

```
C:\Users\Administrator\Desktop\mal\1 matches C:\Users\Administrator\Desktop\mal\3 (93)
```

Answer: **93**

## Signature-based Detection

In the previous task, we learned how hashes could identify identical files. We also found out that hashes can be changed by changing even a single byte of data in a file and how specific hashes like imphash and ssdeep help us identify file similarities. While using imphash or ssdeep provides a way to identify if some files are similar, sometimes we just need to identify if a file contains the information of interest. Hashes are not the ideal tool to perform this task.

### **Signatures:**

Signatures are a way to identify if a particular file has a particular type of content. We can consider a signature as a pattern that might be found inside a file. This pattern is often a sequence of bytes in a file, with or without any context regarding where it is found. Security researchers often use signatures to identify patterns in a file, identify if a file is malicious, and identify suspected behavior and malware family.

### **Yara rules:**

Yara rules are a type of signature-based rule. It is famously called a pattern-matching swiss army knife for malware researchers. Yara can identify information based on binary and textual patterns, such as hexadecimal and strings contained within a file.

TryHackMe has a [dedicated room](#) for Yara rules if it interests you.

The security community publishes a [repository](#) of open-source Yara rules that we can use as per our needs. When analyzing malware, we can use this repository to dig into the community's collective wisdom. However, while using these rules, please keep in mind that some might depend on context. Some others might just be used for the identification of patterns that can be non-malicious as well. Hence, just because a rule hits doesn't mean the file is malicious. For a better understanding, please read the documentation for the particular rule to identify the use case where it will be applicable in the best possible manner.

### **Proprietary Signatures - AntiVirus Scans:**

Besides the open-source signatures, Antivirus companies spend lots of resources to create proprietary signatures. The advantage of these proprietary signatures is that since they have to be sold commercially, there are lesser chances of False Positives (FPs, when a signature hits a non-malicious file). However, this might lead to a few False Negatives (FNs, when a malicious file does not hit any signature).

Antivirus scanning helps identify if a file is malicious with high confidence. Antivirus software will often mention the signature that the file has hit, which might hint at the file's functionality. However, we must note that despite their best efforts, every AV product in the market has some FPs and some FNs. Therefore, when analyzing malware, it is

prudent to get a verdict from multiple products. The [VirusTotal](#) website makes this task easier for us, where we can find the verdict about a file from 60+ AV vendors, apart from some very useful information. We also touched upon this topic in our Intro to Malware Analysis room. Please remember, if you are analyzing a sensitive file, it is best practice to search for its hash on VirusTotal or other scanning websites instead of uploading the file itself. This is done to avoid leaking sensitive information on the internet and letting a sophisticated attacker know that you are analyzing their malware.

Since we have covered Yara rules in detail in the Yara room and VirusTotal scanning in the Intro to malware analysis room, we will not cover them again here. However, the FLARE VM has another very cool tool that can be used for signature scanning.

### **Capa:**

Capa is another open-source tool created by Mandiant. This tool helps identify the capabilities found in a PE file. Capa reads the files and tries to identify the behavior of the file based on signatures such as imports, strings, mutexes, and other artifacts present in the file. For further detail into the background of Capa, we can visit its [Github page](#) or Mandiant's [blog post](#) introducing Capa.

Using Capa is simple. On the command prompt, we just point capa to the file we want to run it against.

```
C:\Users\Administrator\Desktop>capa mal.exe
```

The -h operator shows detailed options.



```
C:\Users\Administrator\Desktop>capa -h
usage: capa.exe [-h] [--version] [-v] [-vv] [-d] [-q] [--color {auto,always,never}] [-f {auto,pe,sc32,sc64,freeze}] [-b {vivisect,smda}]

The FLARE team's open-source tool to identify capabilities in executable files.

positional arguments:
  sample                path to sample to analyze

optional arguments:
  -h, --help            show this help message and exit
  --version            show program's version number and exit
  -v, --verbose         enable verbose result document (no effect with --json)
  -vv, --vverbose      enable very verbose result document (no effect with --json)
  -d, --debug          enable debugging output on STDERR
  -q, --quiet          disable all output but errors
  --color {auto,always,never}
                        enable ANSI color codes in results, default: only during interactive session
  -f {auto,pe,sc32,sc64,freeze}, --format {auto,pe,sc32,sc64,freeze}
                        select sample format, auto: (default) detect file type automatically, pe: Windows PE file, sc32: 32-bit shellcode
                        previously frozen by capa
  -b {vivisect,smda}, --backend {vivisect,smda}
                        select the backend to use
  -r RULES, --rules RULES
                        path to rule file or directory, use embedded rules by default
  -t TAG, --tag TAG    filter on rule meta field values
  -j, --json           emit JSON instead of text

By default, capa uses a default set of embedded rules.
You can see the rule set here:
  https://github.com/fireeye/capa-rules

To provide your own rule set, use the '-r' flag:
  capa --rules /path/to/rules suspicious.exe
  capa -r      /path/to/rules suspicious.exe

examples:
  identify capabilities in a binary
  capa suspicious.exe

  identify capabilities in 32-bit shellcode, see '-f' for all supported formats
  capa -f sc32 shellcode.bin

  report match locations
  capa -v suspicious.exe

  report all feature match details
  capa -vv suspicious.exe

  filter rules by meta fields, e.g. rule name or namespace
  capa -t "create TCP socket" suspicious.exe

FLARE Sun 09/18/2022 18:10:17.58
C:\Users\Administrator\Desktop>
```

