AUTOMATIC DETECTION OF MALICIOUS SOFTWARE NIIT INFORMATION SECURITY RESEARCH GROUP (ISRG)

By

Neelma Nadeem
(2003-NUST-BIT-41)

A project report submitted in partial fulfillment of
The requirement for the degree of
Bachelors in Information Technology

In

NUST Institute of Information Technology
National University of Sciences and Technology
Rawalpindi, Pakistan
(2007)

CERTIFICATE

It is certified that the contents and form of thesis entitled **“Automatic Detection of Malicious Software”** submitted by Ms. Neelma Nadeem have been found satisfactory for the requirement of the degree.

Project Advisor: __________________
Assistant Professor (Dr. Fauzan Mirza)

Co-Advisor: _______________________________
Lecturer (Mr. Shahrzad Khattak)

Member: _______________________________
Associate Professor (Dr. Hafiz Farooq)

Member: _______________________________
Lecturer (Mr. Awais Shibli)

DEDICATION

I dedicate this humble effort, the fruits of my thoughts and study to my affectionate parents and family members, who inspired me to higher ideas of life.

ACKNOWLEDGEMENTS

I wish to express my sincerest gratitude and indebtedness to my advisor/ supervisor, Professor Dr. Fauzan Mirza, for his timely assistance, guidance and encouragement at every stage of this study. My deepest appreciation and thanks are extended to my co-advisor, Mr. Sharzad Khattak, for his continuous and valuable suggestions, guidance and untiring assistance, especially in the provision of requisite facilities, throughout my thesis work.

I am highly thankful to Dr. Ali Khayam for his encouragement, motivation and help.

I am especially grateful to my friends Ms Rabail Javed and Ms Midhat Batool for their valuable suggestions, encouragements and support throughout the study.

My sincere thanks are due to all of my teachers for their effective teaching and contributions enhancement of my knowledge and ability to undertake this challenging research work. I am especially thankful to Dr. Fauzan Miraza and Sharzad Khattak, who motivated me for the hard work required in conducting of the research work.

I am thankful to many friends who encouraged me to work harder and harder and continued their moral support throughout my course and project work.

With humble regards, I offer my deepest indebtedness to my affectionate parents and all other family members for their encouragement and prayers to Almighty God for my success.

It is almost impossible to make note of all those, whose inspirations have been vital in the completion of this thesis, I am grateful to all of them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

"There has been significant interest in malware since they first appeared in 1981 and especially in the past few years as they have outreached a large number of computer environments. Malware Threat is serious global problems that's causes productivity and time both to be wasted. Various Techniques are being used by the anti-virus companies to detect and remove malware but each technique has its own pros and cons. This report begins with a description of the current techniques used by anti-virus companies to detect malware. And then proposes a new technique by analyzing malware and benign software through reverse code engineering and develops a methodology to detect and classify malware from benign software based on the features that are distinct in malware."

Chapter 1

INTRODUCTION

The threat of malicious software can easily be considered as the greatest threat to Internet security. Earlier, viruses were, more or less, the only form of malware. Nowadays, the threat has grown to include network-aware worms, trojans, DDoS agents, IRC Controlled bots, spyware, and so on. The infection vectors have also changed and grown and malicious agents now use techniques like email harvesting, browser exploits, operating system vulnerabilities, polymorphism, metamorphic viruses, code obfuscation and P2P networks to spread. A relatively large percentage of the software that a normal internet user encounters in his online journeys is or can be malicious in some kind of way. Most of this malware is stopped by antivirus software, spyware removal tools and other similar tools. However, this protection is not always enough and there are times when a small, benign looking binary sneaks through all levels of protection and compromises user data. There may be many reasons for this breach, such as

- a user irregularly updating his AV signatures,
- a failure of AV heuristics,
- the introduction of new or low-profile malware which has not yet been discovered by AV vendors, and
- custom coded malware which cannot be detected by antivirus software.

Though AV software is continually getting better, a small but very significant percentage of malware escapes the automated screening process and manages to enter and wreak havoc on networks. Unfortunately, this percentage is also growing everyday.

Malware is a serious global problem that causes productivity and time both to be wasted in the process of installing anti-virus updates, scanning for malware, removing malware, system backup and data recovery. The major research works on malware are done at AV companies and they are usually not published.

1.1 BACKGROUND

Currently, all commercial anti-virus software is based on string recognition. Each anti-virus product maintains its own database of virus ‘signatures’. Anti-virus software scans programs on a computer system for occurrences of any virus signature in its database. This enables it to positively identify a program as an implementation of a particular virus, which facilitates subsequent removal.

A single method of detecting computer viruses has nearly eclipsed all others: scanning for known viruses [1]. Originally, a string of bytes was selected from some known virus, and the virus scanner looked for that string in files as a way of determining if that file was infected with that virus. Later, more complex techniques were developed which involved looking for various substrings in various parts of the file. But all of these techniques have one thing in common: they look for static characteristics of viruses that are already known.

The virus signature database is maintained by the OEM of the anti-virus and distributed to users through the mechanism of updates. The disadvantages of the existing methodology are:

- The dependence by users on the OEM and anti-virus software to maintain their systems, which usually will be a running expense due to the cost of the subscription to the signature database updates; and
- Anti-virus software can only detect malware that is contained in its database, and users that do not maintain up-to-date signature databases (e.g., those users whose update subscriptions have lapsed) are susceptible to new viruses. Current anti-virus technology relies almost entirely on finding a particular virus

before being able to deal with it well. This is referred to as a reactive technology [1]. Customers are required to update their anti-virus software periodically to deal with new threats. Customers (and anti-virus marketers!) have long desired anti-virus solutions that did not require constant updates. Some anti-virus vendors have gone so far as to claim that their products could detect all possible viruses, never make mistakes, and never need updates, a claim that can be easily shown to be mathematically impossible.

1.1.1 Static Analysis

There are many ways to study a program's behavior. With static analysis, study a program without actually executing it. Tools of the trade are disassemblers, decompilers, source code analyzers, and even such basic utilities as strings. Static analysis has the advantage that it can reveal how a program would behave under unusual conditions, because we can examine parts of a program that normally do not execute. In real life, static analysis gives an approximate picture at best. It is impossible to fully predict the behavior of all but the smallest programs..

1.1.2 Dynamic Analysis

With dynamic analysis, study a program as it executes. Here, tools of the trade are debuggers, function call tracers, registry monitors, file system monitors, and network sniffers. The advantage of dynamic analysis is that it can be fast and accurate. It is not possible to predict the behavior of a non-trivial program and it is also not possible to make a non-trivial program traverse all paths through its code.

1.2 PROBLEM STATEMENT

“Analyze malware and benign software through reverse code engineering and develop a methodology to detect and classify malware from benign software based on the features that are distinct in malware”.

1.3 PROPOSED SOLUTION

The aim of this project is to develop a methodology to classify malware from benign software. There are various approaches that can be taken to in an attempt to achieve this goal. The approach that we chose is based on a manual analysis of various existing malicious software programs to determine a set of features that are common to malware. These will then form the basis for a ‘malware detection algorithm’ that will try to distinguish malicious software from benign software. The features give decision criteria that are then assigned weights (depending on the likelihood of the feature being present in a malicious/benign program) and used to decide if a program is malicious.

This project is concerned not only with computer viruses, but also other common malicious software. For example, worms are currently a greater threat to computer security than viruses, due to the Internet. The implementation of viruses and worms are so extremely different that it would not be trivial to update a typical anti-virus product to detect and eliminate the threat from worms. Rather, the anti-virus product would need to have code added to it to deal specifically with the threat due to worms, in addition to viruses. I stress this because this project is novel in the sense that it attempts to solve the malware problem by looking at malware as a general problem, rather than examining specific types of malware.

1.4 METHODOLOGY

Today, many anti-virus (AV) scanners primarily detect viruses by looking for simple virus signatures1 within the file being scanned. The signature of a virus is typically created by disassembling the virus into assembly code, analyzing it, and then selecting those sections of code that seem to be unique to the virus. The binary bits of those unique sections become the signature for the virus. However, this approach can be easily subverted by polymorphic viruses, which change their code (and virus signature) every time they’re run. In response, AV vendors implemented heuristics and decryption engines that would run the decryptor/loader code of the binary and peak inside the unencrypted binary to determine if it’s a virus. However, the fact is that most viruses are of the “simple” type – not encrypted or polymorphic, and many of them have many variants that come out afterwards.[11]

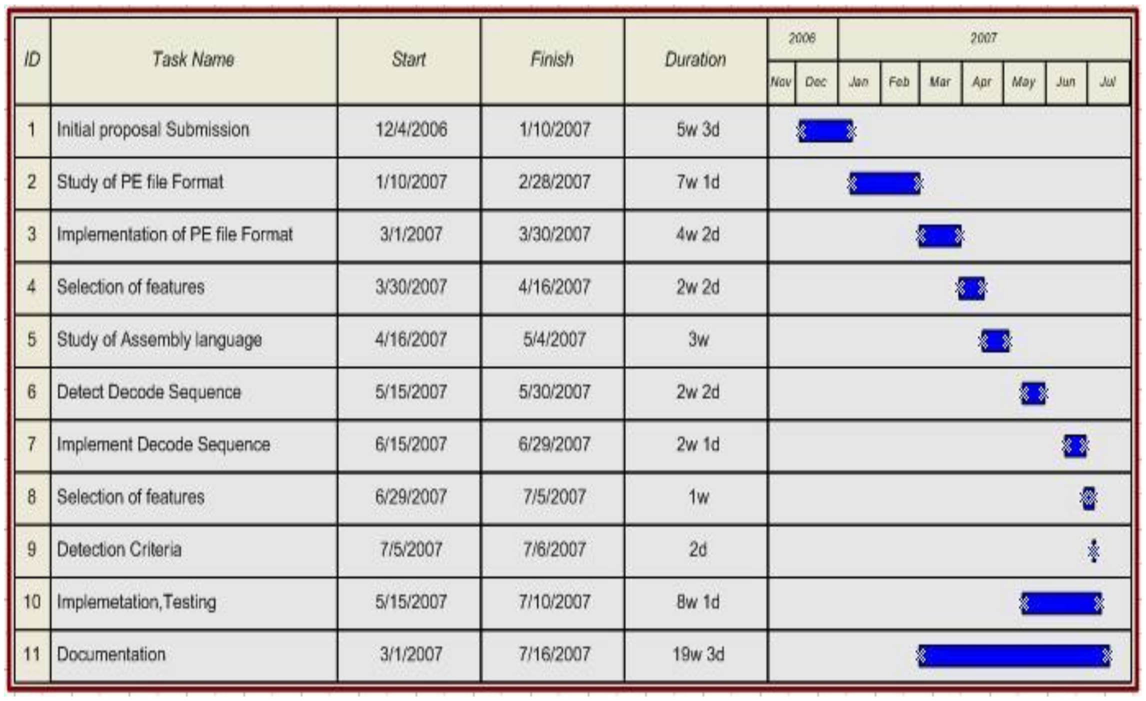
A research at BELL labs believe that reverse code engineering (RCE) can be used to better analyze viruses and provide us with better techniques to protect against them and their variants. Our research aims at detecting malware with the possibility to be further improved by developing methods to prevent and recover from malware threats. RCE can be defined as analyzing and disassembling a software system in order understand its design, components, and inner-workings [11]. RCE also allows us to see hidden behaviors that cannot be directly observed by running the virus or those actions that have yet to be activated. These benefits can be used to prematurely defeat a virus’s future variants by better analyzing the original virus.

We have attempted to document an approach for reverse engineering malicious software. This involves manually investigating various programs and finding common features that distinguishes malicious software from benign software. These features are decision criteria that are then assigned weights (depending on the likelihood of the feature being present in a malicious/benign program) and used to decide if a program is malicious.

We would go for an approach to demonstrate the process of reverse engineering malware using a range of system monitoring tools in conjunction with a disassembler and a debugger.

