The Reverse Engineering of Trojan-Emotet Malware

Quillen Flanigan

Conjure-2

7/16/2023

-In accordance with the ACE core values, I have neither given nor received unauthorized aid on this paper.

Abstract

Reverse engineering is the process of analyzing machines, processes or software to gain an understanding of its behavior and effects. This process is often accomplished with minimal knowledge of how the software was originally constructed. Reverse engineers must have the ability to develop an advanced intuition on an object's malicious intentions despite this lack of insight. We can then use this knowledge to understand how a piece of malware was built, its effects, severity of those effects, defenses, and who it was built by. In this problem, we were given a sample file and tasked with identifying and classifying the malware's family and capabilities, along with developing a remediation strategy after our analysis. Through the reverse engineering of our sample and analyzing our sample both statically and dynamically, we classified our sample as Trojan-Emotet malware. Using various static and dynamic tools to study the malware before and after it was executed, we gained an awareness of the malware's complexity, effects, defenses, and an approximation of its original source code, along with developing remediation strategies for compromised systems.

1. Introduction

Reverse engineering is the process of acquiring a deep understanding of an object with minimal insight on how the object was originally constructed. It is the reconstruction of a crime scene, gathering details about the origins and purpose of an object without explicit awareness of the object's creation. This reverse engineering process can be directly applied to the field of cyber security to analyze malware in an attempt to discover its harmful properties and effects. As cyber warfare becomes more volatile, we must develop a stronger understanding of our adversaries abilities and the effects of their malicious attacks. There were a reported 5.5 billion malware attacks in 2022, illustrating the need for this advanced analysis [1]. If we are able to reconstruct a piece of malware from its low-level form, we can study its potential behaviors and effects, allowing us to enhance both our offensive and defensive capabilities against future malware attacks.

This challenge problem introduces the concept of using reverse engineering to analyze a specific sample of malware. We were given a malicious binary executable file and were tasked with studying this malware to discover its capabilities and classification, while also creating a recommendation for remediating compromised systems. We then use our understanding of the sample's traits to enhance our own tools for designing malware and defending against it. Cyber operators must prioritize the superiority of our capabilities compared to our adversaries. Reverse engineering allows us to continue building upon the existing abilities in the field of malware analysis with these superior capabilities.

1.1. Assumptions and Limitations

This problem involves various assumptions and limits on our process to analyze the malware. Firstly, we are assuming the competency and correctness of the software and tools we used to help analyze the malware. Reverse engineering often depends on using software to help analyze a sample, as computers have superior pattern recognition to humans. These software tools can detect and comprehend patterns within the binary code of malware and give us details that humans would not be able to see. However, using this software for analysis on the strings, imports, dependencies, packer detection and defenses of the malware causes us to be dependent on these tools. If the tools happened to be incorrect, our analysis of the sample would be ineffective. Similarly, we are assuming the tools we use will return not only correct information, but enough information to allow us to gain a comprehensive understanding of the malware's nature.

Secondly, we are assuming the analysis of the malware does not take too much time in order to gain a full understanding and analyze all of its behaviors. Modern malware sometimes has the ability to remain dormant after its initial execution, then executing after a certain time frame, sometimes taking multiple hours or days. For this challenge problem, we are assuming the malware executes fully within the span of about an hour to produce a full analysis within a reasonable scope.

Furthermore, we are assuming the malware does not have the complete ability to avoid dynamic analysis. Modern malware may have the ability to detect if it is being analyzed within a dynamic tool such as a debugger, and may be able to enter a dormant state to avoid detection. We are prepared for some level of anti-debugging ability, but we are assuming the malware is still able to be analyzed without avoiding detection completely.

Lastly, we are limited by the environment used to analyze the malware. Due to the inherent danger of dynamic analysis on malware, we must execute and study the malware within a virtual machine (VM) to avoid corrupting our machines. We are also limited by the operating system (OS) of our VM as the malware is a Windows binary executable. This means we can only operate within a Windows VM, limiting the tools and software at our disposal.

2. Background

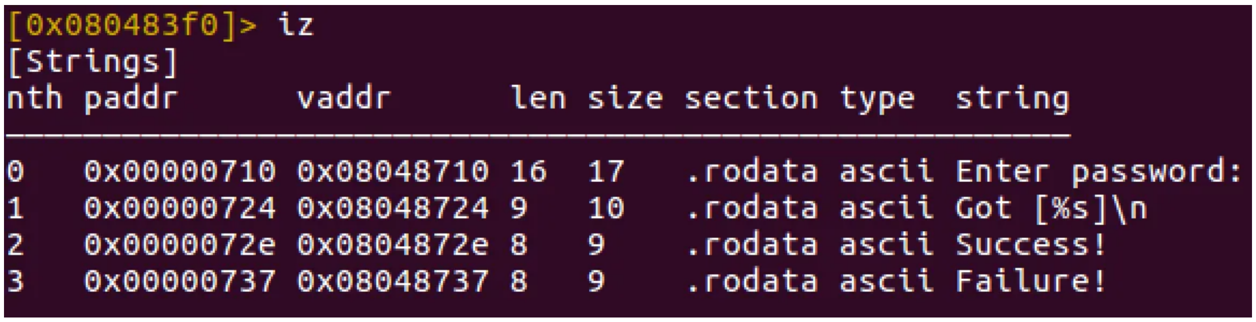
Reverse engineering can be broken down into two stages with static analysis and dynamic analysis. Static analysis refers to studying the properties of the object before it is executed. In the case of reverse engineering malware, it is important to understand the basic properties of the sample before it is executed (static) and also analyze the effects and behaviors of the malware after it is executed (dynamic). The two stages of reverse engineering each come with unique tools and strategies we can use to gather a complete view of the sample.

2.1. Static Analysis

Static analysis refers to studying the sample before it is executed, and is often the first stage of reverse engineering. This allows engineers to build an understanding of what libraries the malware imports, how information and instructions would flow when the process is executed, if the malware is packed, and the potential behaviors of the malware [2]. The first steps in static analysis involve breaking down the malware to analyze the code in terms of strings, the code's dependencies, and the level of packing within the binary machine code.

2.1.1. Strings

The recommended first step in static analysis is constructing the binary executable into a collection of strings. This allows engineers to view the code in a human-readable format to get an initial understanding of potential packing, code instructions, and what functions the code uses. Analysts use these strings to build a foundation of knowledge used in further examination of the sample, and can be used to view a general idea of how the code will behave.



```
[0x080483f0]> iz
[Strings]
nth paddr      vaddr      len size section type  string
―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x00000710 0x08048710 16  17   .rodata ascii Enter password:
1   0x00000724 0x08048724 9   10   .rodata ascii Got [%s]\n
2   0x0000072e 0x0804872e 8   9    .rodata ascii Success!
3   0x00000737 0x08048737 8   9    .rodata ascii Failure!
```

Figure 1. Depiction of converting machine code instructions to strings [3]

The process of converting binary or machine code instructions to strings is depicted above in Fig. 1. This allows the analysts to be able to read the code in a much more efficient manner, as engineers often cannot gather enough information from just machine code. Thus, using strings allows analysts to gain insight on what each instruction of a malware sample may do before the code is executed.

2.1.2. Packer Detection, Obfuscation and Entropy

The act of "packing" a file is often used by malware developers to avoid detection by reverse engineers. It refers to the concept of compressing the data to both make it easier to transfer over a network and make the sample more difficult to read, so reverse engineers are not able to effectively analyze the source code [4]. Packing also decreases the effectiveness of pattern-based malware detection as they cannot observe the true signatures of the code. This concept of altering and encrypting the malware to avoid detection is known as obfuscation, and is a common modern strategy used by malware developers. Obfuscation then produces code with high entropy, measuring the level of randomness and disorder within a given sample. Files that are compressed and obfuscated to avoid detection often result in very high measures of entropy. Thus, engineers can study entropy graphs of a sample to determine packing and obfuscation [5].

Furthermore the common usage of packing, reverse engineers often use several tools known as packer detectors to recognize any packing within a sample. The analysis of strings as mentioned above also gives insight on if a sample has been packed as packing is obvious when processed in a more readable format [6].

2.1.3. Dependencies

Malware is constructed by combining several different functions and libraries that may not be inherently malicious, yet using them in a malicious manner. For example, a simple function such as opening an internet connection is not inherently dangerous, but malicious programs may use this function repeatedly to issue a distributed denial of service attack or communicate with an external system for heightened access to a target system. There are several functions that are often taken advantage of by malware such as file deletion, internet connectivity and spawning child processes. Reverse engineers can often predict malicious intent by using dependency-detection tools to search for these functions and libraries that a sample may import or export.

2.1.4. PEstudio

Reverse engineers utilize several tools to make the process of statically analyzing malware more efficient. One of the most effective tools used by modern engineers is known as PEstudio. This software tool allows analysts to view string data and origins of each string within the malware, view imports and exports of the source code, and see common indicators within the code that depict malicious intent. The ability of PEstudio to display this data to the analyst in an efficient manner makes it an effective initiator of static analysis.