We can test drive capa by running it against the binaries in the Desktop\mal directory. Please note that capa might take some time to complete the analysis. An example output is below.

```
C:\Users\Administrator\Desktop>capa mal\1
loading : 100%| 485/485 [00:00<00:00, 1552.05 rules/s]
matching: 100%| 288/288 [00:12<00:00, 22.23 functions/s]

+-----+-----+
| md5      | 6548eec09f4d8bc6514bee3e5452541c |
| sha1     | 7be46c62d975949fdd6777530940cf6435e8cb90 |
| sha256   | 6ec74cc0a9b5697efd3f4cc4d3a21d9ffe0187b770990df8743fbf4f3b2518 |
| path     | mal\1 |
+-----+-----+

+-----+-----+
| ATT&CK Tactic | ATT&CK Technique |
+-----+-----+
| DEFENSE EVASION | Obfuscated Files or Information::Indicator Removal from Tools [T1027.005] |
|                  | Obfuscated Files or Information [T1027] |
| DISCOVERY       | Application Window Discovery [T1010] |
|                  | System Information Discovery [T1082] |
| EXECUTION       | Command and Scripting Interpreter [T1059] |
|                  | Shared Modules [T1129] |
+-----+-----+

+-----+-----+
| MBC Objective | MBC Behavior |
+-----+-----+
| ANTI-STATIC ANALYSIS | Disassembler Evasion::Argument Obfuscation [B0012.001] |
| CRYPTOGRAPHY         | Encrypt Data::RC4 [C0027.009] |
|                      | Generate Pseudo-random Sequence::RC4 PRGA [C0021.004] |
| FILE SYSTEM          | Delete File [C0047] |
|                      | Read File [C0051] |
|                      | Write File [C0052] |
| OPERATING SYSTEM     | Console [C0033] |
| PROCESS              | Allocate Thread Local Storage [C0040] |
|                      | Set Thread Local Storage Value [C0041] |
|                      | Terminate Process [C0018] |
+-----+-----+
```

CAPABILITY	NAMESPACE
contain obfuscated stackstrings	anti-analysis/obfuscation/string/stackstring
encrypt data using RC4 PRGA	data-manipulation/encryption/rc4
contains PDB path	executable/pe/pdb
contain a resource (.rsrc) section	executable/pe/section/rsrc
accept command line arguments	host-interaction/cli
manipulate console	host-interaction/console
query environment variable	host-interaction/environment-variable
delete file	host-interaction/file-system/delete
read file	host-interaction/file-system/read
write file (2 matches)	host-interaction/file-system/write
enumerate gui resources	host-interaction/gui
get disk information	host-interaction/hardware/storage
get hostname	host-interaction/os/hostname
get thread local storage value (3 matches)	host-interaction/process
set thread local storage value (2 matches)	host-interaction/process
terminate process (5 matches)	host-interaction/process/terminate
link function at runtime (8 matches)	linking/runtime-linking
link many functions at runtime	linking/runtime-linking
parse PE exports (2 matches)	load-code/pe
parse PE header (4 matches)	load-code/pe

FLARE Sun 09/18/2022 18:34:13.15  
C:\Users\Administrator\Desktop>

We can see that Capa has mapped the identified capabilities according to the MITRE ATT&CK framework and Malware Behavior Catalog (MBC). In the last table, we see the capabilities against the matched signatures and the number of signatures that have found a hit against these capabilities. As we might see, it also tells us if there are obfuscated stackstrings in the sample, allowing us to identify if running FLOSS against the sample might be helpful. To find out more information about the sample, we can use the -v or the -vv operator, which will show us the results in verbose or very verbose mode, identifying addresses where we might find the said capability.