1.5 TIME LINE

Various tasks to be carried out for completion of the research project, along with their estimated time frame, have been shown in Figure



ID	Task Name	Start	Finish	Duration
1	Initial proposal Submission	12/4/2006	1/10/2007	5w 3d
2	Study of PE file Format	1/10/2007	2/28/2007	7w 1d
3	Implementation of PE file Format	3/1/2007	3/30/2007	4w 2d
4	Selection of features	3/30/2007	4/16/2007	2w 2d
5	Study of Assembly language	4/16/2007	5/4/2007	3w
6	Detect Decode Sequence	5/15/2007	5/30/2007	2w 2d
7	Implement Decode Sequence	6/15/2007	6/29/2007	2w 1d
8	Selection of features	6/29/2007	7/5/2007	1w
9	Detection Criteria	7/5/2007	7/6/2007	2d
10	Implemetation,Testing	5/15/2007	7/10/2007	8w 1d
11	Documentation	3/1/2007	7/16/2007	19w 3d

Figure 1 - Project Timeline

1.6 REPORT STRUCTURE

This document sheds light on all the aspects of the project, including the technique(s) being implemented and details of the developed tool. This document has been divided on the following pattern:

❖ Chapter 2: Literature Review. This chapter puts light on current malware trends and different techniques used by Malware writers.

❖ Chapter 3: Methodology. This chapter explains the approach that has been used, design of Detection system.

❖ Chapter 4: Results This chapter includes the steps followed and the final results obtained

❖ Chapter 5: Conclusions. This chapter summarizes the whole project report

- Chapter 6: Recommendations. In this chapter proposed methodologies have been discussed which could further affect improvements based on search and research carried out during the execution of this project

Chapter 2

LITERATURE REVIEW

The review of literature for the project has been divided in the following Sections

1) Investigation Tools
2) Malicious Software n its taxonomy
3) Purpose of Malware
4) Vulnerability to Malware
5) Related Work
6) Malware trends

2.1 INVESTIGATION TOOLS

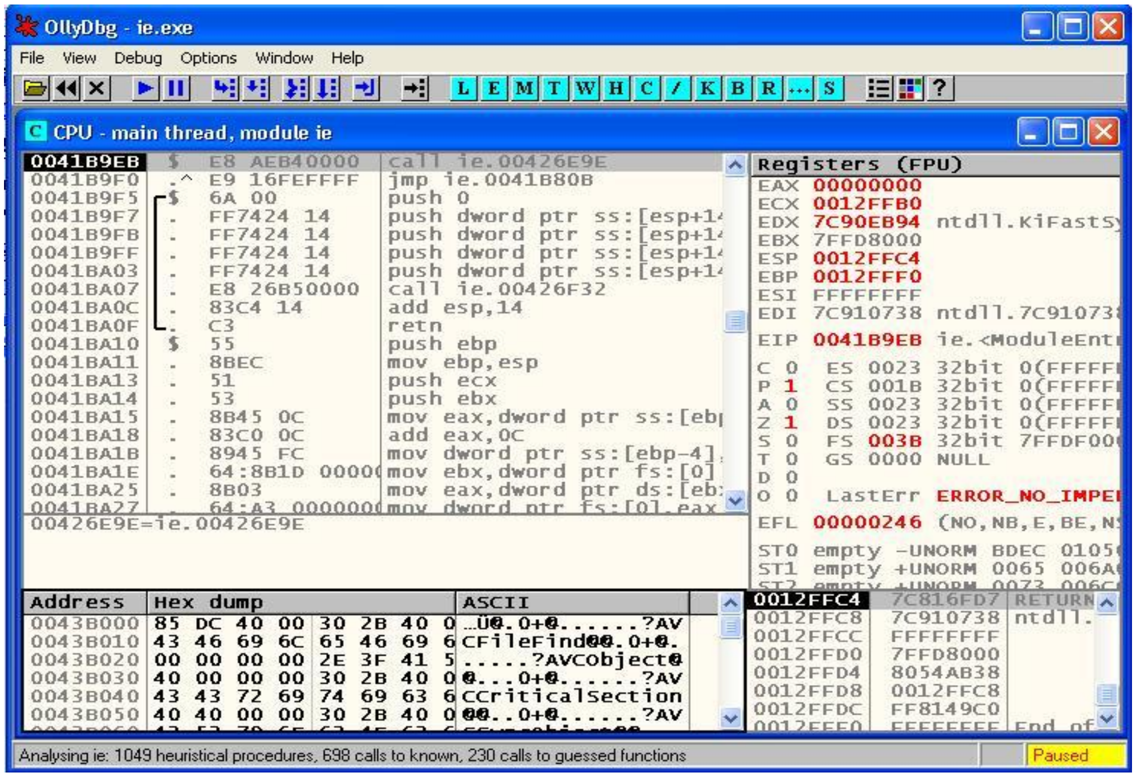
Numbers of tools are used for the investigating Executables. Using these tools one can investigate different properties of the file. To get a detailed and more real picture of an executable, different tools are used to explore a file. Some of the tools that are used for the manual analysis of a file and for reverse engineering purposes are given below:

- ➢ OllyDbg
- ➢ PE Explorer
- ➢ UltraEdit

2.1.1 Ollydbg:-

OllyDbg is a 32-bit assembler level analyzing debugger for Microsoft Windows. Emphasis on **binary code analysis** makes it particularly useful in cases where source is unavailable. OllyDbg is a shareware, but can download and use it **for free**. Special highlights are: [13]

- Code analysis - traces registers, recognizes procedures, loops, API calls, switches, tables, constants and strings
- Directly loads and debugs DLLs
- Saves patches between sessions, writes them back to executable file and updates fixups
- Configurable disassembler, supports both MASM and IDEAL formats
- Dynamically recognizes ASCII and UNICODE strings - also in Delphi format!
- Recognizes complex code constructs, like call to jump to procedure
- Decodes calls to more than 1900 standard API and 400 C functions
- Sets conditional, logging, memory and hardware breakpoints
- Traces program execution, logs arguments of known functions
- Searches whole allocated memory
- Finds references to constant or address range
- Examines and modifies memory, sets breakpoints and pauses program on-the-fly


OllyDbg - ie.exe
File View Debug Options Window Help
L E M T W H C / K B R ... S ?
CPU - main thread, module ie
Paused

```
0041B9EB   $   E8 AEB40000   call ie.00426E9E
0041B9F0   .^  E9 16FEFFFF   jmp ie.0041B80B
0041B9F5  ┌$   6A 00         push 0
0041B9F7  │.   FF7424 14     push dword ptr ss:[esp+14
0041B9FB  │.   FF7424 14     push dword ptr ss:[esp+14
0041B9FF  │.   FF7424 14     push dword ptr ss:[esp+14
0041BA03  │.   FF7424 14     push dword ptr ss:[esp+14
0041BA07  │.   E8 26B50000   call ie.00426F32
0041BA0C  │.   83C4 14       add esp,14
0041BA0F  └.   C3            retn
0041BA10   $   55            push ebp
0041BA11   .   8BEC          mov ebp,esp
0041BA13   .   51            push ecx
0041BA14   .   53            push ebx
0041BA15   .   8B45 0C       mov eax,dword ptr ss:[eb
0041BA18   .   83C0 0C       add eax,0C
0041BA1B   .   8945 FC       mov dword ptr ss:[ebp-4],
0041BA1E   .   64:8B1D 0000( mov ebx,dword ptr fs:[0]
0041BA25   .   8B03          mov eax,dword ptr ds:[eb:
0041BA27   .   64:A3 000000( mov dword ptr fs:[0],eax
00426E9E=ie.00426E9E
```

```
Registers (FPU)
EAX 00000000
ECX 0012FFB0
EDX 7C90EB94 ntdll.KiFastSy
EBX 7FFD8000
ESP 0012FFC4
EBP 0012FFF0
ESI FFFFFFFF
EDI 7C910738 ntdll.7C91073

EIP 0041B9EB ie.<ModuleEnt

C 0  ES 0023 32bit 0(FFFFF
P 1  CS 001B 32bit 0(FFFFF
A 0  SS 0023 32bit 0(FFFFF
Z 1  DS 0023 32bit 0(FFFFF
S 0  FS 003B 32bit 7FFDF00
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_NO_IMPE

EFL 00000246 (NO,NB,E,BE,N

ST0 empty -UNORM BDEC 0105
ST1 empty +UNORM 0065 006A
```

```
Address   Hex dump                  ASCII
0043B000  85 DC 40 00 30 2B 40 0  …Ü@.0+@......?AV
0043B010  43 46 69 6C 65 46 69 6  CFileFind@@.0+@.
0043B020  00 00 00 00 2E 3F 41 5  .....?AVCObject@
0043B030  40 00 00 00 30 2B 40 0  @...0+@......?AV
0043B040  43 43 72 69 74 69 63 6  CCriticalSection
0043B050  40 40 00 00 30 2B 40 0  @@..0+@......?AV
```

```
0012FFC4   7C816FD7  RETURN
0012FFC8   7C910738  ntdll.
0012FFCC   FFFFFFFF
0012FFD0   7FFD8000
0012FFD4   8054AB38
0012FFD8   0012FFC8
0012FFDC   FF8149C0
```

```
Analysing ie: 1049 heuristical procedures, 698 calls to known, 230 calls to guessed functions
```

Figure 2 - Snapshot Of Ollydbg

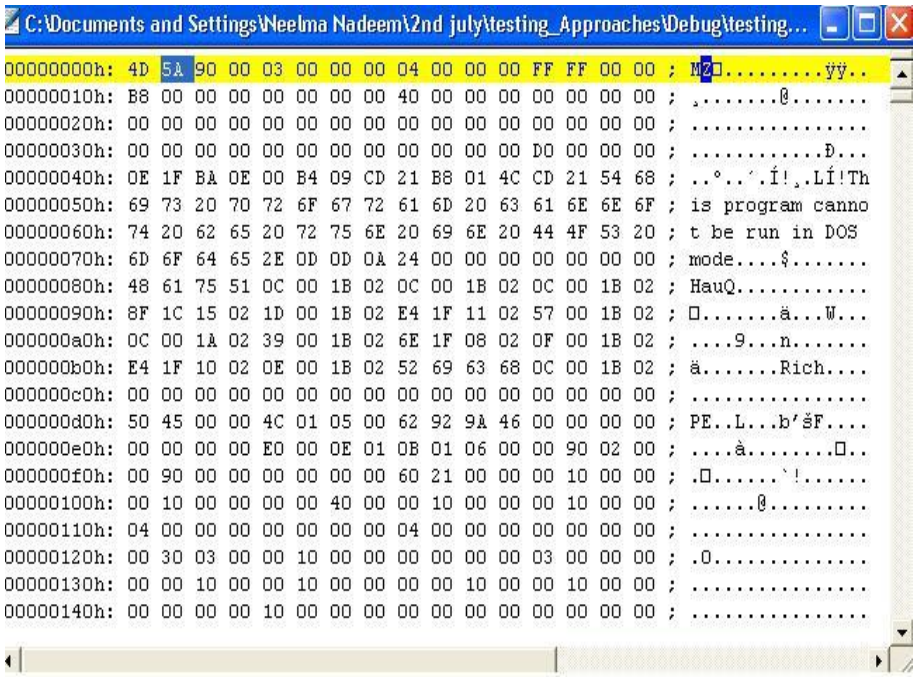
2.1.2 PE Explorer

PE Explorer is designed for inspection and editing of Windows executable files, PE Explorer offers framework for working with EXE, DLL, ActiveX controls, and other executable file formats that run on MS Windows 32-bit platforms.