2.1.5. VirusTotal

Through analyzing malware, reverse engineers have been able to categorize malware into families and store records of malware attacks in databases for future reference. One of the more popular databases is known as VirusTotal, often linked with PEstudio to help reverse engineers research a given malware sample. VirusTotal provides data about the family and type of the sample, along with the known behaviors and effects of the malware such as connectivity abilities, file modification, and process spawning. This database uses dozens of different malware detectors to form a comprehensive scanning of the malware before we execute it. Understanding a malware's capabilities during static analysis can be a useful and efficient way to remediate and prevent future attacks.

2.2. Dynamic Analysis

After analyzing a sample statically, reverse engineers often move to executing the code and directly studying the behaviors and effects of the malware while infecting a system.

Dynamic analysis often takes place within a VM to avoid the malicious effects on our own systems. Analysts then use tools to study the effects while the malware is currently running, gaining an insight on the abilities of the sample that cannot be understood during static analysis. Tools such as process monitors, registry recorders and disassemblers help build a complete perspective about how the sample is affecting the computer at every level.

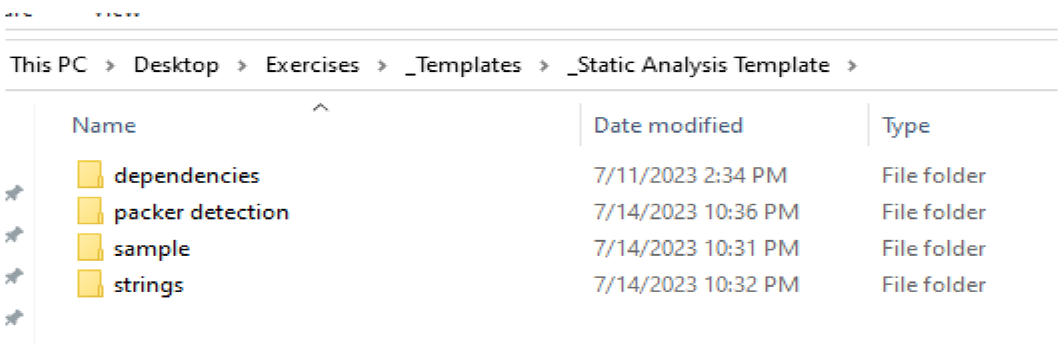
2.2.1. Dynamic Analysis Tools

The software and tools reverse engineers use are essential during dynamic analysis. They allow analysts to record the effects on the system within different states of a malware's execution. Dynamic tools such as ProcMon, PE Detective, 'What's Running', ProcDot, and Ghidra aid in an engineer's ability to dynamically analyze a given sample. ProcMon, or Process-Monitor, allows engineers to capture every process that runs on the system within a given timeframe, and then filter this traffic by process name, type, and behavior. PE detective scans input files and identifies signatures that may indicate malicious intent. 'What's Running' is used to record every process that executes and runs within a given timeframe. This helps record what processes a malicious program might spawn, and its effects on any running processes before the malware's execution. ProcDot helps to study the execution path of a given file once it runs and builds a graph of all the systems the program influences. This helps engineers study the malware's effects on the system as a whole and track where the malware is traveling post-execution. Lastly, Ghidra is a disassembler that engineers use to reconstruct an approximation of the source code that our binary sample was compiled from. This ability to build the source code from an executable is a very effective tool and allows engineers to gain a much deeper insight into how the code is operating, its specific behavior on the system, and build an understanding into how to remediate the effects of the malware. Ghidra serves to link each machine instruction from the binary format to a corresponding instruction in the source code, allowing engineers to see what each instruction is doing as the malware executes.

3. Methods

The first step in our analysis of the provided malware was static analysis, specifically breaking down the executable file into strings. This allows us to gain an initial understanding of some of the imported functions and libraries along with some potential effects of the code. When we begin with studying the malware in a static state, we can build a foundation of understanding the sample code before it even executes. This can allow us to categorize the sample as either malicious or benign, and can give us strong indications of what to expect when executing the file for the first time. This pre-execution awareness can allow us to prepare for the potential harm of a malicious file, and can teach us both what to look out for and where to look within our testing environment to view the effects of the sample.

Within our static analysis folder, we had folders named strings, dependencies, and packer detection. This initial directory structure depicted in Fig. 2 allowed for efficient analysis of the code with all of our static analysis tools.

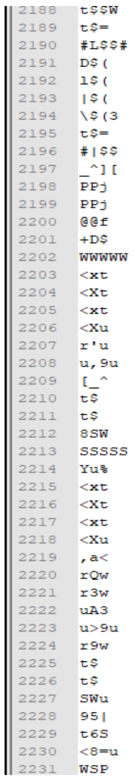

This PC > Desktop > Exercises > _Templates > _Static Analysis Template >

Name	Date modified	Type
dependencies	7/11/2023 2:34 PM	File folder
packer detection	7/14/2023 10:36 PM	File folder
sample	7/14/2023 10:31 PM	File folder
strings	7/14/2023 10:32 PM	File folder

Figure 2. Initial directory structure for static analysis

We then placed our sample named ‘267.bin’ into each directory, allowing us to quickly travel and use the tools within each directory. We began with strings, entering the strings directory in our command line terminal. For this challenge problem, our tool to analyze the malware’s strings is called strings64.exe. This executable process takes in another executable and converts the machine code into readable strings. Using the command “strings64.exe 267.bin > strings_final.txt”, we were able to generate the file in a string format and pipe it into a text file named strings_final.txt.

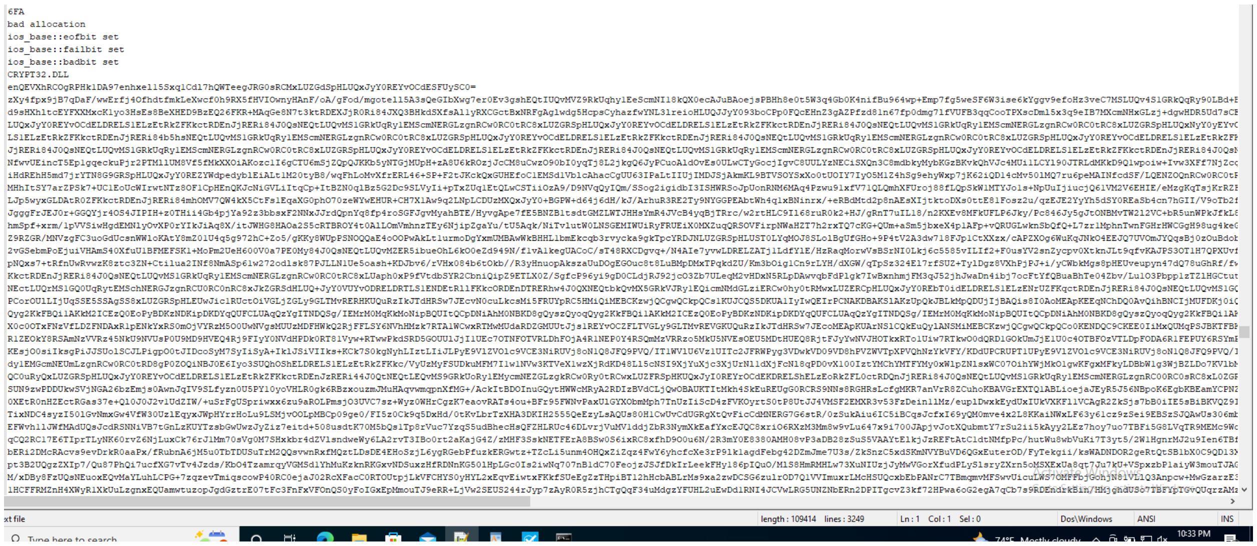
Through reviewing this text file, we initially thought the malware showed signs of packing. Packed malware will often be formatted in an ordered, tight column and is obvious to someone viewing the file as strings. The image in Fig. 3 depicts this packed formation.



```
t$$W
t$=
#L$$#
D$(
l$(
|$(
\$(3
t$=
#|$$
_^][
PPj
PPj
@@f
+D$
WWWWWW
<xt
<Xt
<xt
<Xu
r'u
u,9u
[_^
t$
t$
8SW
SSSSS
Yu%
<xt
<Xt
<xt
<Xu
,a<
rQw
r3w
uA3
u>9u
r9w
t$
t$
SWu
95|
t6S
<8=u
WSP
```

Figure 3. Depiction of packed malware, given the vertical, column-like shape

This initial view of the file also contains strings that are not considered readable, as they do not seem to have any intuitive meaning. This lack of readability points to obfuscation of the code, making it harder for us to analyze the code and find any significant patterns of usage or behavior. This pattern of obfuscation and packing continued for several hundred lines until the strings expanded and became more readable.



```
6FA
bad allocation
ios_base::eofbit set
ios_base::failbit set
ios_base::badbit set
CRYPT32.DLL
enQEVXhRCOgRPHk1DA97enhxel15Sxq1Cdl7hQWTeegJRG0sRCMxLUZGdSpHLUQxJyY0REYvOCdESFUySC0=
```

[illegible]

```
length : 109414   lines : 3249   Ln : 1   Col : 1   Sel : 0   Dos\Windows   ANSI   INS
```

Figure 4. Depiction of the packing being unpacked, expand into unpacked text

Fig. 4 displays this packing breakdown, as the code expands and begins to include readable phrases such as function calls and imports as shown in Fig. 5. The enhanced readability of the strings give us our first chance of gaining any knowledge of how the malware will operate once it is executed.