Now let's use capa to analyse the file Desktop\mal\4 and answer the following questions.

\*\*\*\*\*

**Answer the questions below:**

C:\Users\Administrator\Desktop>capa mal\4		485/485 [00:00<00:00, 1633.68 rules/s]	7123/7123 [02:02<00:00, 58.35 functions/s]
loading : 100%			
matching: 100%			
md5	6b9fed8d483b08e653bee7e79bc6181		
sha1	5100440948af36c5fcfe56234b54701298f10fde		
sha256	7a25f3fb02c078cc6b3f5ac524924a0693a858508b55952d2be3df5aeeb06650d		
path	mal\4		
ATT&CK Tactic	ATT&CK Technique		
COLLECTION	Input Capture::Keylogging [T1056.001]		
DEFENSE EVASION	File and Directory Permissions Modification [T1222]		
	Hide Artifacts::Hidden Window [T1564.003]		
	Obfuscated Files or Information::Indicator Removal from Tools [T1027.005]		
	Obfuscated Files or Information [T1027]		
	Virtualization/Sandbox Evasion::System Checks [T1497.001]		
DISCOVERY	Virtualization/Sandbox Evasion::User Activity Based Checks [T1497.002]		
	Application Window Discovery [T1010]		
	File and Directory Discovery [T1083]		
	Process Discovery [T1057]		
	Query Registry [T1012]		
	System Information Discovery [T1082]		
EXECUTION	Command and Scripting Interpreter [T1059]		
	Shared Modules [T1129]		
PERSISTENCE	Boot or Logon Autostart Execution::Registry Run Keys / Startup Folder [T1547.001]		

MBC Objective	MBC Behavior
ANTI-BEHAVIORAL ANALYSIS	Debugger Detection::Process Environment Block BeingDebugged [B0001.035]
	Debugger Detection::Process Environment Block NtGlobalFlag [B0001.036]
	Debugger Detection::Software Breakpoints [B0001.025]
	Virtual Machine Detection::Human User Check [B0009.012]
	Virtual Machine Detection::Instruction Testing [B0009.029]
ANTI-STATIC ANALYSIS	Disassembler Evasion::Argument Obfuscation [B0012.001]
COLLECTION	Keylogging::Polling [F0002.002]
COMMUNICATION	HTTP Communication::Read Header [C0002.014]
DATA	Encoding::XOR [C0026.002]
	Non-Cryptographic Hash::MurmurHash [C0030.001]
DEFENSE EVASION	Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]
FILE SYSTEM	Create Directory [C0046]
	Delete File [C0047]
	Get File Attributes [C0049]
	Read File [C0051]
	Set File Attributes [C0050]
	Write File [C0052]
OPERATING SYSTEM	Environment Variable::Set Variable [C0034.001]
	Registry::Create Registry Key [C0036.004]
	Registry::Delete Registry Key [C0036.002]
	Registry::Delete Registry Value [C0036.007]
	Registry::Open Registry Key [C0036.003]
	Registry::Query Registry Value [C0036.006]
	Registry::Set Registry Key [C0036.001]
PROCESS	Allocate Thread Local Storage [C0040]
	Create Mutex [C0042]
	Create Process [C0017]
	Create Thread [C0038]
	Resume Thread [C0054]
	Set Thread Local Storage Value [C0041]
	Suspend Thread [C0055]
	Terminate Process [C0018]