PE Explorer tells just about every little detail that could possibly want to know about a *PE file.*[14]

2.1.3 Ultraedit

UltraEdit is a commercial text editor for Microsoft Windows created in 1994 by IDM Computer Solutions. It supports Unicode and hex editing modes which is the main requirment of our investigaion. Files can be browsed and edited in tabs. UltraEdit compares well feature-wise with other development editors.[15]



C:\Documents and Settings\Neelma Nadeem\2nd july\testing_Approaches\Debug\testing...

```
00000000h: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 ; MZ□.........ÿÿ..
00000010h: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ; ¸.......@.......
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 D0 00 00 00 ; ............Ð...
00000040h: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ; ..º..´.Í!¸.LÍ!Th
00000050h: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F ; is program canno
00000060h: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 ; t be run in DOS 
00000070h: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 ; mode....$.......
00000080h: 48 61 75 51 0C 00 1B 02 0C 00 1B 02 0C 00 1B 02 ; HauQ............
00000090h: 8F 1C 15 02 1D 00 1B 02 E4 1F 11 02 57 00 1B 02 ; □.......ä...W...
000000a0h: 0C 00 1A 02 39 00 1B 02 6E 1F 08 02 0F 00 1B 02 ; ....9...n.......
000000b0h: E4 1F 10 02 0E 00 1B 02 52 69 63 68 0C 00 1B 02 ; ä.......Rich....
000000c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000d0h: 50 45 00 00 4C 01 05 00 62 92 9A 46 00 00 00 00 ; PE..L...b’šF....
000000e0h: 00 00 00 00 E0 00 0E 01 0B 01 06 00 00 90 02 00 ; ....à........□..
000000f0h: 00 90 00 00 00 00 00 00 60 21 00 00 00 10 00 00 ; .□......`!......
00000100h: 00 10 00 00 00 00 40 00 00 10 00 00 00 10 00 00 ; ......@.........
00000110h: 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 ; ................
00000120h: 00 30 03 00 00 10 00 00 00 00 00 00 03 00 00 00 ; .0..............
00000130h: 00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00 ; ................
00000140h: 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

Figure 3 - Snapshot Of Ultraedit

2.2 MALICIOUS SOFTWARE

Malware or malicious software is software designed to penetrate or damage a computer system without the owner's informed approval. The expression is a general term used by computer professionals to mean a variety of forms of hostile, intrusive, or annoying software or program code. All malicious software affects productivity.

Many normal computer users are however still unfamiliar with the term, and most never use it. Instead, "(computer) virus" is used in common parlance and often in the general media to describe all kinds of malware.

Software is considered malware based on the perceived intent of the creator rather than any particular features. It includes computer viruses, worms, trojan horses, spyware, adware, and other malicious and unwanted software. Malware should not be confused with defective software, that is, software which has a legitimate purpose but contains harmful bugs [2].

According to [4], malicious programs can be divided into two categories

- Those that need a host program (that cannot exit independently of some actual application program, utility or system program e.g. virus, logic bombs, backdoors are examples); and
- Those that are independent (self contained programs that can be scheduled and run by the operating system, e.g., worms, zombies)

And also they can be differentiated between those software threats that do not replicate (activated by a trigger, e.g., logic bombs, backdoors and zombie programs) and those that do (e.g., viruses and worms). [4]

Taxonomy of malicious programs is described in table 1 [4]

Table 1- Taxonomy of Malware

Name	**Description**
Virus	Attaches itself to a program and propagates copies if itself to other programs
Worm	Program that propagates copies of itself to other computer
Logic Bombs	Triggers actions when condition occurs
Trojan Horse	Programs that contains unexpected additional functionality
Backdoor(trapdoor)	Program modification that allows unauthorized access to functionality
Exploits	Code specific to a single vulnerability or set of vulnerabilities
Downloaders	Program that installs other items on a machine that is under attack, usually a downloader is sent in an email
Auto rooter	Malicious hacker tools used to break into new machines remotely
Kit(virus generator)	Set of tools for generating new viruses automatically

Spammer Programs	Used to send large volumes of unwanted emails
Flooders	Used to attack network computer systems with a large volume of traffic to carry out a denial of service (DoS) attack
Key-Loggers	Captures keystrokes on a compromised system
Rootkit	Set of hacker tools used after attacker has broken into a computer system and gained root level access
Zombie	Program activated on an infected machine that is activated to launch attacks on other machines.

2.3 PURPOSE OF MALWARE

Since the rise of widespread broadband Internet access, more malicious software has been designed for a profit motive. For instance, since 2003, the majority of widespread viruses and worms have been designed to take control of users' computers for black-market exploitation [2].

A description of several famous malicious computer programs (computer viruses and worms) that caused extensive harm and reviews of legal consequences of each incident, including the nonexistent or lenient punishment of the program's author are described in detail at [3]

- E-mail delivering these malicious programs is deceptively or fraudulently labeled, so to encourage victims to open an e-mail attachment containing the malicious program.
- Many malicious programs delete or alter data in files on the victim's hard drive, a result that has no benefit to the author of the malicious program, except glee in harming other people. This is clearly a criminal act by the author of the malicious program.
- There is an enormous total cost of removing the virus or worm from many computers. Some of these malicious programs described in [3] infected

more than 10^5 computers worldwide. The cost of removing the program from each computer is in millions of dollars.

- Beginning with the Melissa virus in March 1999, many of these malicious programs sent copies of the program in e-mail bearing the victim's from: address, when the victim had neither composed the e-mail message nor authorized the transmission. I believe that such sending of e-mail is, or ought to be, a criminal act.

- Malicious programs that propagate by e-mail will clog e-mail servers with millions of copies of a virus or worm, thus delaying receipt of useful e-mail, or causing valid messages to be lost in a flood of useless e-mail i. e denial of Service.

According to [10], Malicious programs can be divided into the following groups: worms, viruses, Trojans, hacker utilities and other malware (DoS and DDoS Tools ,Hacker Tools and Exploits ,Flooders ,Constructors and VirTools ,Nukers ,FileCryptors and PolyCryptors ,PolyEngines). All of these are designed to damage the infected machine or other networked machines.

2.4 VULNERABILITY TO MALWARE

The "system" under attack may be of various types, e.g. a single computer and operating system, a network or an application [2]. Various factors make a system more vulnerable to malware:

- Homogeneity – e.g. when all computers in a network run the same OS, if you can break that OS, you can break into any computer running it.
- Bugginess – most systems containing errors which may be exploited by malware.
- Unconfirmed code – code from a floppy disk, CD or USB device may be executed without the user's agreement.

- Over-privileged users – some systems allow all users to modify their internal structures.
- Over-privileged code – most popular systems allow code executed by a user all rights of that user.

According to [10], Malware appears in any given environment when the following criteria are met:

- The operating system is widely used
- Reasonably high-quality documentation is available
- The targeted system is insecure or has a number of documented vulnerabilities

Internet is the main way all the malware reaches victim computers. The main channels are email, Usenet, peer-to-peer (P2P) file sharing networks and different 'live chat' networks like Internet Relay Chat (IRC), numerous 'Instant Messengers', and so on.[6].

2.5 RELATED WORK

In the mid-late 90's, IBM Research conducted research into developing an anti-virus modeled on the human biological immune system [5]. Their research led to the development and deployment of a prototype system that demonstrably worked and could automatically detect and generate code to remove new (previously undetected) viruses (immunization) and transmit the removal code to all their customers (e.g., over the Internet). This system required no manual (human) intervention and worked incredibly well (the system was patented by IBM Research and eventually the technology was bought by a major anti-virus OEM that decided not to deploy it, most probably to protect their main revenue stream which was from anti-virus signature updates).

In a paper at IBM research center they have outlined the problems, to suggest approaches, and to encourage those interested in research in this field to pursue them [1]. They have discussed five problems out of which 3 are relevant to our research Firstly as more viruses are written for new platforms; new heuristic detection techniques must be developed and deployed. But we often have no way of knowing, in advance, the extent to which these techniques will have problems with false positives and false negatives. That is, we don't know how well they will work or how many problems they will cause. IBM showed that analytic techniques can be developed which estimate these characteristics and suggest how these might be developed for several classes of heuristics. Secondly IBM managed to deploy a “digital immune system” that finds new viruses, transmits them to an analysis center, analyzes them, and distributes cures worldwide, automatically, and very quickly. At that time, there were few instances of worms - freestanding virus-like programs that spread themselves and may never be present in the computer's file system at all, and consequently, virtually all anti-virus technology (which is based on technology developed more than 10 years ago) relies on detecting and removing viruses from a file system

This project also addresses this research issue by providing an engine that may be deployed in a proactive malware analysis system (e.g., a network router) or a reactive malware analysis system (e.g., desktop application).

Another major problem faced by anti virus companies is the code obfuscation by the malware writers. According to [7], code obfuscation is used extensively by authors of malicious code to avoid detection. Many viruses utilize obfuscation techniques to subvert virus scanners by continually changing their code signature with obfuscating transformations. The two main types of malicious code that use obfuscation techniques to hide themselves from virus scanners are polymorphic and metamorphic viruses. They rely on techniques that change their code signature each infection generation, making it impossible for string matching algorithms to detect their presence. This is a relevant research issue that is currently outside the scope of this project, but may be addressed in future (as a continuation of this research project).

2.5.1 Malware Trend

An interesting observation described in [6] that over the past few years malware writers apparently shifted their efforts from creating viruses and worms ‘for fun’, from cyber vandalism, to creating backdoors, remotely-controlled bots, password stealers, etc. pretty much ‘for profit’. In fact, today we are seeing 8 to 10 times more new non-replicating malware per month than new viruses or worms. The chart below shows annual numbers of replicating and non-replicating malware samples added to the *McAfee AVERT* master malware collection. Today the trend is mostly non-replicating malware, trying to stay inconspicuous, for theft and control monetary gains.

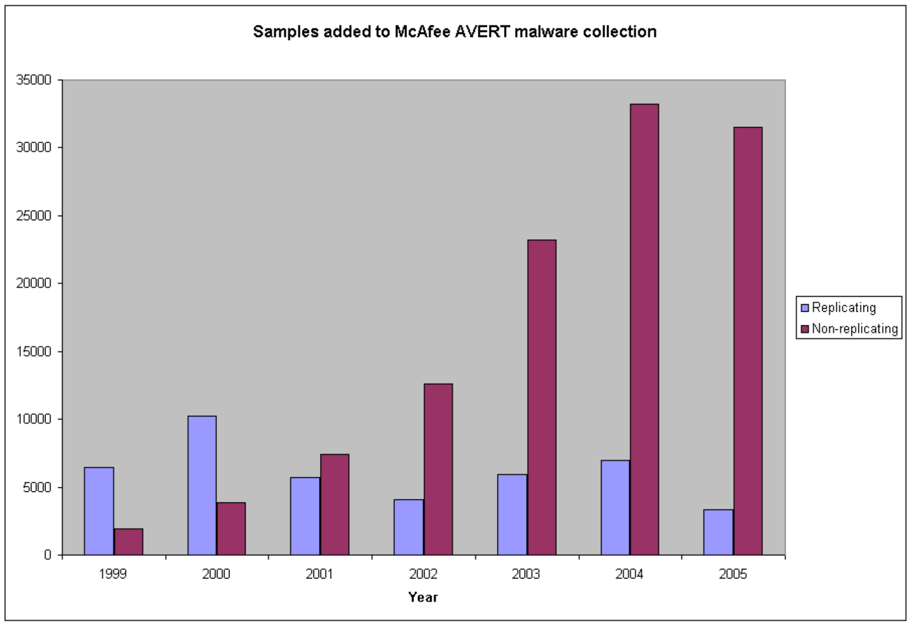


Figure 4 - Malware Trends[6]

So by looking at the above chart we can clearly see the increasing trends of malware other then just viruses and worms.

The trends in virusology that we observe today have their primary roots in the second half of 2003. Internet worms Lovesan, Sobig, Swen and Sober all not only caused global epidemics, but also profoundly changed the malware landscape. Each of these malicious programs set new standards for virus writers [10].

Once a piece of malware which uses fundamentally new techniques to propagate or infect victim machines appears, virus writers are quick to adopt the new approach. Today's new threats all incorporate characteristics of Lovesan, Sobig, Swen or Sober. Therefore, in order to understand what virus writers are doing currently, and to predict what the future may bring, we need to examine this quartet of worms carefully [10].

To summarize, these malware set the following trends as described in [10]:

- Exploiting critical vulnerabilities in MS Windows
- Propagation via the Internet through direct connections to victim machines
- Organising DoS and DDos attacks on key websites
- The creation of networks of infected machines to serve as epidemic platforms
- Mass mailing of malware using spammer techniques
- Mass mailings of links to infected sites via email or ICQ
- Trojan proxies become a separate class of malware closely linked to spammers
- Using vulnerabilities or holes created by other viruses
- Active deletion of competitor viruses
- Propagation in archived files (Bagle & NetSky variants)
- Propagation in password-protected compressed files: passwords were either included as text strings or as graphics (Bagle)
- Abandoning propagation by email: instead, the malicious programs spread by directing infected machines to sites where the worm's body was

downloaded or downloading the worm's body from previously infected machines (NetSky)

Malicious software has turned into a big business. As technology has evolved, so have malware. In the space of a couple of decades, we have seen computers change almost beyond recognition. The extremely limited machines which booted from a floppy disk are now powerful systems that can send huge volumes of data almost instantaneously, route email to hundreds or thousands of addresses, and entertain individuals with movies, music and interactive Web sites. And virus writers have kept pace with these changes [10].

While the viruses of the 1980s targeted a variety of operating systems and networks, most viruses today are written to exploit vulnerabilities in the most commonly used softwares [10].

Chapter 3

METHODOLOGY

Since windows operating systems are most vulnerable to malware than any other operating system as they share the home user market at the max therefore we began our investigation with the analysis of the structure of executable (image) files i.e. PE file format under the Microsoft Windows NT® operating system. The name "Portable Executable" refers to the fact that the format is not architecture-specific. The PE file format draws primarily from the COFF (Common Object File Format) specification that is common to UNIX® operating systems. And to remain compatible with previous versions of the MS-DOS® and Windows operating systems, the PE file format also retains the old familiar MZ header from MS-DOS. lets began with the structure of PE files.

We developed a program that dumps the structure of PE files. It is used to analyze both malicious and benign softwares. Let's look at the structure of PE files first.

3.1 STRUCTURE OF PE FILES

The PE file format is organized as a linear stream of data. It begins with an MS-DOS header, a real-mode program stub, and a PE file signature. Immediately following is a PE file header and optional header. Beyond that, all the section headers appear, followed by all of the section bodies. Closing out the file are a few other regions of miscellaneous information, including relocation information, symbol table information, line number information, and string table data. All of this is more easily absorbed by looking at it graphically, as shown in Figure 4.[12]

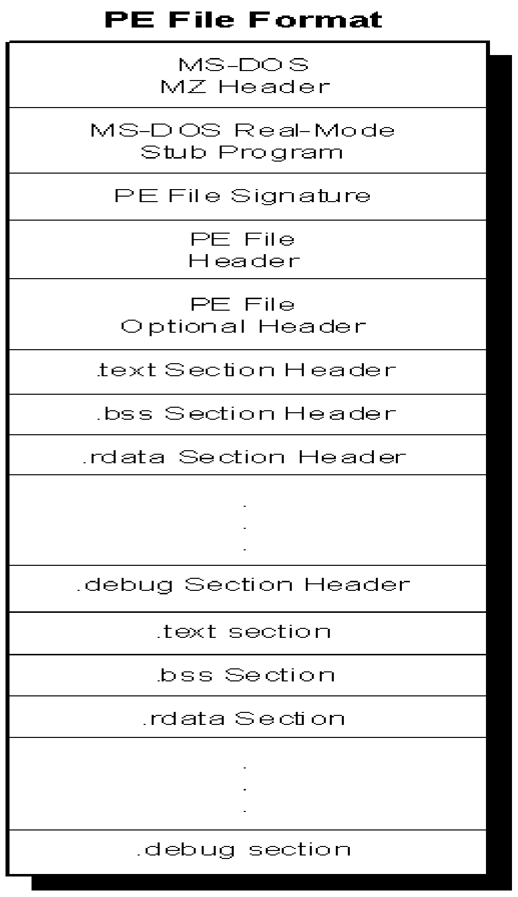


Figure 5 - PE File Format [12]

Starting with the MS-DOS file header structure, each of the components in the PE file format is discussed below in the order in which it occurs in the file.

3.2 DEMO RESULTS:-

PEStructureAnalyzer.exe

Sample malware: **RavMon.exe**

Input: file name to analyze

Output: Dump of the file

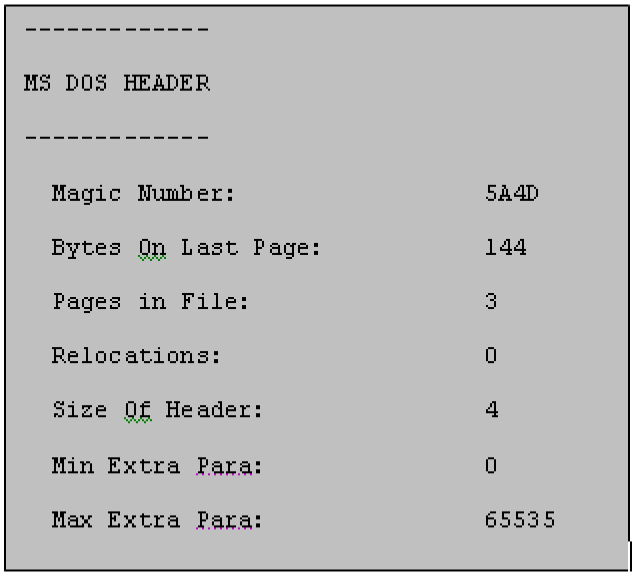
This program dumps the structure of PE files and gives the following information:-

The program first opens a file using **CreateFile** function and checks if the file exist or not and then calls **CreateFileMapping** function which creates a named or unnamed file-mapping object for the specified file and then calls **MapViewOfFile** function which maps a view of a file mapping into the address space of a calling process.

Now that we have the pointer of the memory mapping of the file, we can now dumps its structure, the out put of the program is in the following sequence

3.2.1 Ms Dos Header:-

It first dumps MS Dos header and gives the following attributes