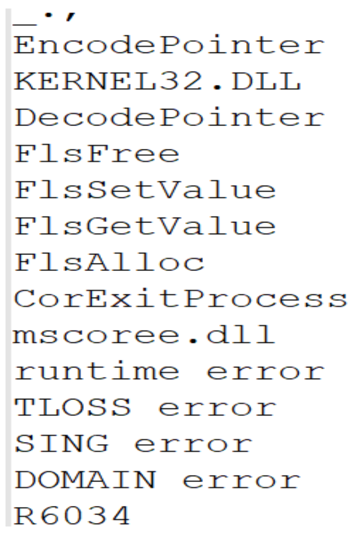
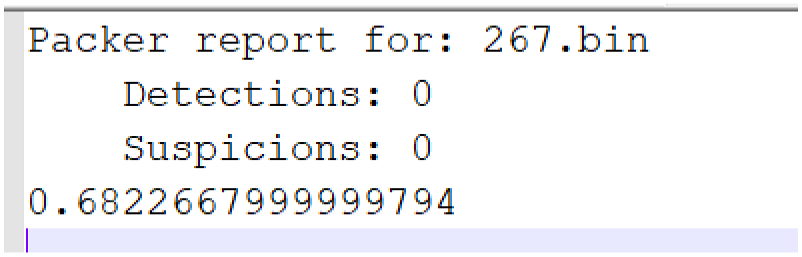

```
_.,
EncodePointer
KERNEL32.DLL
DecodePointer
FlsFree
FlsSetValue
FlsGetValue
FlsAlloc
CorExitProcess
mscoree.dll
runtime error
TLOSS error
SING error
DOMAIN error
R6034
```

Figure 5. Readable strings from strings_final.txt

Fig. 5 gives us our first knowledge of a potential function within the malware, as it is importing the KERNEL32.dll dynamic link library (DLL) and displaying some of the function calls that could be in the original source code such as the FlsSetValue and FlsAlloc. These functions indicate the sample is getting access to high-privileged, low-level systems within the target computer, while also allocating its own memory space. Both of these functions are a few of several common actions that malicious programs often take advantage of. As we continued to breakdown the strings_final.txt file, we found multiple functions being utilized that indicate a malicious presence. This sample displayed the ability to destroy processes, along with allocating and destroying memory space. This initial investigation of the code in string format demonstrated a strong indication of malicious intent.

After processing our file of strings, we moved to studying the imports and packing of the sample file. Firstly, using the packer detection directory referenced in Fig. 2, we can use two tools to analyze the sample. Namely, 'peid.exe' and 'pypackerdetect' serve to detect any packing within the sample, and which encryption algorithms the source code writer used. We used the two commands "peid.exe 267.bin -a -b -v –verbose > peid_final.txt" and "pypackerdetect

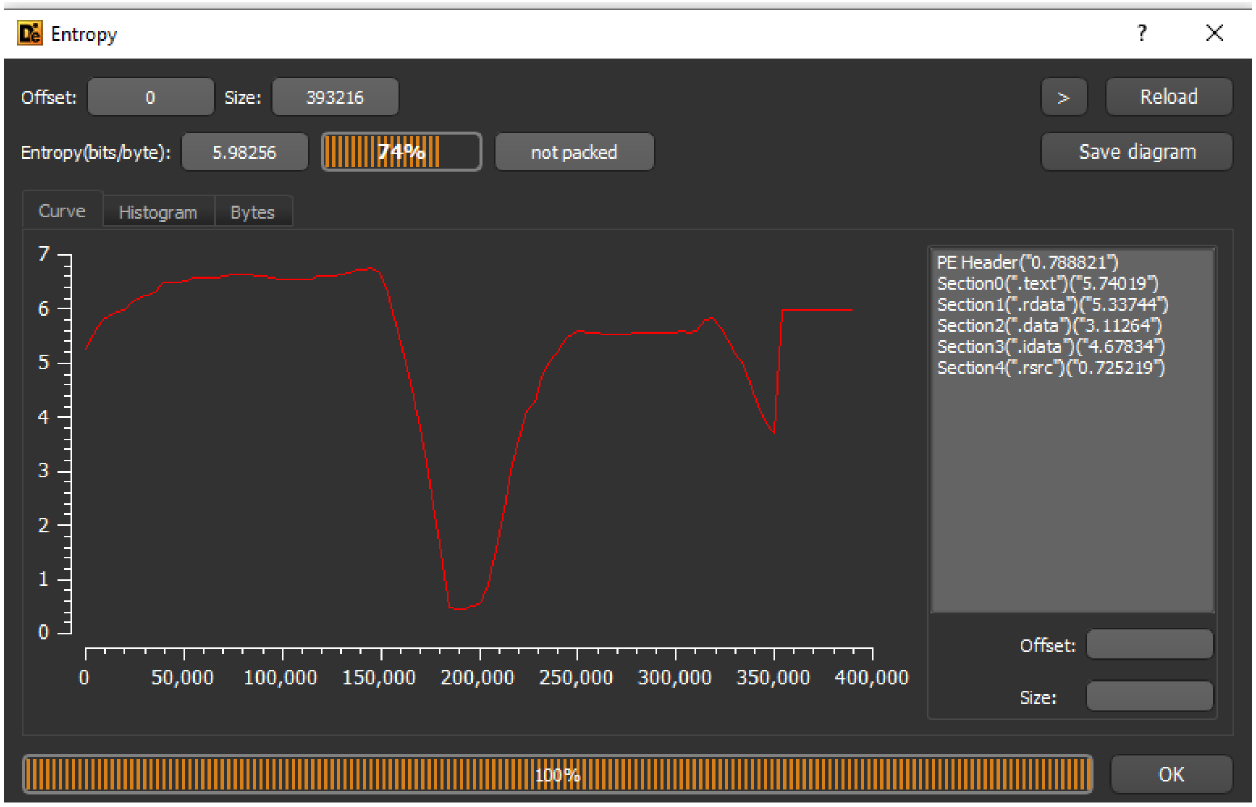
267.bin -b -v –verbose > pypacker_final.txt” to pipe our packing detection results into the corresponding files.



```
Packer report for: 267.bin
    Detections: 0
    Suspicions: 0
0.6822667999999794
```

Figure 6. Evidence of a lack of packing from the pypackerdetect tool

As evidenced in Fig. 6 and the expansion of our strings shown in Fig. 4, our sample does not contain any recorded packing that can be discovered by either our peid.exe or pypackerdetect tools. We observed strong indicators of malicious intent within our strings file through observation of common functions and modules being imported, yet the lack of packing is often a sign of benignware as packing is so popular within malware. To continue analyzing the level of packing, we used the ‘die.exe’ tool to measure the entropy within the sample. Packed code often leads to high levels of obfuscation and a high measure of entropy as it is difficult to recognize patterns within an obfuscated binary file. A higher difficulty of recognizing patterns means a higher entropy, so we use entropy as a predictor for packing. Using our die.exe, we can generate an entropy graph to represent the levels of packing throughout the program. This entropy graph is shown below in Fig. 7.



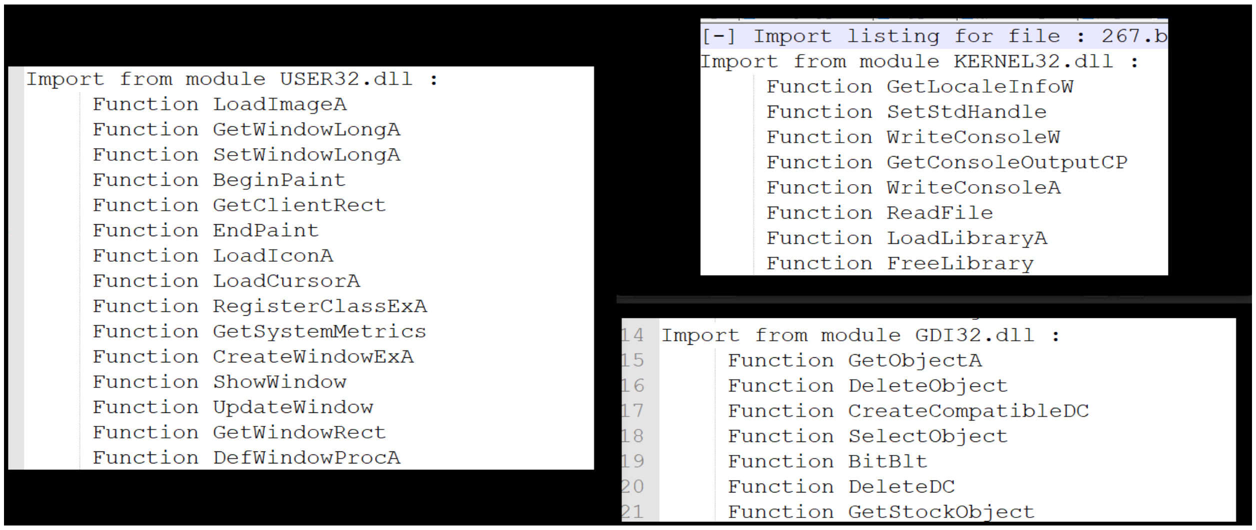
PE Header("0.788821")
Section0(".text")("5.74019")
Section1(".rdata")("5.33744")
Section2(".data")("3.11264")
Section3(".idata")("4.67834")
Section4(".rsrc")("0.725219")

Figure 7. Graph of entropy within 267.bin

Fig. 7 illustrates the changing levels of entropy as we progress through the sample code. Around 85% of malware contains packing, and those samples often have a measure of entropy around 7.5 [7]. This sample was recorded at an entropy level of 5.9, and categorized as not packed, giving further verification of the results shown in Fig. 6. The entropy graph can also be tracked with the results from our strings command, as the first and last hundreds of lines in the strings_final.txt file indicated high levels of packing while the middle strings became much more readable and expanded past the packed vertical format. This explains the high entropy at the beginning and ends of the graph, with much lower entropy in the middle where the strings became more readable. Therefore, we have seen a lack of packing, yet the majority of malware contains packing to avoid detection and reduce pattern recognition by anti-malware software.