CAPABILITY	NAMESPACE
check for PEB BeingDebugged flag	anti-analysis/anti-debugging/debugger-detection
check for PEB NtGlobalFlag flag (2 matches)	anti-analysis/anti-debugging/debugger-detection
check for software breakpoints (4 matches)	anti-analysis/anti-debugging/debugger-detection
check for unmoving mouse cursor	anti-analysis/anti-vm/vm-detection
execute anti-VM instructions (86 matches)	anti-analysis/anti-vm/vm-detection
contain obfuscated stackstrings (18 matches)	anti-analysis/obfuscation/string/stackstring
log keystrokes via polling (2 matches)	collection/keylog
check HTTP status code	communication/http/client
encode data using XOR (4 matches)	data-manipulation/encoding/xor
hash data using murmur3 (3 matches)	data-manipulation/hashing/murmur
authenticate HMAC	data-manipulation/hmac
contains PDB path	executable/pe/pdb
contain a resource (.rsrc) section	executable/pe/section/rsrc
extract resource via kernel32 functions (4 matches)	executable/resource
accept command line arguments	host-interaction/cli
query environment variable (2 matches)	host-interaction/environment-variable
set environment variable	host-interaction/environment-variable
get common file path (3 matches)	host-interaction/file-system
get file system object information	host-interaction/file-system
create directory	host-interaction/file-system/create
delete file	host-interaction/file-system/delete
check if file exists	host-interaction/file-system/exists
enumerate files via kernel32 functions (2 matches)	host-interaction/file-system/files/list
get file attributes (2 matches)	host-interaction/file-system/meta
get file size	host-interaction/file-system/meta
set file attributes (2 matches)	host-interaction/file-system/meta
move file	host-interaction/file-system/move
read .ini file (2 matches)	host-interaction/file-system/read
read file (2 matches)	host-interaction/file-system/read
write file (3 matches)	host-interaction/file-system/write
enumerate gui resources	host-interaction/gui
set application hook (6 matches)	host-interaction/gui
hide graphical window	host-interaction/gui/window/hide
get number of processors (3 matches)	host-interaction/hardware/cpu
get disk information (3 matches)	host-interaction/hardware/storage
get disk information (3 matches)	host-interaction/hardware/storage
print debug messages	host-interaction/log/debug/write-event
create mutex (2 matches)	host-interaction/mutex
check OS version (2 matches)	host-interaction/os/version
get process heap flags (7 matches)	host-interaction/process
get process heap force flags (5 matches)	host-interaction/process
get thread local storage value (7 matches)	host-interaction/process
set thread local storage value (6 matches)	host-interaction/process
create process	host-interaction/process/create
terminate process (2 matches)	host-interaction/process/terminate
query or enumerate registry value (3 matches)	host-interaction/registry
delete registry key	host-interaction/registry/delete
delete registry value (2 matches)	host-interaction/registry/delete
create thread	host-interaction/thread/create
resume thread (2 matches)	host-interaction/thread/resume
suspend thread	host-interaction/thread/suspend
access PEB ldr_data	linking/runtime-linking
link function at runtime (6 matches)	linking/runtime-linking
link many functions at runtime	linking/runtime-linking
parse PE exports	load-code/pe
parse PE header (6 matches)	load-code/pe
persist via Run registry key	persistence/registry/run

How many matches for anti-VM execution techniques were identified in the sample?

execute anti-VM instructions (86 matches)	anti-analysis/anti-vm/vm-detection
---	------------------------------------

Answer: 86

Does the sample have the capability to suspend or resume a thread? Answer with Y for yes and N for no.

```
| create thread | host-interaction/thread/create  
| resume thread (2 matches) | host-interaction/thread/resume  
| suspend thread | host-interaction/thread/suspend  
| ... | ...
```

Answer: **Y**

What MBC behavior is observed against the MBC Objective 'Anti-Static Analysis'?

```
| ANTI-STATIC ANALYSIS | Disassembler Evasion::Argument Obfuscation [B0012.001]  
| COLLECTION | ...
```

Answer: **Disassembler Evasion::Argument Obfuscation [B0012.001]**

At what address is the function that has the capability 'Check HTTP Status Code'?

Following the hint for the question I used the verbose flag for capa and captured the results into a text file. From there I was able to find the “check HTTP status code” section and find the address.

```
check HTTP status code  
namespace communication/http/client  
author moritz.raabe@fireeye.com  
scope function  
mbc Communication::HTTP Communication::Read Header [C0002.014]  
examples 54ac78733552a62d1d05ea4ba3fc604bb49fe000d7fc948da45335b726e64d75:0x10001a20  
function @ 0x486921  
and:  
or:  
  number: 0x13 = HTTP_QUERY_STATUS_CODE @ 0x486E5A, 0x486EDC  
subscope:  
and:  
  or:  
    mnemonic: test @ 0x486F10  
  or:  
    number: 0xC8 = OK @ 0x486F00
```

Answer: **0x486921**

## Leveraging the PE Header

So far in this room, we have covered techniques that work regardless of the file type of the malware. However, those techniques are a little hit-and-miss, as they don't always provide us with deterministic information about the malware. The PE headers provide a little more deterministic characteristics of the sample, which tells us much more about the sample.