```
-------------
MS DOS HEADER
-------------
  Magic Number:                  5A4D
  Bytes On Last Page:            144
  Pages in File:                 3
  Relocations:                   0
  Size Of Header:                4
  Min Extra Para:                0
  Max Extra Para:                65535
```

Figure 6- MSDOS Header Dump

The first field, **`Magic Number`**, is the so-called magic number. This field is used to identify an MS-DOS-compatible file type. All MS-DOS-compatible executable files set this value to **0x54AD**, which represents the ASCII characters ***MZ*** as defined in WINNT.h

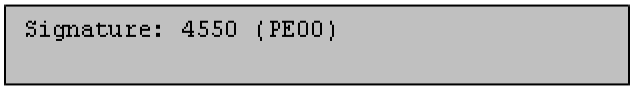
```
#define IMAGE_DOS_SIGNATURE             0x5A4D      // MZ
```

The final field, **`Addr Of New Header`**, is a 4-byte offset into the file where the PE file header is located. It is necessary to use this offset to locate the PE header in the file. For PE files in Windows NT, the PE file header occurs soon after the MS-DOS header with only the real-mode stub program between them.

3.2.2 Signature:-

Here signature of the file is verified. Three different file types are defined in WINNT.h

```
#define IMAGE_OS2_SIGNATURE           0x454E      // NE
#define IMAGE_OS2_SIGNATURE_LE        0x454C      // LE
#define IMAGE_NT_SIGNATURE            0x00004550  // PE00
```

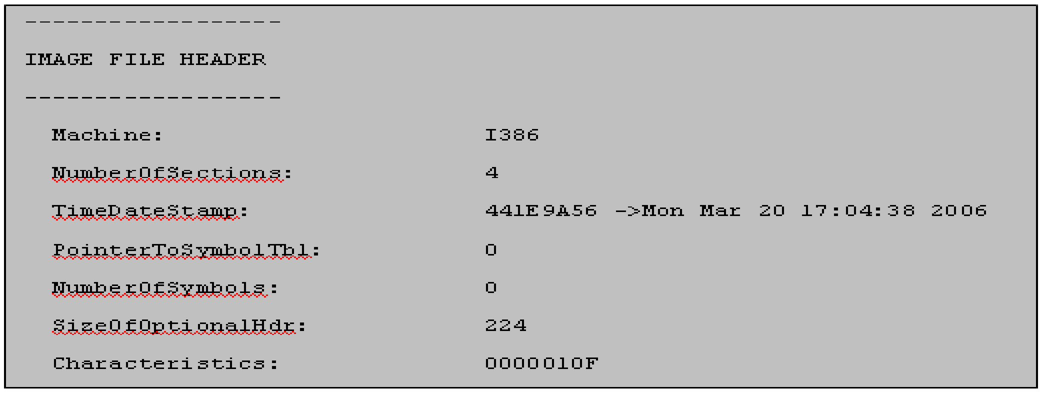


```
Signature: 4550 (PE00)
```

Figure 7-Signature in PE File

3.2.3 Image File Header:-

Through NT header we can get the pointer to the image file header. PEStructureAnalyzer gives the following output for the sample malware RavMon.exe



```
------------------
IMAGE FILE HEADER
------------------
  Machine:                       I386
  NumberOfSections:              4
  TimeDateStamp:                 441E9A56 ->Mon Mar 20 17:04:38 2006
  PointerToSymbolTbl:            0
  NumberOfSymbols:               0
  SizeOfOptionalHdr:             224
  Characteristics:               0000010F
```

Figure 8-Image File Header

The information in the PE file is basically high-level information that is used by the system or applications to determine how to treat the file. The ***Machine*** field is used to indicate what type of machine the executable was built for, such as the DEC® Alpha, MIPS R4000, Intel® x86, or some other processor. The system uses this information to quickly determine how to treat the file before going any further into the rest of the file data.

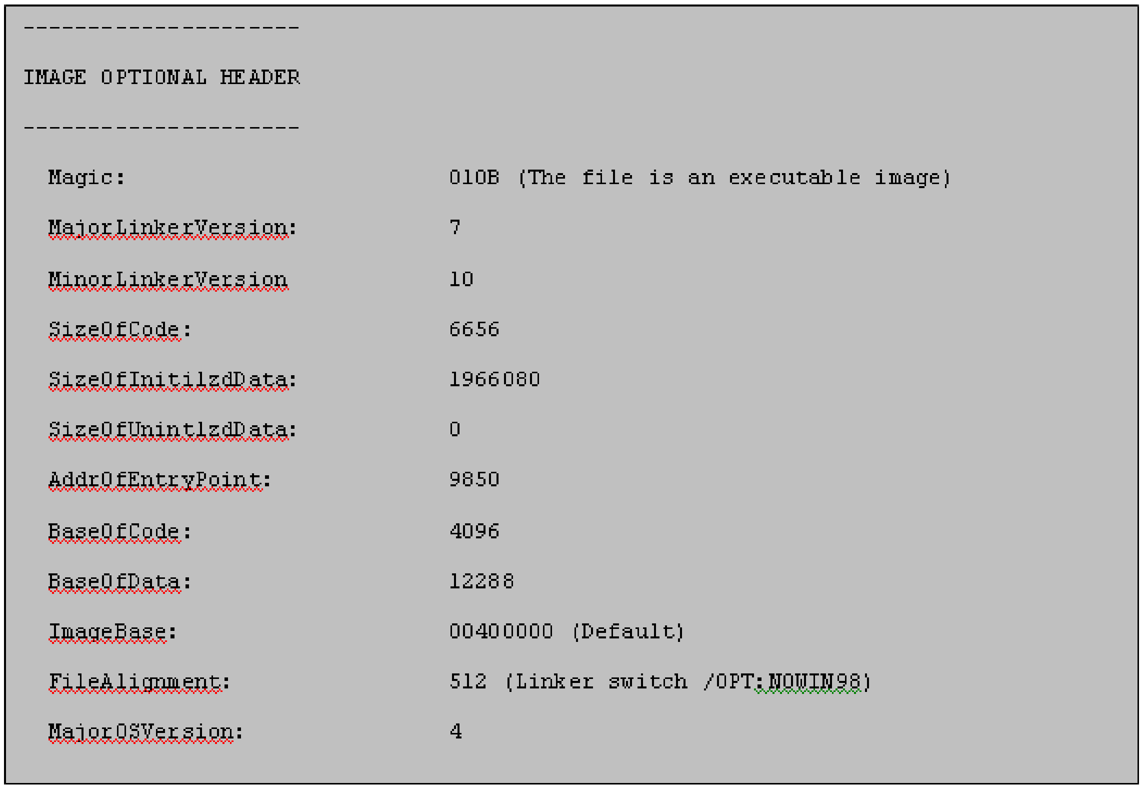
The ***Characteristics*** field identifies specific characteristics about the file.

One other useful entry in the PE file header structure is the ***NumberOfSections*** field. It turns out that you need to know how many sections--more specifically, how many section headers and section bodies--are in the file in order to extract the information easily. Each section header and section body is laid out sequentially in the file, so the number of sections is necessary to determine where the section headers and bodies end.

TimeDateStamp field represents the date and time the image was created by the linker. The value is represented in the number of seconds elapsed since midnight (00:00:00), January 1, 1970, Universal Coordinated Time, according to the system clock.

3.2.4 Image Optional Header:-

The next 224 bytes in the executable file make up the PE optional header. Though its name is "optional header," but this is not an optional entry in PE executable files. The optional header contains most of the meaningful information about the executable image,as given below in the following example:-