This conflict between indicators means our specific categorization of the sample remains unclear as we continue through our static analysis.

Our next step in the static analysis of our sample was using the dependencies directory to find the imports and exports the sample uses. Studying the imports and exports a program uses can allow us to predict the nature of its behavior once we execute it. Using the command “dependencies.exe -imports 267.bin > imports_final.txt”, we generated every module and function imported by the program and piped the results to our imports_final.txt file displayed in Fig. 8.



```
Import from module USER32.dll :
        Function LoadImageA
        Function GetWindowLongA
        Function SetWindowLongA
        Function BeginPaint
        Function GetClientRect
        Function EndPaint
        Function LoadIconA
        Function LoadCursorA
        Function RegisterClassExA
        Function GetSystemMetrics
        Function CreateWindowExA
        Function ShowWindow
        Function UpdateWindow
        Function GetWindowRect
        Function DefWindowProcA
```

```
[-] Import listing for file : 267.b
Import from module KERNEL32.dll :
        Function GetLocaleInfoW
        Function SetStdHandle
        Function WriteConsoleW
        Function GetConsoleOutputCP
        Function WriteConsoleA
        Function ReadFile
        Function LoadLibraryA
        Function FreeLibrary
```

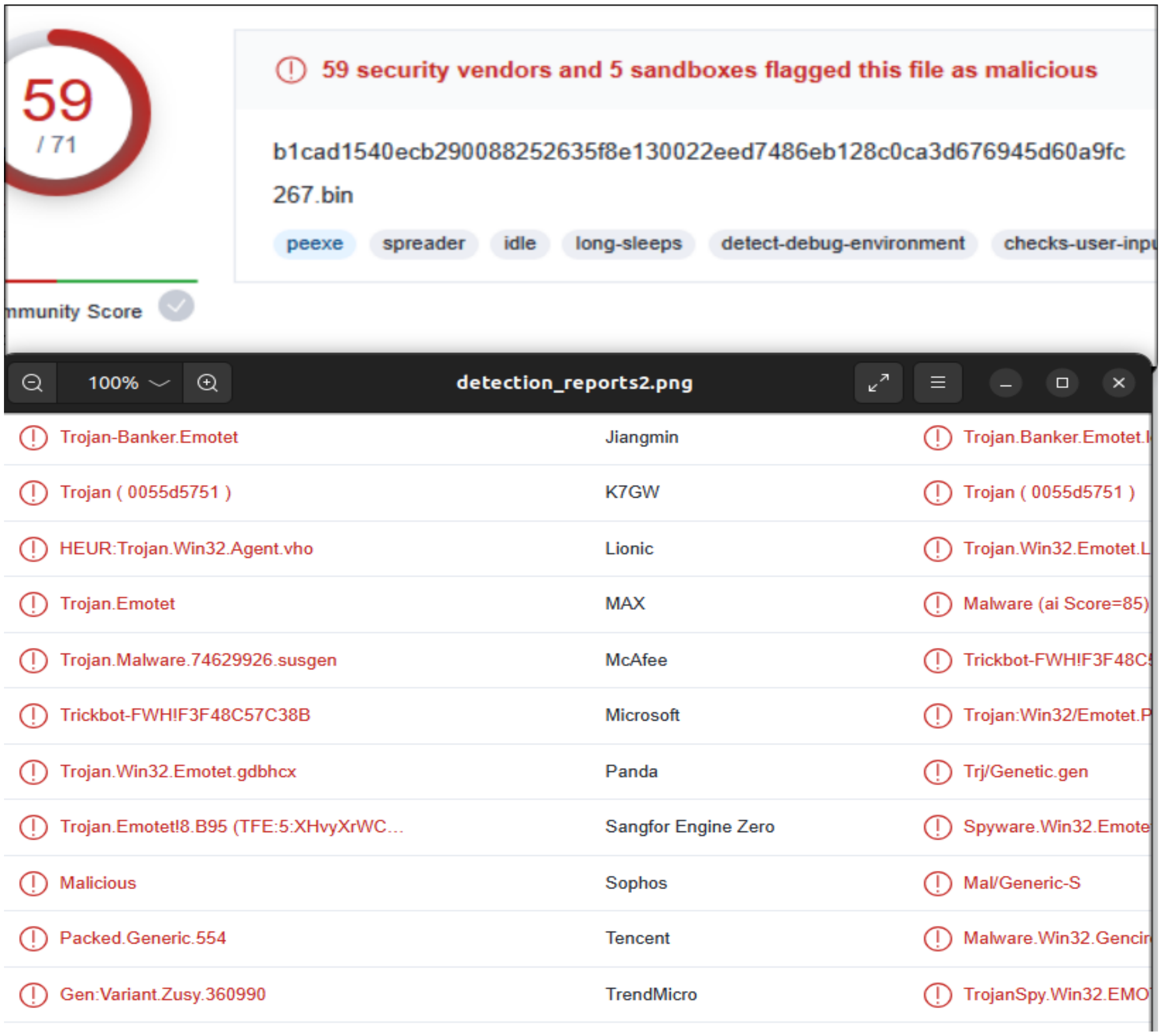
```
Import from module GDI32.dll :
        Function GetObjectA
        Function DeleteObject
        Function CreateCompatibleDC
        Function SelectObject
        Function BitBlt
        Function DeleteDC
        Function GetStockObject
```

Figure 8. Display of the three main imported modules, along with imported functions

Fig. 8 displays the three imported modules that the sample uses to provide enhanced functionality. Namely, USER32.dll, GDI32.dll and KERNEL32.dll, along with several corresponding functions, allow the sample to access dozens of low level functions needed to manipulate a system. This limited number of imported modules with excessive function use is often a sign of malware, along with the type of functions that are being imported. Functions that create and destroy objects, write and read files, and influence memory space all indicate potential malicious intent. Analyzing these imports, along with the functions recorded in our strings file, give us insight that the sample demonstrates the usage of several functions commonly seen with malicious samples.

After our initial analysis of the sample’s strings, dependencies, level of packing and entropy, we now move on to using our PEstudio software to build a final, comprehensive understanding of our sample before its execution. We decided to save PEstudio, along with the VirusTotal database, for our final step of static analysis so we could build an understanding of the sample with minimal assumptions about its classification. Starting with the simplest analysis and building our way up allowed us to take every detail into account before we see a general classification from PEstudio and VirusTotal. After inputting our sample file into PEstudio, we were able to view data on the sample’s strings, imports, exports, dependencies, general indicators and individual classifications from various anti-malware software from the VirusTotal database.

Lastly, we can use the VirusTotal database to search for our sample using the MD5 hash from PEstudio. If the database has a record of this hash we can determine the sample’s recorded family type, behavior, dependencies, encryption, and classification. The results of this database search are shown below in Fig. 9.