The PE header:

The programs that we run are generally stored in the executable file format. These files are portable because they can be taken to any system with the same Operating System and dependencies, and they will perform the same task on that system. Therefore, these files are called Portable Executables (PE files). The PE files consist of a sequence of bits stored on the disk. This sequence is in a specific format. The initial bits define the characteristics of the PE file and explain how to read the rest of the data. This initial part of the PE file is called a PE header.

Several tools in the FLARE VM can help us analyze PE headers. PESTudio is one of them. We already familiarized ourselves with PESTudio in a previous task, so we will just use that in this task as well.

The PE header contains rich information useful for malware analysis. We will learn about this data in detail in the [Dissecting PE headers](#) room. However, we can get the following information from a PE header as an overview.

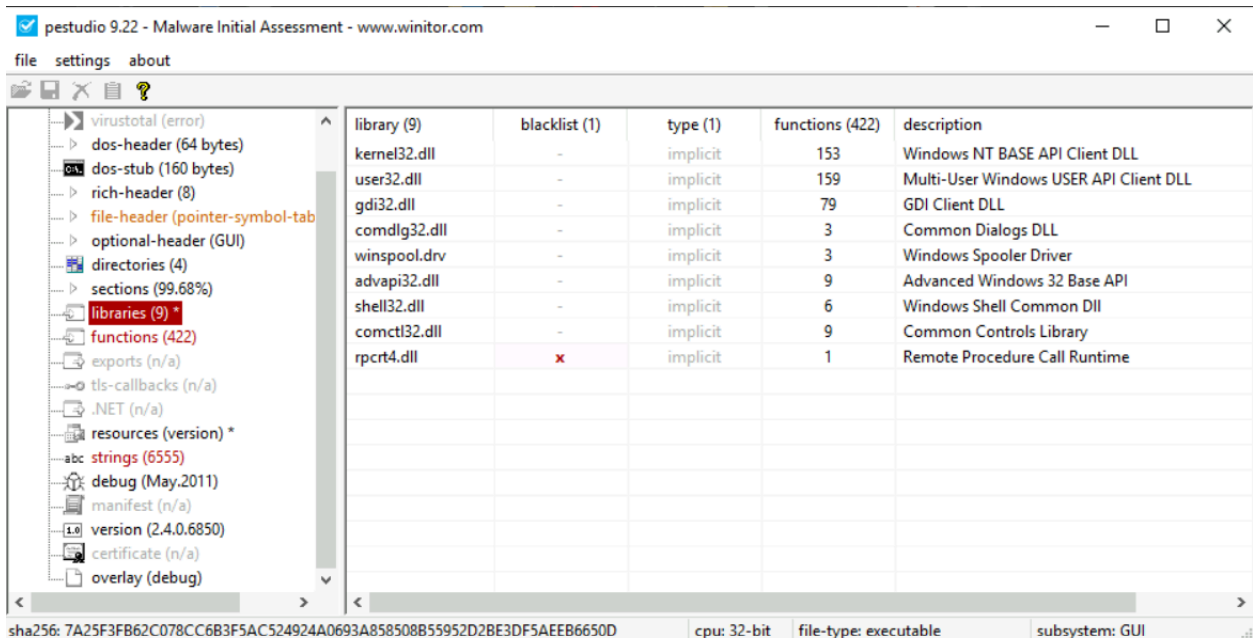
### **Linked Libraries, imports, and functions:**

A PE file does not contain all of its code to perform all the tasks. It often reuses code from libraries, often provided by Microsoft as part of the Windows Operating System. Often, certain functions from these libraries are imported by the PE file. The PE header contains information about the libraries that a PE file uses and the functions it imports from those libraries. This information is very useful. A malware analyst can look at the libraries and functions that a PE file imports and get a rough idea of the functionality of a malware sample. For example, if a malware sample imports the `CreateProcessA` function, we can assume that this sample will create a new process.