```
---------------------
IMAGE OPTIONAL HEADER
---------------------
  Magic:                        010B (The file is an executable image)
  MajorLinkerVersion:           7
  MinorLinkerVersion            10
  SizeOfCode:                   6656
  SizeOfInitilzdData:           1966080
  SizeOfUnintlzdData:           0
  AddrOfEntryPoint:             9850
  BaseOfCode:                   4096
  BaseOfData:                   12288
  ImageBase:                    00400000 (Default)
  FileAlignment:                512 (Linker switch /OPT:NOWIN98)
  MajorOSVersion:               4
```

Figure 9-Image Optional Header

- ***Magic*** shows the state of the image file
- ***MajorLinkerVersion, MinorLinkerVersion.*** Indicates version of the linker that linked this image.
- ***SizeOfCode.*** Size of executable code.
- ***AddressOfEntryPoint.*** Of the standard fields, the *AddressOfEntryPoint* field is the most interesting for the PE file format. This field indicates the location of the entry point for the application and, perhaps more importantly to system hackers, the location of the end of the Import Address Table (IAT). The following function demonstrates how to retrieve the entry point of a Windows NT executable image from the optional header.
- *BaseOfCode*. Relative offset of code (".text" section) in loaded image.
- *BaseOfData*. Relative offset of uninitialized data (".bss" section) in loaded image.

3.2.4.1 Windows NT Additional Fields

The additional fields added to the Windows NT PE file format provide loader support for much of the Windows NT-specific process behavior. Following is a summary of these fields.

- ***ImageBase.*** Preferred base address in the address space of a process to map the executable image to. The linker defaults to 0x00400000.
- ***SectionAlignment.*** Each section is loaded into the address space of a process sequentially, beginning at *ImageBase*. *SectionAlignment* dictates the minimum amount of space a section can occupy when loaded--that is, sections are aligned on *SectionAlignment* boundaries. Section alignment can be no less than the page size (currently 4096 bytes on the *x*86 platform) and must be a multiple of the page size as dictated by the behavior of Windows NT's virtual memory manager. 4096 bytes is the *x*86 linker default, but this can be set using the **-ALIGN: linker** switch.
- ***FileAlignment.*** Minimum granularity of chunks of information within the image file prior to loading. For example, the linker zero-pads a section body (raw data for a section) up to the nearest *FileAlignment* boundary in the file. This value is constrained to be a power of 2 between 512 and 65,535.
- ***SizeOfImage.*** Indicates the amount of address space to reserve in the address space for the loaded executable image. This number is influenced greatly by *SectionAlignment*. For example, consider a system having a fixed page size of 4096 bytes. If you have an executable with 11 sections, each less than 4096 bytes, aligned on a 65,536-byte boundary, the *SizeOfImage* field would be set to 11 * 65,536 = 720,896 (176 pages). The same file linked with 4096-byte alignment would result in 11 * 4096 = 45,056 (11 pages) for the *SizeOfImage* field. This is a simple example in which each section requires less than a page of memory. In reality, the linker determines the exact *SizeOfImage* by figuring each section individually. It first determines how many bytes the section

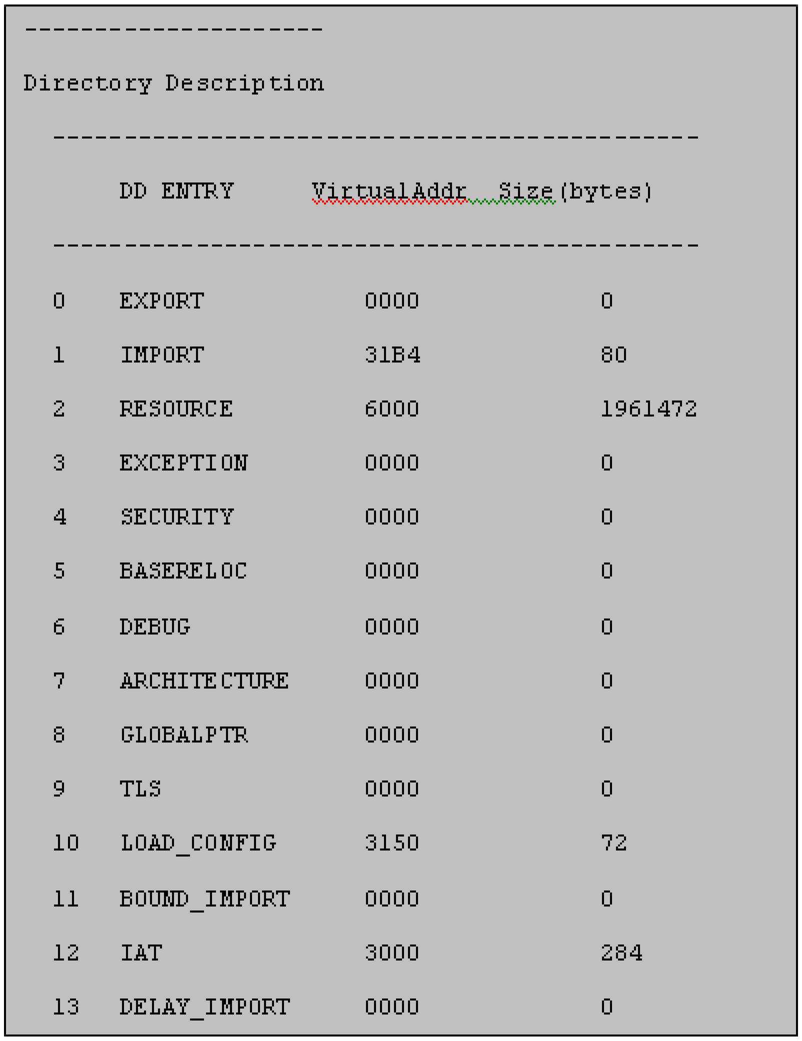
requires, then it rounds up to the nearest page boundary, and finally it rounds page count to the nearest *SectionAlignment* boundary. The total is then the sum of each section's individual requirement.

- ***SizeOfHeaders.*** This field indicates how much space in the file is used for representing all the file headers, including the MS-DOS header, PE file header, PE optional header, and PE section headers. The section bodies begin at this location in the file.
- ***CheckSum.*** A checksum value is used to validate the executable file at load time. The value is set and verified by the linker. The algorithm used for creating these checksum values is proprietary information and will not be published.
- ***Subsystem:.*** Field used to identify the target subsystem for this executable. Values are listed in winnt.h
    - IMAGE_SUBSYSTEM_UNKNOWN
    - IMAGE_SUBSYSTEM_NATIVE
    - IMAGE_SUBSYSTEM_WINDOWS_GUI
    - IMAGE_SUBSYSTEM_WINDOWS_CUI
    - IMAGE_SUBSYSTEM_OS2_CUI
    - IMAGE_SUBSYSTEM_OS2_CUI
    - IMAGE_SUBSYSTEM_POSIX_CUI
    - IMAGE_SUBSYSTEM_NATIVE_WINDOWS
    - IMAGE_SUBSYSTEM_WINDOWS_CE_GUI
- ***DllCharacteristics.*** Flags used to indicate if a DLL image includes entry points for process and thread initialization and termination.
- ***SizeOfStackReserve, SizeOfStackCommit, SizeOfHeapReserve, S izeOfHeapCommit.*** These fields control the amount of address space to reserve and commit for the stack and default heap. Both the stack and heap have default values of 1 page committed and 16 pages reserved. These values are set with the linker switches **-STACKSIZE:** and **-HEAPSIZE:**. (NOTE:256 pages are now for reserved stack/heap)

- ***LoaderFlags***. Tells the loader whether to break on load, debug on load, or the default, which is to let things run normally.
- ***NumberOfRvaAndSizes***. This field identifies the length of the *DataDirectory* array that follows. It is important to note that this field is used to identify the size of the array, not the number of valid entries in the array.
- ***DataDirectory*****.** The data directory indicates where to find other important components of executable information in the file. It is really nothing more than an array of **IMAGE_DATA_DIRECTORY** structures that are located at the end of the optional header structure. The current PE file format defines 16 possible data directories.

3.2.5 Data Directories:-

The data directory contains the locations and sizes of the important data structures in the PE file As defined in WINNT.H, the data directories are as given as output by PEStructureAnalyzer



```
---------------------

Directory Description

  --------------------------------------------

       DD ENTRY      VirtualAddr  Size(bytes)

  --------------------------------------------

  0    EXPORT             0000            0

  1    IMPORT             31B4            80

  2    RESOURCE           6000            1961472

  3    EXCEPTION          0000            0

  4    SECURITY           0000            0

  5    BASERELOC          0000            0

  6    DEBUG              0000            0

  7    ARCHITECTURE       0000            0

  8    GLOBALPTR          0000            0

  9    TLS                0000            0

  10   LOAD_CONFIG        3150            72

  11   BOUND_IMPORT       0000            0

  12   IAT                3000            284

  13   DELAY_IMPORT       0000            0
```

Figure 10-Data Directory

Each data directory is basically a structure defined as an **MAGE_DATA_DIRECTORY**.

To locate a particular directory, you determine the relative address from the data directory array in the optional header. Then use the virtual address to determine which section the directory is in. Once you determine which section contains the directory, the section header for that section is then used to find the exact file offset location of the data directory.

3.2.6 Section Table:-

Section headers are located sequentially right after the optional header in the PE file format. An application for Windows NT typically has the nine predefined sections named .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, and .debug Each section header is 40 bytes with no padding between them.

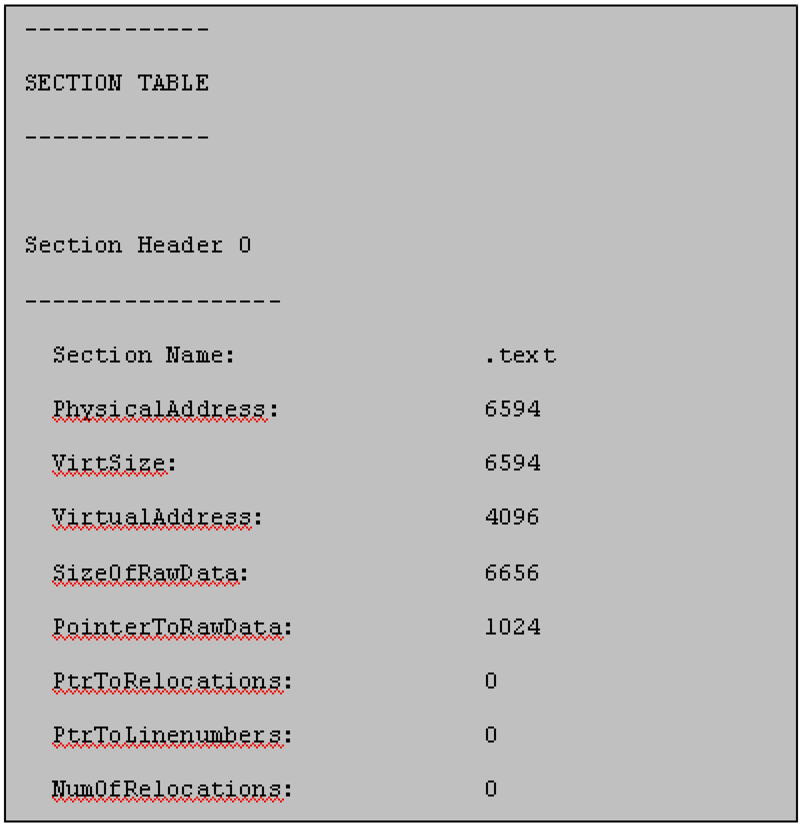
→The default behavior combines all code segments into a single section called **".text"** in Windows NT

→The **.bss** section represents uninitialized data for the application, including all variables declared as static within a function or source module.

→The **.rdata** section represents read-only data, such as literal strings, constants, and debugs directory information.

→All other variables (except automatic variables, which appear on the stack) are stored in the .data section. Basically, these are application or module global variables.

The dump of the first section header of the sample malware RavMon.exe is given below.



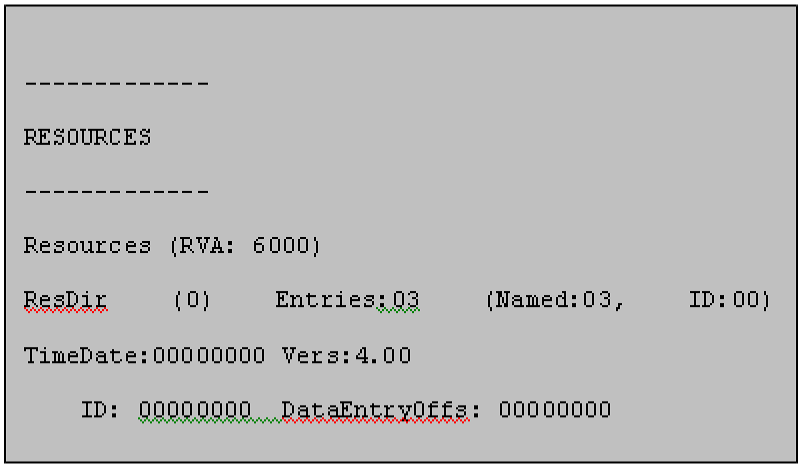
```
-------------