59
/ 71

59 security vendors and 5 sandboxes flagged this file as malicious

b1cad1540ecb290088252635f8e130022eed7486eb128c0ca3d676945d60a9fc
267.bin

peexe spreader idle long-sleeps detect-debug-environment checks-user-inpu

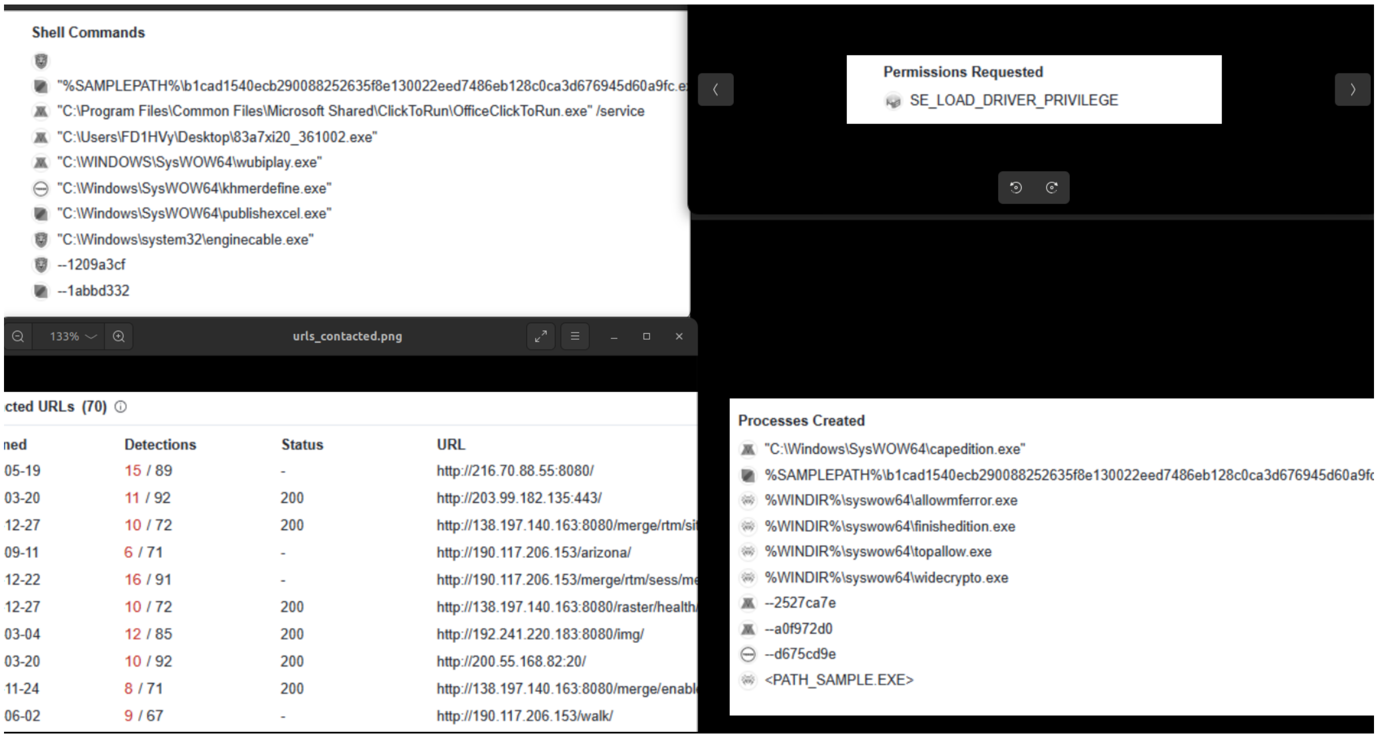
mmunity Score

Trojan-Banker.Emotet	Jiangmin	Trojan.Banker.Emotet.l
Trojan (0055d5751)	K7GW	Trojan (0055d5751)
HEUR:Trojan.Win32.Agent.vho	Lionic	Trojan.Win32.Emotet.L
Trojan.Emotet	MAX	Malware (ai Score=85)
Trojan.Malware.74629926.susgen	McAfee	Trickbot-FWH!F3F48C
Trickbot-FWH!F3F48C57C38B	Microsoft	Trojan:Win32/Emotet.P
Trojan.Win32.Emotet.gdbhcx	Panda	Trj/Genetic.gen
Trojan.Emotet!8.B95 (TFE:5:XHvyXrWC…	Sangfor Engine Zero	Spyware.Win32.Emote
Malicious	Sophos	Mal/Generic-S
Packed.Generic.554	Tencent	Malware.Win32.Gencir
Gen:Variant.Zusy.360990	TrendMicro	TrojanSpy.Win32.EMO

Figure 9. Initial classification of sample 267.bin from VirusTotal

As displayed in Fig. 9, the VirusTotal database contains a record of our sample and has classified it as malicious, with 59/71 malware detectors within the database flagging the program as malicious. This database gives further information into our malware sample, recorded in Fig. 10. Using this information gives us a strong understanding of our malware without even running it. VirusTotal also provides us with a confident prediction in the family and classification of this malware sample. As referenced in Fig. 9, the majority of anti-virus detectors used in the database categorize our sample as a Trojan-Emotet. The Emotet family is often initiated through phishing attacks in emails, and is regarded as a prolific, worm-like malware due to its ability to spread across a network. The malware is able to brute-force user credentials and spread across a network, altering shared drives, registers, and file data while creating new connections within the target network.

Displayed in Fig. 10, we now know it is a malicious sample with the ability to spawn new processes, affect memory space, create files, issue shell commands, create internet connections, uses encryption for obfuscation, can detect if it is being run in a VM and has the ability to use evasive loops to hide from dynamic analysis.



Shell Commands

```
"%SAMPLEPATH%\b1cad1540ecb290088252635f8e130022eed7486eb128c0ca3d676945d60a9fc.e
"C:\Program Files\Common Files\Microsoft Shared\ClickToRun\OfficeClickToRun.exe" /service
"C:\Users\FD1HVy\Desktop\83a7xi20_361002.exe"
"C:\WINDOWS\SysWOW64\wubiplay.exe"
"C:\Windows\SysWOW64\khmerdefine.exe"
"C:\Windows\SysWOW64\publishexcel.exe"
"C:\Windows\system32\enginecable.exe"
--1209a3cf
--1abbd332
```

Permissions Requested

```
SE_LOAD_DRIVER_PRIVILEGE
```

urls_contacted.png

cted URLs (70)

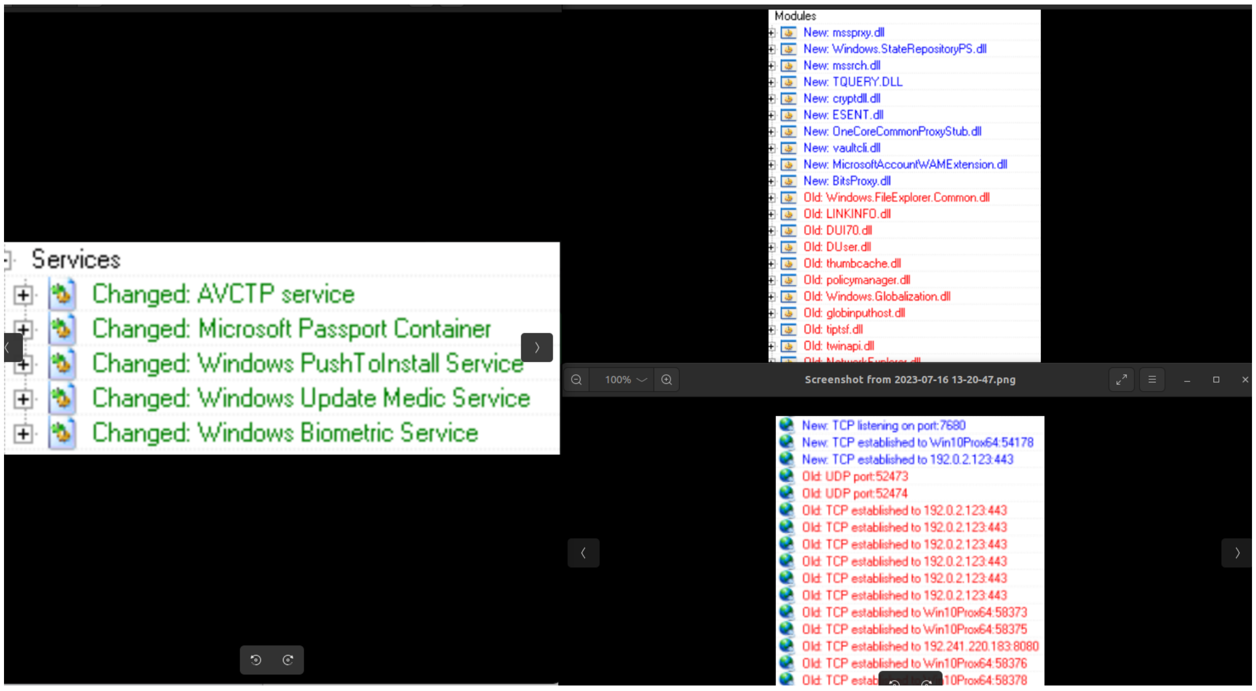
ned	Detections	Status	URL
05-19	15 / 89	-	http://216.70.88.55:8080/
03-20	11 / 92	200	http://203.99.182.135:443/
12-27	10 / 72	200	http://138.197.140.163:8080/merge/rtm/si
09-11	6 / 71	-	http://190.117.206.153/arizona/
12-22	16 / 91	-	http://190.117.206.153/merge/rtm/sess/m
12-27	10 / 72	200	http://138.197.140.163:8080/raster/health/
03-04	12 / 85	200	http://192.241.220.183:8080/img/
03-20	10 / 92	200	http://200.55.168.82:20/
11-24	8 / 71	200	http://138.197.140.163:8080/merge/enabl
06-02	9 / 67	-	http://190.117.206.153/walk/

Processes Created

```
"C:\Windows\SysWOW64\capedition.exe"
%SAMPLEPATH%\b1cad1540ecb290088252635f8e130022eed7486eb128c0ca3d676945d60a9f
%WINDIR%\syswow64\allowmferror.exe
%WINDIR%\syswow64\finishedition.exe
%WINDIR%\syswow64\topallow.exe
%WINDIR%\syswow64\widecrypto.exe
--2527ca7e
--a0f972d0
--d675cd9e
<PATH_SAMPLE.EXE>
```