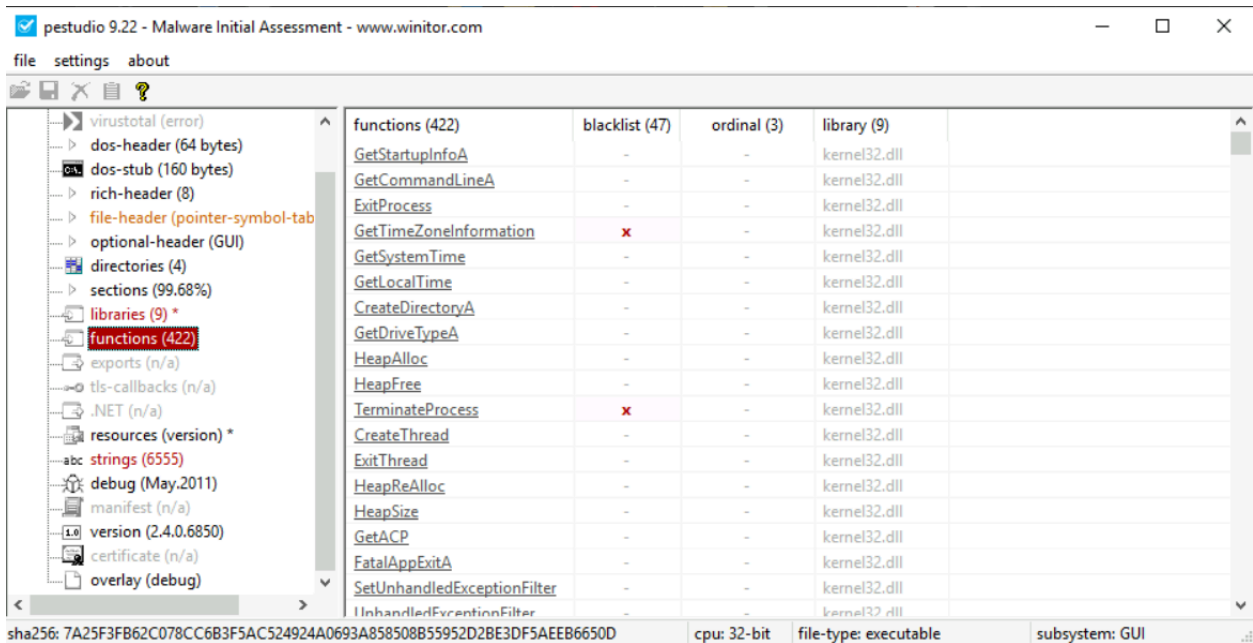
Similarly, other functions can provide further information about the sample. However, it must be noticed that we don't know the context in which these functions are called by just looking at the PE headers. We need to dig deeper into that, which we will cover in the upcoming rooms.

PEStudio has a libraries option in the right pane, which, when selected, shows us the libraries that a PE file will use.





The functions option just below shows the functions imported from these libraries.



## Identifying Packed Executables:

As we have seen so far, static analysis bares a lot of information that can be used against the malware. Malware authors understand this as a problem. Therefore, they often go to great lengths to thwart analysis. One of the ways they can do this is by packing the original sample inside a shell-type code that obfuscates the properties of the actual malware sample. This technique is called packing, and the resultant PE file is called a packed PE file. Packing greatly reduces the effectiveness of some of the malware analysis techniques we have learned about so far. For example, if we try to



search for strings in a packed executable, we might not find anything useful because of packing. Similarly, searching for similar samples using ssdeep might return more samples packed with the same packer instead of samples behaviorally similar to the sample of interest. Some signatures might also be evaded due to a malware sample being packed.

As we will see in the upcoming [Dissecting PE headers](#) room, we can identify packed executables by analyzing the PE header of a malware sample. The PE header contains important information such as the number of sections, permissions of different sections, sizes of the sections, etc. This information can give us pointers to help identify if the malware sample is packed and, if so, which type of packer has packed the executable.

\*\*\*\*\*

**Answer the questions below:**

**Open the sample Desktop\mal\4 in PEstudio. Which library is blacklisted?**

library (9)	blacklist (1)	type (1)	functions (422)	description
kernel32.dll	-	implicit	153	Windows NT BASE API Client DLL
user32.dll	-	implicit	159	Multi-User Windows USER API Client DLL
gdi32.dll	-	implicit	79	GDI Client DLL
comdlg32.dll	-	implicit	3	Common Dialogs DLL
winspool.drv	-	implicit	3	Windows Spooler Driver
advapi32.dll	-	implicit	9	Advanced Windows 32 Base API
shell32.dll	-	implicit	6	Windows Shell Common Dll
comctl32.dll	-	implicit	9	Common Controls Library
rpcrt4.dll	x	implicit	1	Remote Procedure Call Runtime

Answer: **rpcrt4.dll**

**What does this dll do?**

Answer: **Remote Procedure Call Runtime**

## Conclusion

That wraps up our basic static analysis room. So far, we have learned the following:

- Lab setup for malware analysis
- Searching for strings and obfuscated strings
- Fingerprinting malware using hashes and identifying similar samples using imphash and ssdeep
- Using signature-based detection like Yara and Capa
- Identifying artifacts from the PE header