SECTION TABLE

-------------


Section Header 0

------------------

  Section Name:                    .text

  PhysicalAddress:                 6594

  VirtSize:                        6594

  VirtualAddress:                  4096

  SizeOfRawData:                   6656

  PointerToRawData:                1024

  PtrToRelocations:                0

  PtrToLinenumbers:                0

  NumOfRelocations:                0
```

Figure 11-Section Header

3.2.7 Resources:-

The .rsrc section contains resource information for a module. It begins with a resource directory structure like most other sections, but this section's data is further structured into a resource tree.

PEStructureAnalyzer checks for string table and dialogs table as well.

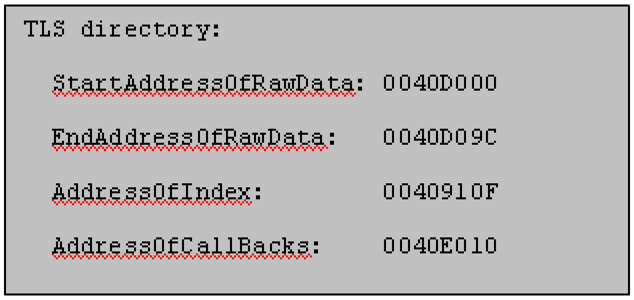


```
-------------

RESOURCES

-------------

Resources (RVA: 6000)

ResDir     (0)     Entries:03     (Named:03,     ID:00)

TimeDate:00000000 Vers:4.00

    ID: 00000000  DataEntryOffs: 00000000
```

Figure 12-Resource Header

3.2.8 TLS Directory:-

Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process may allocate locations in which to store thread-specific data. The TLS directory for the sample malware 09116343.EXE is given below:-



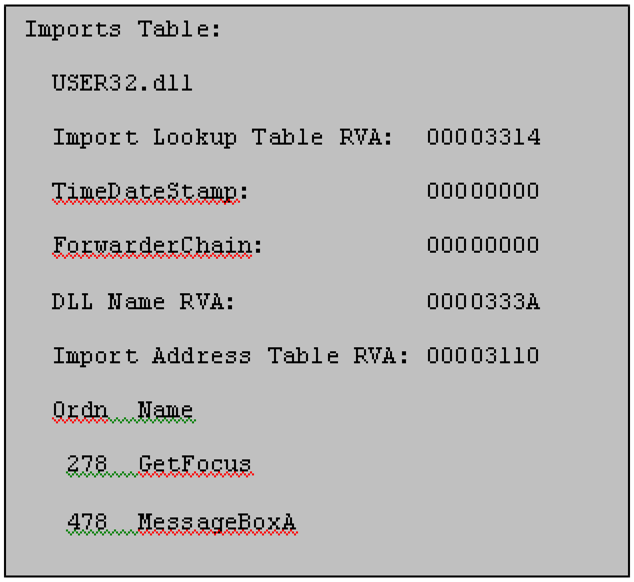
```
TLS directory:

  StartAddressOfRawData: 0040D000

  EndAddressOfRawData:   0040D09C

  AddressOfIndex:        0040910F

  AddressOfCallBacks:    0040E010
```

Figure 13-TLS Directory

3.2.9 Import Table:-

An import function is a function that is not in the caller's module but is called by the module, thus the name "import". The import functions actually reside in one or more DLLs. Only the information about the functions is kept in the caller's module. That information includes the function names and the names of the DLLs in which they reside.

Data directory keeps the information of import tables. The part of import table in the sample malware retrieved through PEStructureAnalyzer is given below.



```
Imports Table:
  USER32.dll
  Import Lookup Table RVA:   00003314
  TimeDateStamp:             00000000
  ForwarderChain:            00000000
  DLL Name RVA:              0000333A
  Import Address Table RVA:  00003110
  Ordn  Name
   278  GetFocus
   478  MessageBoxA
```

Figure 14-Import Table Dump

3.2.10 Export Table:-

The place in the DLLs where the PE loader looks for the addresses of the functions is the export table. When a DLL/EXE exports a function to be used by other DLL/EXE, it can do so in two ways:

it can export the function

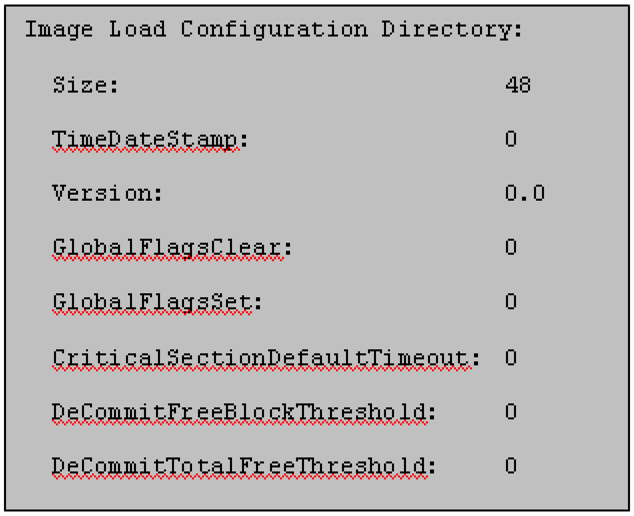
- by name or

- by ordinal only (An ordinal is a 16-bit number that uniquely identifies a function in a particular DLL)

Say if there is a function named "GetSysConfig" in a DLL, it can choose to tell the other DLLs/EXEs that if they want to call the function, they must specify it by its name, ie. GetSysConfig. The other way is to export by ordinal.. This number is unique only within the DLL it refers to. For example, in the above example, the DLL can choose to export the function by ordinal, say, 16. Then the other DLLs/EXEs which want to call this function must specify this number in GetProcAddress. This is called export by ordinal only.

3.2.11 Image Load Configuration Directory:-

This directory contains the load configuration data of an image. The output for sample malware is given below:-

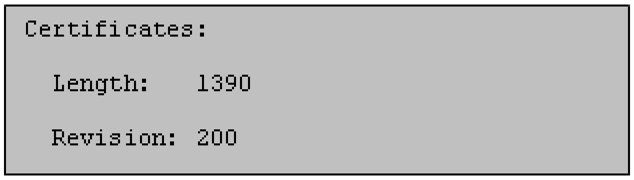


```
Image Load Configuration Directory:

  Size:                             48

  TimeDateStamp:                    0

  Version:                          0.0

  GlobalFlagsClear:                 0

  GlobalFlagsSet:                   0

  CriticalSectionDefaultTimeout:    0

  DeCommitFreeBlockThreshold:       0

  DeCommitTotalFreeThreshold:       0
```

Figure 15-Image Load Configuration Directory

3.2.12 Certificates:-

This encapsulates a signature used in verifying executable files. The output retrieved from the program for sample exe i.e. SmileboxInstaller.exe is given below:-