Figure 10. Recorded behaviors from VirusTotal of sample 267.bin

Given our comprehensive understanding of our malware sample before execution, we now move into dynamic analysis of the file. This requires running the program, and taking snapshots of our VMs before and after execution to gain insight on what changes to the system the malware may have caused. We begin with using our "What's Running" software to query every running process before and after execution, allowing us to compare what new processes were spawned by the malware. We are able to break down the changing processes in categories by IP connections, new services, and new modules being created and used. For these figures, the "What's Running" software deems 'new' (blue) data to be pre-execution and 'old' (red) data to be post-execution.


```
Services
Changed: AVCTP service
Changed: Microsoft Passport Container
Changed: Windows PushToInstall Service
Changed: Windows Update Medic Service
Changed: Windows Biometric Service
```

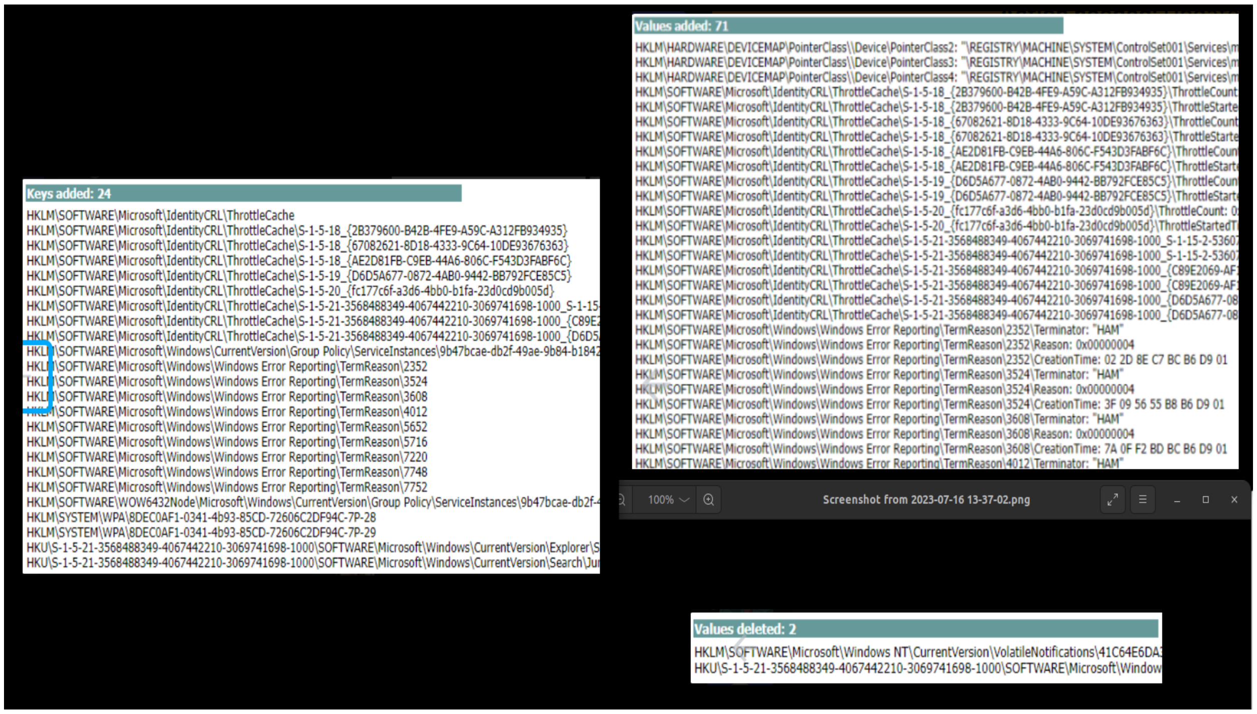
```
Modules
New: mssprxy.dll
New: Windows.StateRepositoryPS.dll
New: mssrch.dll
New: TQUERY.DLL
New: cryptdll.dll
New: ESENT.dll
New: OneCoreCommonProxyStub.dll
New: vaultcli.dll
New: MicrosoftAccountWAMExtension.dll
New: BitsProxy.dll
Old: Windows.FileExplorer.Common.dll
Old: LINKINFO.dll
Old: DUI70.dll
Old: DUser.dll
Old: thumbcache.dll
Old: policymanager.dll
Old: Windows.Globalization.dll
Old: globinputhost.dll
Old: tiptsf.dll
Old: twinapi.dll
```

```
New: TCP listening on port:7680
New: TCP established to Win10Prox64:54178
New: TCP established to 192.0.2.123:443
Old: UDP port:52473
Old: UDP port:52474
Old: TCP established to 192.0.2.123:443
Old: TCP established to 192.0.2.123:443
Old: TCP established to 192.0.2.123:443
Old: TCP established to 192.0.2.123:443
Old: TCP established to 192.0.2.123:443
Old: TCP established to 192.0.2.123:443
Old: TCP established to Win10Prox64:58373
Old: TCP established to Win10Prox64:58375
Old: TCP established to 192.241.220.183:8080
Old: TCP established to Win10Prox64:58376
Old: TCP estab 10Prox64:58378
```

Figure 11. Comparison from pre-post execution, "What's Running"

Fig. 11 displays the services (left), modules (top right), and IP connections (bottom right) the malware spawned after its execution. The malware has created several services while also changing the services from being 'stand alone' to 'shared', representing a desire to expose the security of these services within the system. The red modules and TCP and UDP connections depict the new modules created by our sample, along with the internet connections made by the malware. These created services, connections and modules represent the malicious intent of exposing the target system and creating widespread connections throughout the network for greater infection.

After analyzing the malware's spawned and modified process, we now use the RegShot software to determine the malware's effects on registry values throughout the system. These registry values contain sensitive data about different modules and functions within a computer. Fig. 11 displays the malware's capability to exploit these registry values to compromise the system.



```
Keys added: 24
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-18_{2B379600-B42B-4FE9-A59C-A312FB934935}
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-18_{67082621-8D18-4333-9C64-10DE93676363}
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-18_{AE2D81FB-C9EB-44A6-806C-F543D3FABF6C}
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-19_{D6D5A677-0872-4AB0-9442-BB792FCE85C5}
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-20_{fc177c6f-a3d6-4bb0-b1fa-23d0cd9b005d}
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-21-3568488349-4067442210-3069741698-1000_S-1-15
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-21-3568488349-4067442210-3069741698-1000_{C89E
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-21-3568488349-4067442210-3069741698-1000_{D6D5
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Group Policy\ServiceInstances\9b47bcae-db2f-49ae-9b84-b1842
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\2352
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\3524
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\3608
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\4012
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\5652
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\5716
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\7220
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\7748
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\7752
HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Group Policy\ServiceInstances\9b47bcae-db2f-
HKLM\SYSTEM\WPA\8DEC0AF1-0341-4b93-85CD-72606C2DF94C-7P-28
HKLM\SYSTEM\WPA\8DEC0AF1-0341-4b93-85CD-72606C2DF94C-7P-29
HKU\S-1-5-21-3568488349-4067442210-3069741698-1000\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\S
HKU\S-1-5-21-3568488349-4067442210-3069741698-1000\SOFTWARE\Microsoft\Windows\CurrentVersion\Search\Ju
```

```
Values added: 71
HKLM\HARDWARE\DEVICEMAP\PointerClass\\Device\PointerClass2: "\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\m
HKLM\HARDWARE\DEVICEMAP\PointerClass\\Device\PointerClass3: "\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\m
HKLM\HARDWARE\DEVICEMAP\PointerClass\\Device\PointerClass4: "\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\m
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-18_{2B379600-B42B-4FE9-A59C-A312FB934935}\ThrottleCount
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-18_{2B379600-B42B-4FE9-A59C-A312FB934935}\ThrottleStarte
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-18_{67082621-8D18-4333-9C64-10DE93676363}\ThrottleCount
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-18_{67082621-8D18-4333-9C64-10DE93676363}\ThrottleStarte
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-18_{AE2D81FB-C9EB-44A6-806C-F543D3FABF6C}\ThrottleCoun
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-18_{AE2D81FB-C9EB-44A6-806C-F543D3FABF6C}\ThrottleStart
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-19_{D6D5A677-0872-4AB0-9442-BB792FCE85C5}\ThrottleCoun
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-19_{D6D5A677-0872-4AB0-9442-BB792FCE85C5}\ThrottleStart
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-20_{fc177c6f-a3d6-4bb0-b1fa-23d0cd9b005d}\ThrottleCount: 0
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-20_{fc177c6f-a3d6-4bb0-b1fa-23d0cd9b005d}\ThrottleStartedT
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-21-3568488349-4067442210-3069741698-1000_S-1-15-2-5360
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-21-3568488349-4067442210-3069741698-1000_S-1-15-2-5360
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-21-3568488349-4067442210-3069741698-1000_{C89E2069-AF
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-21-3568488349-4067442210-3069741698-1000_{C89E2069-AF
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-21-3568488349-4067442210-3069741698-1000_{D6D5A677-08
HKLM\SOFTWARE\Microsoft\IdentityCRL\ThrottleCache\S-1-5-21-3568488349-4067442210-3069741698-1000_{D6D5A677-08
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\2352\Terminator: "HAM"
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\2352\Reason: 0x00000004
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\2352\CreationTime: 02 2D 8E C7 BC B6 D9 01
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\3524\Terminator: "HAM"
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\3524\Reason: 0x00000004
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\3524\CreationTime: 3F 09 56 55 B8 B6 D9 01
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\3608\Terminator: "HAM"
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\3608\Reason: 0x00000004
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\3608\CreationTime: 7A 0F F2 BD BC B6 D9 01
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\4012\Terminator: "HAM"
```

```
Values deleted: 2
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\VolatileNotifications\41C64E6DA
HKU\S-1-5-21-3568488349-4067442210-3069741698-1000\SOFTWARE\Microsoft\Window
```

Figure 11. RegShot capture of sample 267.bin, displaying registry modification

This figure illustrates the malicious effects of the sample in a limited time frame, as it demonstrates the ability to add registry keys, values and delete values to increase its effectiveness. These operations demonstrate significant malicious intent, as compromising registry values at this volume highlights the abnormal behavior commonly seen in malicious files.