```
Certificates:
  Length:   1390
  Revision: 200
```

Figure 16-Certificates in PE file

3.2.13 Hex Dump:-

This function gives the hex dump of an executable from 1st byte to the end of the file.

3.3 COMMON FEATURES IN MALWARE THROUGH PE DUMP:-

By taking the Pe Dumps of various executables we were able to find out the following features that distinct a malware from benign software.

3.3.1 Size

By taking the average of sizes of more than 300 malware samples, we came to the conclusion that most of the samples are less than 200K

3.3.2 MZ

Often things are not always as they seem. Spammers, Internet bottom feeders, and others with ill intent often try to mask what is in reality malware so if a file is looking suspicious, the characters "MZ" in the file can tell us that it is actually an executables rather than an image, ziped file, a video file etc.

3.3.3 Packed

What is the point of compressing malware down to a smaller size? Well it is not really all about simply shrinking the actual size, but more importantly about trying to evade anti-virus scanners so check whether a file is packed or not?

3.3.4 Suspicious Dates

- TimeDateStamp: 00001000 ->Thu Jan 01 06:08:16 **1970**
- TimeDateStamp: 63617055 -> Wed Nov 02 00:15:33 **2022**
- No Creation Date at all.

3.3.5 Window Subsystem:-

It is seen that most if the malware samples run in windows character subsystem rather than running in graphical user interface subsystem.

3.3.6 Common Dll's

- "ADVAPI32.DLL"--Tries to Access Registry
- "WININET.DLL"--Tries to Conect the Internet:can be a Malware
- "URLMON.DLL"--Downloads Data from internet

  "URLDownloadToFileA"--Downloads bits from the Internet and saves them to a file
- "WS2_32.DLL"--Tries to Access Internet
- "WSOCK32.DLL"--"Tries to Access Internet

Chapter 4

REVERSE ENGINEERING

Then we attempted to document an approach to reverse engineer malicious software by looking at the binary code of the executable through OlyDbg.exe .we found the following code sequence in order to retrieve a sequence of API calls in an executable. The code sequence for two types of call instructions (i.e. FF15 and E8) in any executable is given below.

4.1 DECODE SEQUENCE

FF15 | FF25 | FF35

o00402BF0 FFI5 a54714200

- ➔ followed by 4 Byte Memory address
    - o locate import table and see if Memory address (as offset) is with in import table range
    - o compare offset if greater than code section and less than end of import table
    - o if true
        - ▪ Calculate VA and RVA difference of the import section and subtract this difference from offset and jump to this offset.
- ➔ See next 4 Byte Memory Address
- ➔ Jump to this address after subtracting VA and RVA difference of the import section
- ➔ Retrieve the string

Example PEStructureAnalyzer.exe

o00402BF0 FFI5

- ➔ followed by 4 Byte Memory Address a54714200
- ➔ o00427154 > o00401000 (VA of code section)
- ➔ o00427154 < o00428000 (VA of end of import section)
- ➔ true
    - o VA of import section is o00427000
    - o RVA of import section is o00426000

- VA and RVA difference = 1000
- Therefore subtract 1000 from o00427154 i.e. o00426154
- Jump to o00426154 and retrieve next 4 Bytes a9C720200
- subtract 1000 from o0020729C i.e. o0020629C
- jump to o0020629C +1
- Retrieve the string CreateFileA

<u>E8</u>

o0040108D E8 aB2120000

- ➔ Calculate VA and RVA difference of the code section and subtract this difference from offset (pointing to E8) and jump to this offset
- ➔ followed by 4 Byte Memory address
- ➔ if 4th Byte is 00 then proceed.
  - add the offset and the next offset after the instruction E8
  - jump to that offset
  - if FF15 || FF25 || FF35 then follow the above procedure

<u>Example msblast.exe</u>

o0040108D E8

- ➔ VA of code section o00401000
- ➔ RVA of code section o00000400
- ➔ Code section VA n RVA difference is C00
- ➔ Therefore subtracting C00 from o0040108D gives o0040048D and jump to it
- ➔ followed by 4 Byte Memory Address aB2120000
- ➔ 4th Byte is 00 hence true
  - adding a000012B2 and next offset after E8 instruction i.e. 00000492 + 000012B2 gives o00001744
  - jump to this offset
  - if FF15 || FF25 || FF35 then follow the above procedure to retrieve the string i.e. RtlUnwind

<u>E8</u>

o004010D2 E8 aC9FFFFFF

- ➔ Calculate VA and RVA difference of the code section and subtract this difference from offset (pointing to E8) and jump to this offset
- ➔ followed by 4 Byte Memory address
- ➔ if 3rd and 4th Bytes are FFFF then proceed.
  - o add 1st two Bytes of offset and the offset of next instruction E8
  - o subtract 10000
  - o jump to that offset
  - o if FF15 || FF25 || FF35 then follow the above procedure

<u>Example SVOHOST.exe</u>

o004010D2 E8

- ➔ VA of code section o00401000
- ➔ RVA of code section o00001000
- ➔ Code section VA n RVA difference is 00000000
- ➔ Therefore subtracting 00000000 from 004010D2 gives 004010D2 and jump to it
- ➔ followed by 4 Byte Memory Address aC9FFFFFF
- ➔ 3rd n 4th Bytes are FFFF true
  - o adding FFC9 and next offset after E8 instruction i.e. 000010D3 + FFC9 gives o000010A0
  - o jump to this offset
  - o if FF15 || FF25 || FF35 then follow the above procedure to retrieve the string i.e. GetStartupInfoA

4.2 ANALYSIS:--

After taking the sequence of Api calls we developed a program which associates each function call with its Dll and counts the number of calls made for each Dll, the percentages of the count for the 35 Dlls that we found in our sample test sets for both malware n benign samples is shown in the following section.

4.2.1 Dump of Dlls Percentages for Our Test Samples

Our sample test set consists of 135 malware and 51 benign samples and based on the testing we found the following 35 Dll’s and combination of any of these Dll’s an executable is created.

KERNEL32.DLL
USER32.DLL
GDI32.DLL
ADVAPI32.DLL
VERSION.DLL
WSOCK32.DLL
SHELL32.DLL
COMCTL32.DLL
URLMON.DLL
WININET.DLL
WS2_32.DLL
CDINTOUCH50.DLL
ACDCLCLIENT30.DLL
SHFOLDER.DLL
SHLWAPI.DLL
MPR.DLL
SHELLINTMSR30.DLL
LMCLIENT.DLL
IPWSS15.DLL