We established the malware now has the ability to create and modify registry values, services, modules and internet connections. After using the RegShot and "What's Running" tools, we then directed our analysis to the ProcDot tool to build a visual representation of the malware's effects. Fig. 12 further implies this ability to alter registry values, along with changing file names and locations. The malware was able to change its name from 267.bin to 'classtangent.exe', and moved it from Desktop to a new location within the AppData directory. This advanced level of system control indicates the high sophistication levels of this malware and the difficulty in detecting and remediating the corruption on the target system.

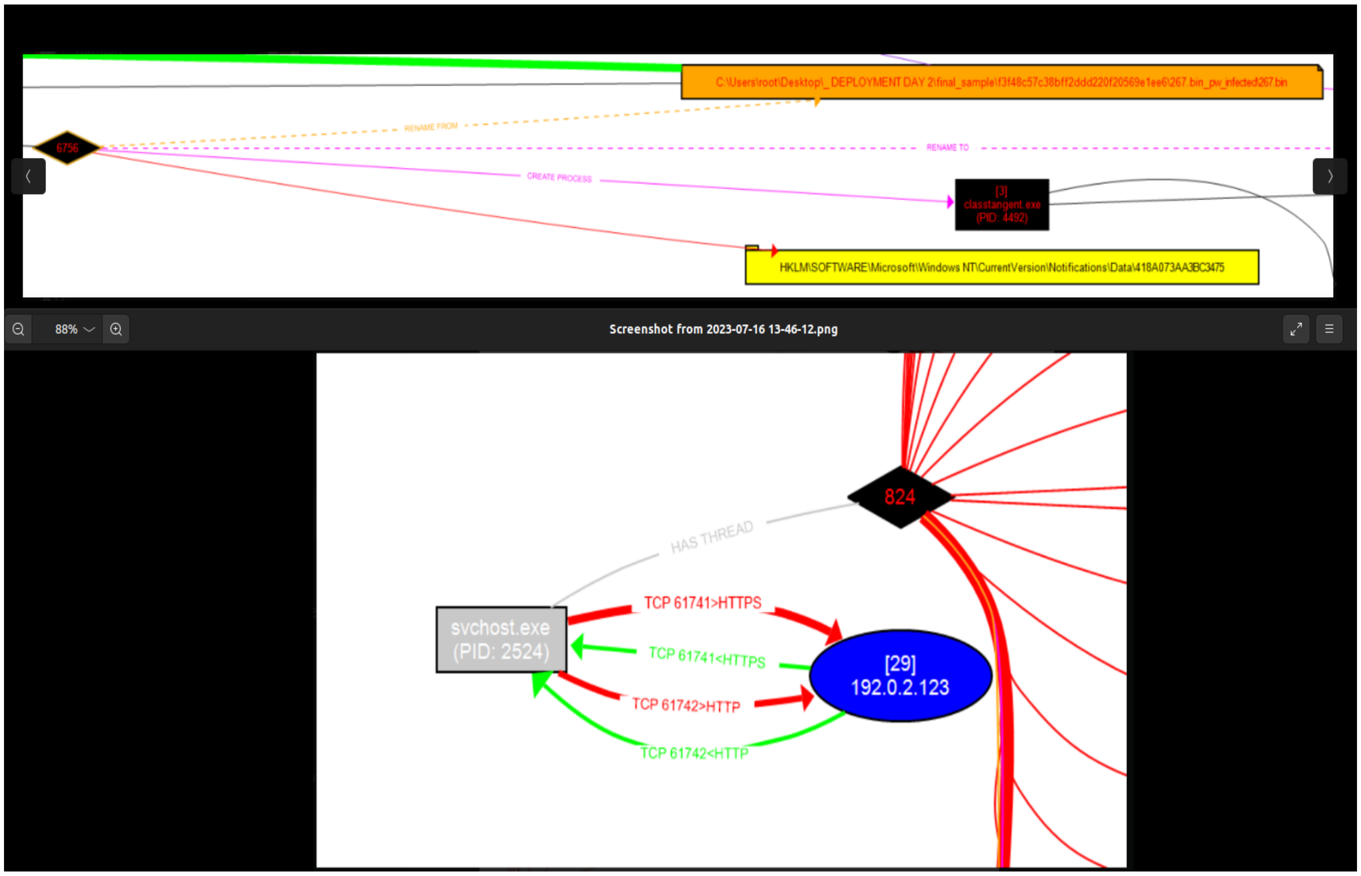

C:\Users\root\Desktop_DEPLOYMENT DAY 2\final_sample\f3f48c57c38bff2ddd220f20569e1ee6\267.bin_pw_infected267.bin
RENAME FROM
6756
RENAME TO
CREATE PROCESS
[3]
classtangent.exe
(PID: 4492)
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Notifications\Data\418A073AA3BC3475
88%
Screenshot from 2023-07-16 13-46-12.png
824
HAS THREAD
svchost.exe
(PID: 2524)
TCP 61741>HTTPS
TCP 61741<HTTPS
TCP 61742>HTTP
TCP 61742<HTTP
[29]
192.0.2.123

Figure 12. Visual representation of directory modification and C2 connection

Fig. 12 also depicts a TCP connection with the IP address 192.0.2.123, although it is being disguised as an http connection. This network activity is indicative of interactions with command and control (C2) frameworks, used to gain external access to a target system and exploit it for information, system access and persistence. This level of internet connectivity further points to malicious behavior. Our process of dynamic analysis established that the malware has the ability to create and modify files, registry values, services and IP connections, along with using common malware stealth techniques of moving to new locations in the system to avoid detection. This dynamic analysis, along with the data collected from static analysis, indicate the sample 267.bin is a Trojan-Emotet malware with the malicious capabilities as mentioned.

4. Results

Through static and dynamic analysis, we identified and classified the sample as Trojan-Emotet malware. We discovered several capabilities of our sample malware, including internet connectivity, spawning its own processes, querying low-level processes, file modification, and debugger defenses. Static analysis was able to check the strings, dependencies, entropy and packing of our sample to get an initial understanding of the sample. This also allowed us to gain insight into potential behaviors of the malware, and to classify the sample as malicious. Furthermore, static analysis highlighted possible attack paths of the malware, including where the malware would attack and what processes it would target. This information proved valuable during dynamic analysis as we were able to prepare for the malware prior to execution. Specifically, the VirusTotal database allowed us to search for our sample, finding several recordings of the malware. This database helped in classifying the sample as malware within the Trojan-Emotet family, and was the driver in explaining its potential offensive and defensive capabilities.

After our initial analysis of a static sample, we moved onto studying the effects of the sample after its execution. Using various tools and software, namely Process Monitor, Process

Hacker, ProcDot, and Regshot, we were able to observe the precise changes made to the system after we executed the malware. Scanning the system highlighted the malware's ability to create new processes by itself, as shown through our use of Process Hacker. We also used Process Monitor to view the file modifications made by the system and the internet connections it established. We then Regshot to analyze the registers and memory space that the malware sample affected. This memory space examination allowed us to understand the level of system corruption caused by the malware, along with becoming cognisant of what precise memory the malware targets. Lastly, we used our ProcDot software to construct a comprehensive diagram of the malware's infection across our system. This visualization of the malware's infection allowed us to understand the infection timeline, what processes the sample depended on and which systems it focused on most. It also underlined the attack path, allowing us to build a preliminary remediation strategy, as we learned which systems had been affected. Along with our ProcDot diagrams, Fig. 13 displays the total recorded timeline of the malware's infection during our dynamic analysis.

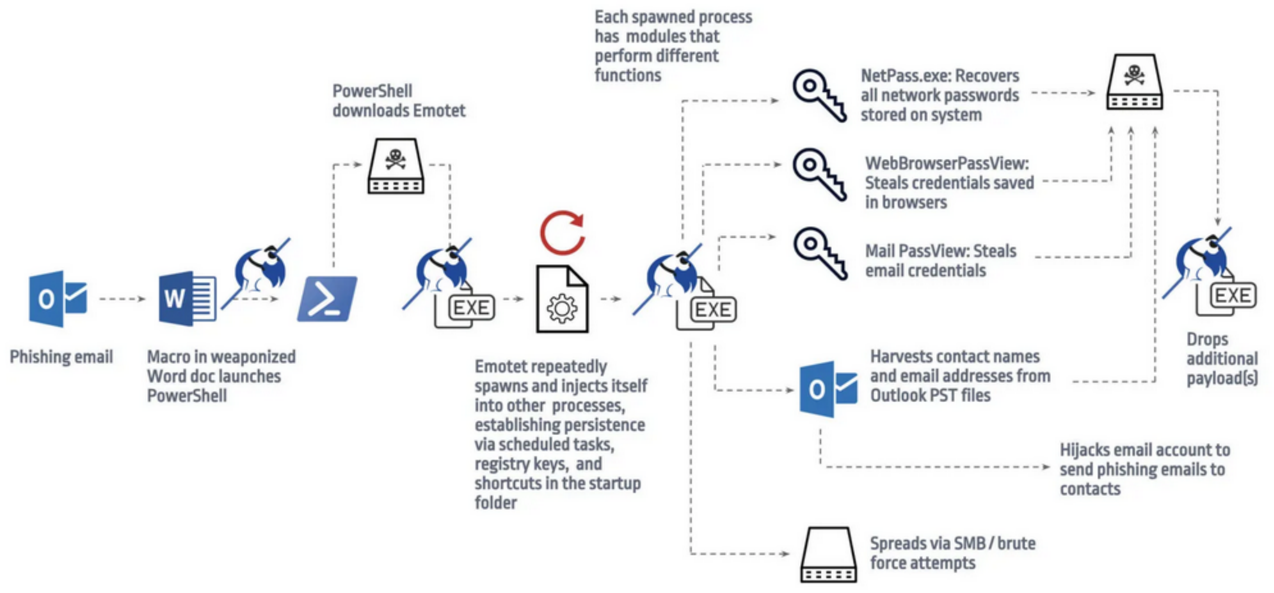

Phishing email
Macro in weaponized Word doc launches PowerShell
PowerShell downloads Emotet
Emotet repeatedly spawns and injects itself into other processes, establishing persistence via scheduled tasks, registry keys, and shortcuts in the startup folder
Each spawned process has modules that perform different functions
NetPass.exe: Recovers all network passwords stored on system
WebBrowserPassView: Steals credentials saved in browsers
Mail PassView: Steals email credentials
Harvests contact names and email addresses from Outlook PST files
Hijacks email account to send phishing emails to contacts
Spreads via SMB / brute force attempts
Drops additional payload(s)
EXE

Figure 13. Visual representation of 267.bin infection timeline

After completion of our static and dynamic analysis, we focused on creating solutions for compromised systems in the future. We divided our remediation into three steps, namely containment, removal, and preparation. The rapid infection of this malware places containment as our top priority. Once the malware is contained and limited in its ability to infect more systems, we can focus on removing any artifacts created by the malware and replacing any lost or modified data. The first step in containment is hindering the malware's ability to create new connections and spread further, so we recommend shutting down internet access to the initial system and any connected system the process infected. We then add the IP addresses the malware sample attempted to access to our firewall, enabling infected systems to block traffic from these dangerous sites in the future.

Our next step in remediation of this malware is its removal from all infected systems, along with restoring any changes the malware made. Our sample malware is able to spawn new processes, change its name, and move locations repeatedly. Using our ProcDot diagram, we can trace the processes created and the current locations of the malware itself. We then kill and delete any traces of the malware to prevent future execution, including its child processes and created files. Also, the Trojan-Emotet family often originates from email and phishing attacks. Removing and bolstering security within these mediums of attack can provide protection against future attacks of a similar nature. We then work to replace all lost or modified data, such as deleted and modified files, along with restoring all registries to their values from the pre-execution state.

After we contain and remove the malware, we focus on preparation and prevention of future attacks from this malware. Using the MD5 and SHA-256 hashes of the original malware file, we can add these hashes to our anti-virus software so it can work to identify and block future infections. Lastly, we recommend learning about different attack methods and vulnerabilities within a system. This can apply to the lap-top user in their homes and large-scale organizations. Training and becoming aware of different attack vectors, such as phishing attacks through email, can help prevent initial infection of a system.

Despite our success in discovering the capabilities of the malware and developing remediation strategies, we were not able to develop a source code approximation from the 267.bin executable file. We believe this is due to the malware's recorded ability to avoid analysis from the debugger and disassemblers. Therefore we do not know the direct origins of the executable sample, but were still able to develop a deep understanding of its capabilities.

5. Discussion

Through reverse engineering and using various static and dynamic analysis tools, we identified a given sample binary file as malicious, and were able to classify it as Trojan-Emotet malware. We were able to study the behaviors of this specific malware and develop a remediation strategy to prevent future infections. Although we were successful in these tasks, our process comes with a few limitations and bounds to consider.

5.1. Bounds and Limitations

Static and dynamic analysis depends on using several software tools to analyze the sample files, as humans often do not have the pattern recognition to understand the samples in a binary format. With this dependency on software, we are limited in the usage abilities of these tools and bounded by their accuracy. Our classification and remediation of this malware sample is linked to the analysis of tools such as ProcDot, VirusTotal and Process Monitor, and if these tools were to give us incorrect data, our solutions to this problem would be rendered ineffective. This dependency on these tools also limits our ability to apply this solution on a larger scale, as not everyone will have access to the several sets of software we use. The necessity of these tools within our process may limit the ability of others to conduct the same analysis, as the tools we used are focused for the Windows OS, and may not be feasible for others to acquire.

Secondly, we are bounded by the environment we analyze the sample in. Due to the malicious nature of the sample file, we cannot execute it and use dynamic analysis on our own systems, so we were forced to use a VM to study the malware in its dynamic state. This limits the scalability of this process, as access to VMs is not guaranteed, yet they are necessary for this problem.

Furthermore, we discovered our sample malware has the ability to detect if it is being dynamically analyzed within a VM. This could limit our analysis in terms of effectiveness, as we are not sure if the malware's ability to detect analysis affected our final findings. We also were unable to construct a source code approximation using the Ghidra software, most likely due to the anti-disassembler defenses the malware is capable of. This limits our ability to discover the origins and creator of the executable sample and the ability in using the source code to evolve our offensive and defensive tactics in the future.

Lastly, the assumption of a limited scope with respect to the available examination time limits our ability to study the malware in an expanded time frame. Modern malware has the ability to delay its execution, so the analyst thinks they are finished with the sample analysis within an hour or two, yet in actuality it will take a few days for the malware to fully execute. Our full dynamic analysis took place within a few hours, so we are limited in knowledge by the possible dormant ability of this malware.

5.2. Impact

The field of reverse engineering has become more important as malicious abilities of our adversaries continue to evolve. The ability to analyze samples for malicious intent is the first step in developing stronger defenses, while also learning how to improve our own attacks. As we have demonstrated, static and dynamic analysis prove to be effective in building an understanding of how a specific malware sample or family operates. Using this knowledge and comprehending the inner workings of a piece of code allows us to continue enhancing the technology around us. Furthermore, reverse engineering within the field of malware is unique in its ability to reconstruct an approximation of the source code that a sample was compiled from. Having this source code can provide us with wider growth and enhancements within the cyber field as we can trace who, where and when the code was developed. This knowledge can aid in the cyber field, but also expand its influence into further domains of warfare.

5.3. Future Work

Reverse engineering will continue to be one of the forefront fields of cyber warfare due to its inherent value in discovering advanced knowledge from minimal information. The future of this field will rely on the engineers ability to enhance the software and tools we use for pattern recognition and breaking the executable files down into readable formats. We believe the future of this field should be geared towards developing stronger debuggers to create more accurate, available predictions of source code, along with enhancing our analysis environments to bypass the malware's ability to detect analysis. This will help overcome the limitations of our analysis environment along with being able to analyze the malware without needing to adjust for delayed execution times or dormant features.

6. Works Cited

[1] “Number of malware attacks per year 2018,” *Statista*.

https://www.statista.com/statistics/873097/malware-attacks-per-year-worldwide/

[2] L. O'brien, 2005. Accessed: Jul. 16, 2023. [Online]. Available:

https://www.cs.cmu.edu/~aldrich/courses/654-sp05/handouts/MSE-RevEng-05.pdf

[3] medium.com, *Conversion of binary to strings*. Accessed: 2023. [Online]. Available:

https://miro.medium.com/v2/resize:fit:4800/format:webp/1*JQ1K5fwdGPsczcTiNbmN6w.png

[4] G. yadav, “Packing and Obfuscation,” *RIXED_LABS*, Mar. 14, 2021.

https://medium.com/ax1al/packing-and-obfuscation-fe6b03bbc267

[5] “What is an entropy graph,” *Reverse Engineering Stack Exchange*.

https://reverseengineering.stackexchange.com/questions/21555/what-is-an-entropy-graph

(accessed Jul. 16, 2023).

[6] R. Lyda and J. Hamrock, “Using Entropy Analysis to Find Encrypted and Packed Malware,”

IEEE Security and Privacy Magazine, vol. 5, no. 2, pp. 40–45, Mar. 2007, doi:

https://doi.org/10.1109/msp.2007.48.

[7] “Emotet Malware | CISA,” *Cybersecurity and Infrastructure Security Agency CISA*.

https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-280a