CINDCODE.DLL
MFC71.DLL
MSVCR71.DLL
OLEAUT32.DLL
MSVCP71.DLL
MSVCRT.DLL
OLE32.DLL
SETUPAPI.DLL
WINTRUST.DLL
CRYPT32.DLL
MSVFW32.DLL
OLEACC.DLL
WINSPOOL.DRV
COMDLG32.DLL
CRTDLL.DLL
IPHLPAPI.DLL

Table 2-35 Dll's

The percentages of each Dll for our sample test set are shown in the following graph.

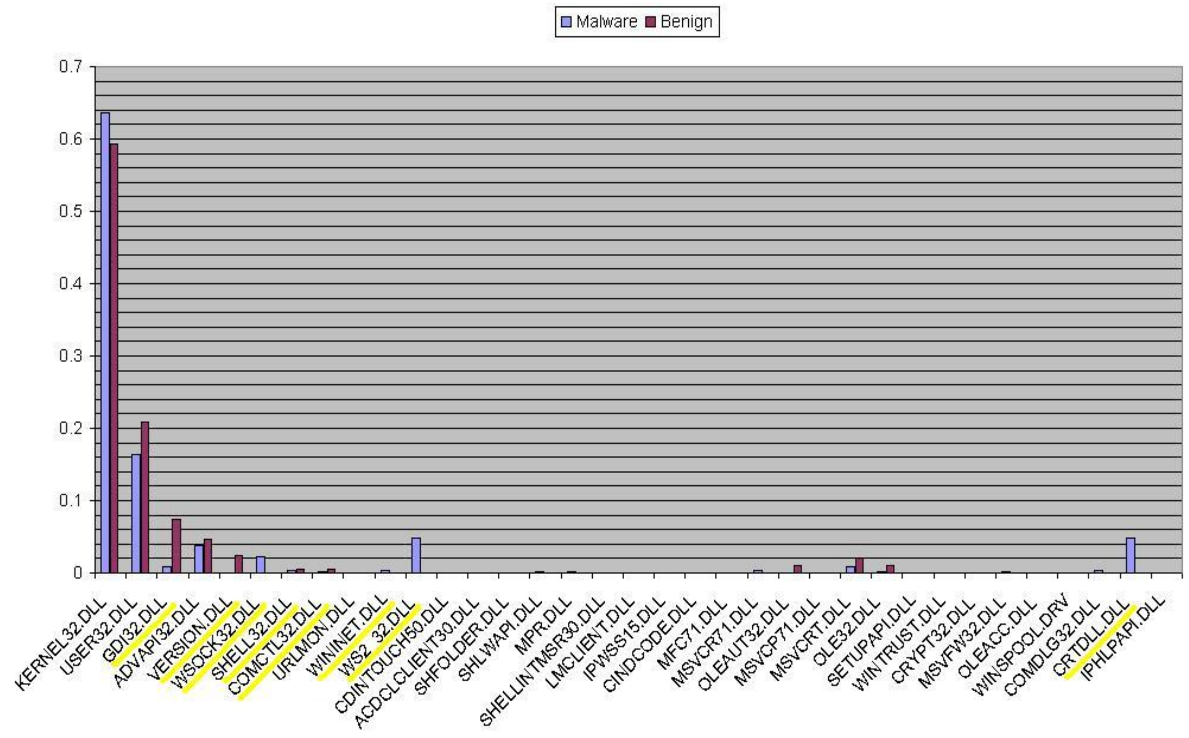


Figure 17-Percentages of Each Dll

The results clearly show the Dlls that are differing in both malware and benign samples. On the basis of the above result we selected the 8 Dll's as the basis for our decision criteria. The percentages of the selected Dlls' for each malware and benign sample from our program are shown in the following graph.

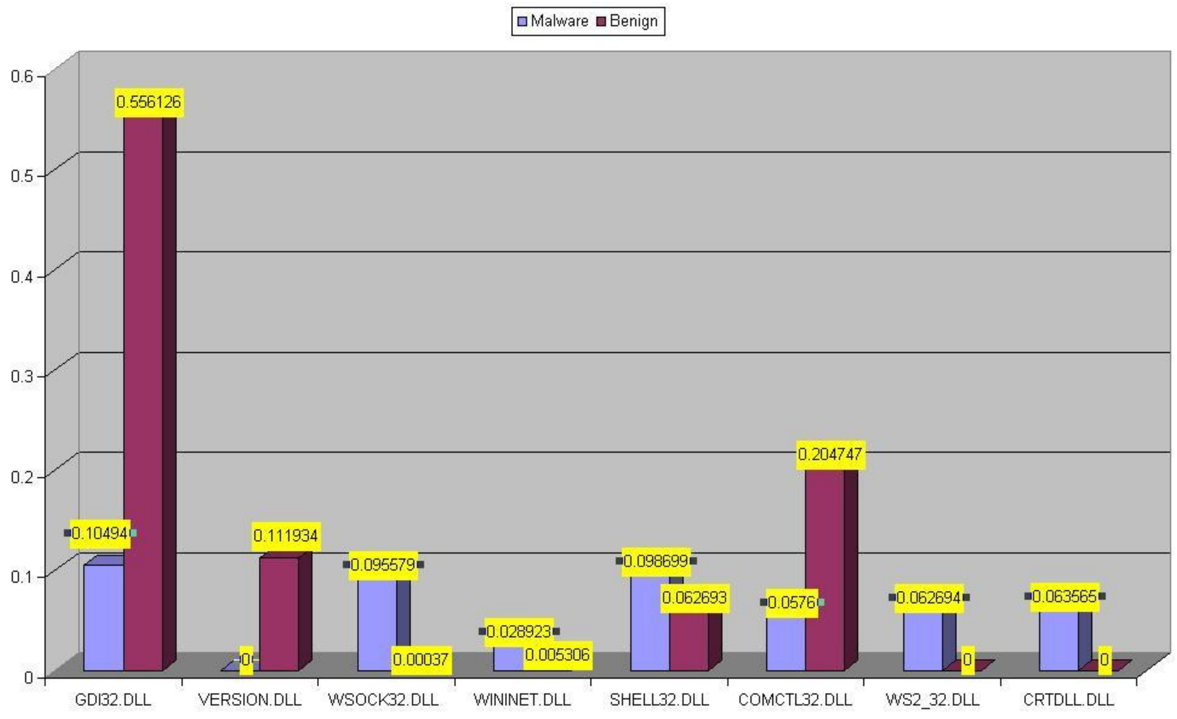


Figure 18-Percentages of Selected Dlls

The above graph clearly shows the usage of the selected Dll's in both malware and benign samples, which is quite distinct. In order to improve the results, total API calls and total number of Dll's used for each sample were caclulated for each sample test set .The following graph shows the average number of API calls and Dll's that are used for malware and benign samples each.

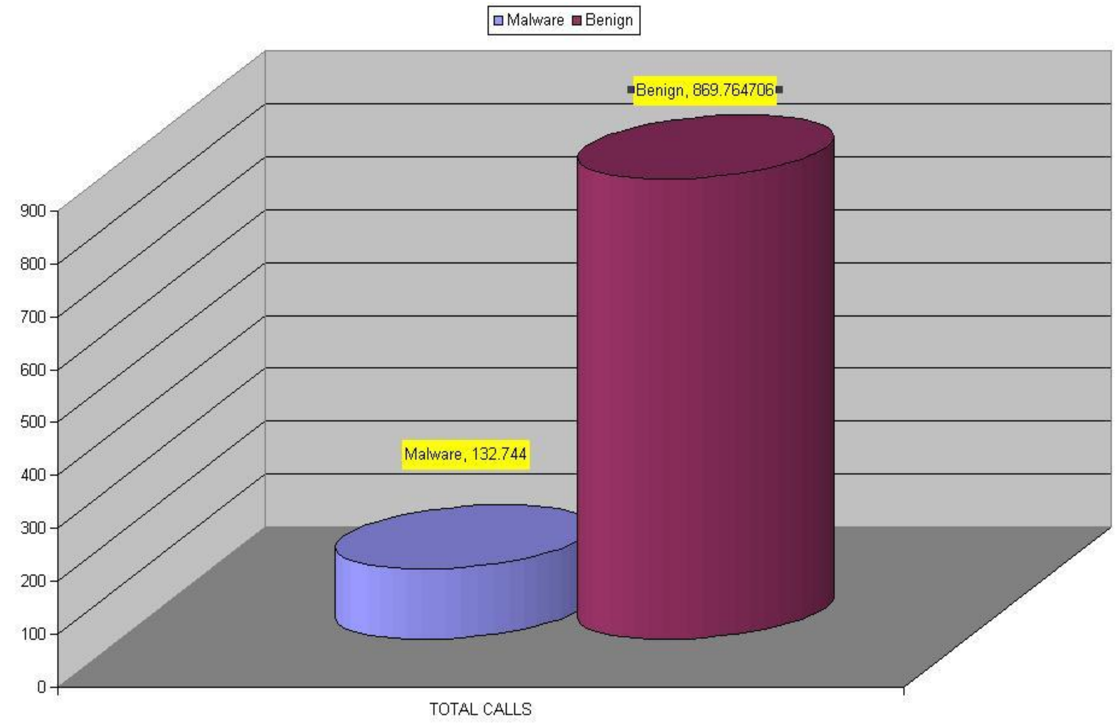


Figure 19-Total API Calls Averages

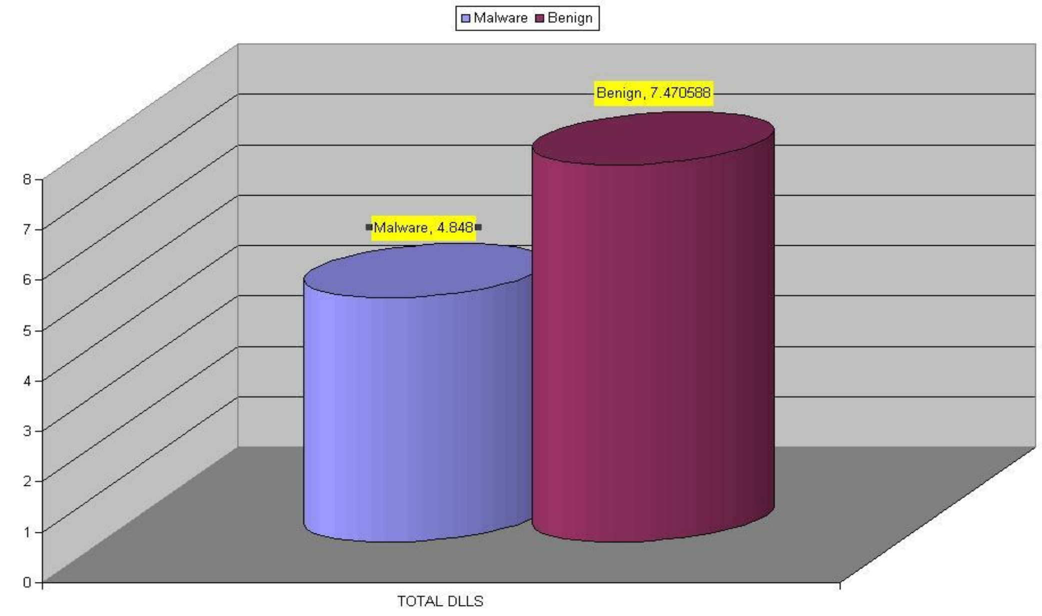


Figure 20-Averages Count of Dll's

4.2.2 Averages of selected Dll's;-

The following table shows the averages of percentages of the selected Dll's, total number of API calls, number of Dll's used for malware and benign samples which shows the difference between both malware and benign samples .

DLL Name	**Malware**	**Benign**
GDI32.DLL	0.104940	0.556126
VERSION.DLL	0	0.111934
WSOCK32.DLL	.095579	0.000370
WININET.DLL	.028923	0.005306
SHELL32.DLL	0.098699	0.062693
COMCTL32.DLL	.057600	0.204747
WS2_32.DLL	.062694	0.000000
CRTDLL	.063565	0.000000
TOTAL API CALLS	132.744000	869.764706
TOTAL DLLS USED	4.848000	7.470588

Figure 21 - Averages Result

So on the basis of the above averages we chose Euclidean Distance and information Divergence as two approaches to see the detection rate.

4.2.3 Euclidean Distance:-

The Algorithm that we developed is based on the following formula. For each malware and benign sample, the percentage of each Dll is calculated and then a value of each Dll is compared with averages of both malware and benign samples.

- E(m,M) = sqrt((p1-ma1)+(p2-ma2)+…. +(p8-ma8))
- E(m,B) = sqrt((p1-ba1)+(p2-ba2)+…. +(p8-ba8))

- E(b,M) = sqrt((p1-ma1)+(p2-ma2)+…. +(p8-ma8))
- E(b,B) = sqrt((p1-ba1)+(p2-ba2)+…. +(p8-ba8))

m=malware sample

b=benign sample

p1=percentage of 1st Dll 1.e GDI32.DLL.

ma1=average percentage of all tha malware samples for 1st Dll i.e GDI32.DLL

The Decision Criteria is the following

E(m,M)<E(m,B) || [(mApi - mAverage) && (mDllCount - mAverageCount)]$\rightarrow$

malwar

E(b,M)<E(b,B) || [(bApi - bAverage) && (bDllCount - bAverageCount)] $\rightarrow$

malware

mApi=number of Api call for given malware sample

mAvergae=number of Dlls for given malware sample

bApi=number of Api call for given benign sample

bAvergae=number of Dlls for given benign sample

4.2.4 Information Divergence:-

The Formula for Information Divergence is the following. For each malware and benign sample, the percentage of each Dll is calculated and then a value of each Dll is compared with averages of both malware and benign samples.

- E(m,M) = p1*log(p1÷ ma1)+ p1*log(p2÷ ma2)+…. + p8*log(p8÷ ma8)

- E(m,B) = p1*log(p1÷ ma1)+ p1*log(p2÷ ma2)+…. + p8*log(p8÷ ma8)

- E(b,M) = p1*log(p1÷ ma1)+ p1*log(p2÷ ma2)+…. + p8*log(p8÷ ma8)
- E(b,B) = p1*log(p1÷ ma1)+ p1*log(p2÷ ma2)+…. + p8*log(p8÷ ma8)

.The Decision Criteria is the following

E(m,M) < E(m,B) || [(mApi - mAverage) && (mDllCount - mAverageCount)]→ malware

E(b,M) < E(b,B) || [(bApi - bAverage) && (bDllCount - bAverageCount)] → malware

After applying the above approaches on our samples test set we found the following detection rates for each approach

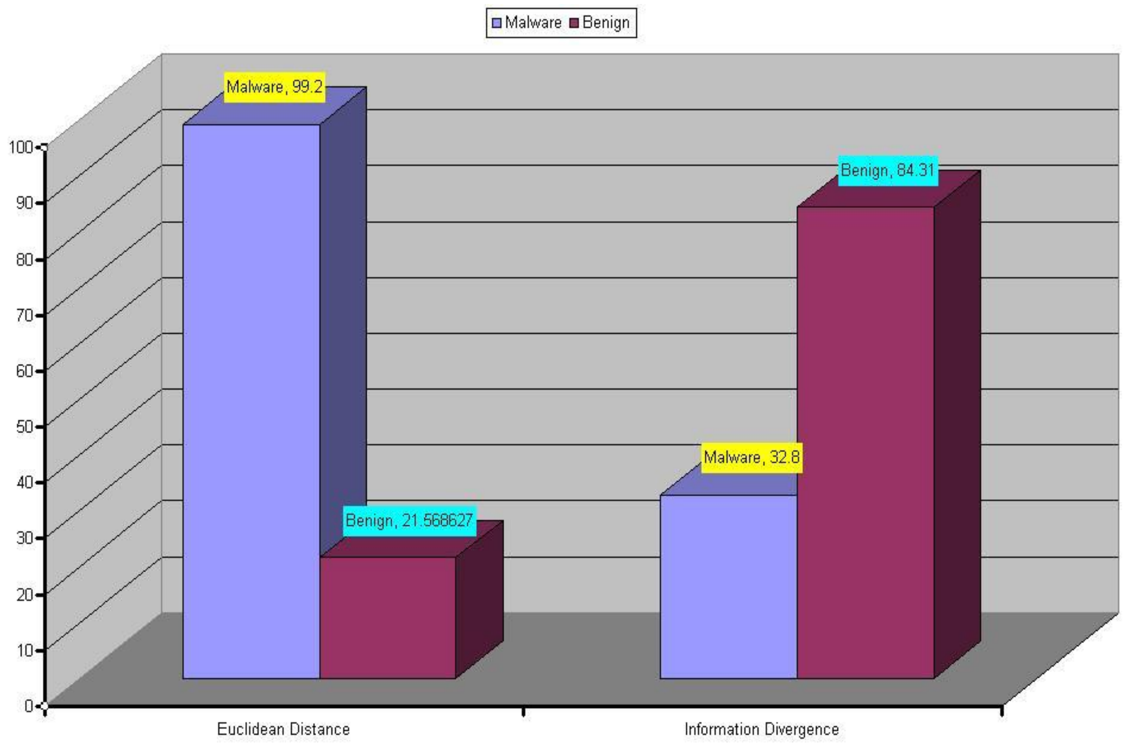


Figure 22 - Two Appraoches Percentages

	Malware Detection Rate	**Benign Detection Rate**	**False Positive**
Euclidean Distance	92.2%	21.568%	78.432%
Information Divergence	32.8%	84.34%	15.66%

Table 3 - Obtained Percentages of both the Approaches

The Above result is quite strange as the false positive rate is quite high for Euclidean Distance and malware detection rate is quite low for information divergence .So we decided to combine both the approaches and see the result .the truth table that we developed based on the above results is given below .

Euclidean Distance	**Information DIvergence**	**Output**
B	B	**B**
B	M	**B**
B	X	**B**
M	B	**B**
M	M	**M**
M	X	**M**

Table 4 - Truth Table

X→Undetectable

4.2.5 Explanation of Truth Table:-

As Euclidean Distance Declares almost everything as malware since is detection rate and false positive both are high therefore whenever a situation arises where Euclidean distance declares as M and information divergence as B, we have given priority to Information Divergence so output is B.

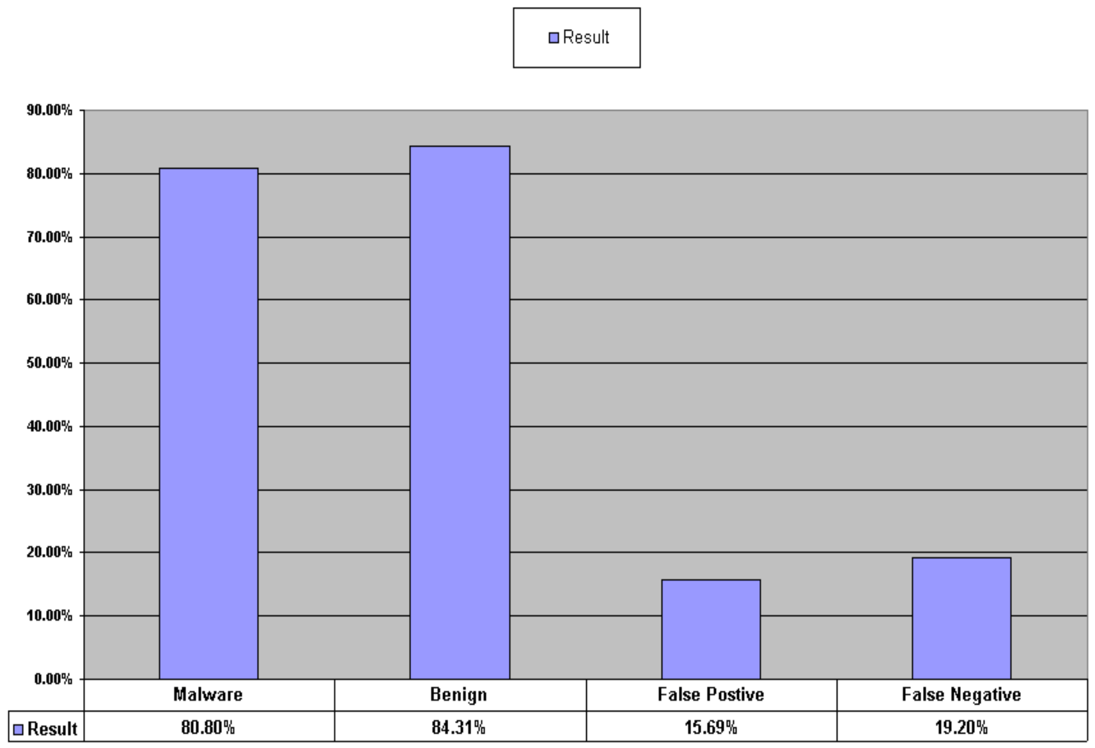
The situation where Euclidean distance gives B and Information Divergence M , the output is B since Euclidean distance declares B to very less number of samples so we have given priority to Euclidean Distance.

There are situations where none of the selected Dll is used by some executables then the result is declared as X (undetectable).

So there are only two situations where an executable is declared as malware as shown in the above table.

4.2.6 Result After combining both the approaches

After combining both the approaches following is the given graph



	Malware	Benign	False Postive	False Negative
Result	80.80%	84.31%	15.69%	19.20%

Figure 23 -Result based on previous Truth Table

Now the results are quite encouraging but in order to further improve the results we selected three more features to see if result would increase or not i.e. size, timestamp, console based feature of malware and benign samples based on the following criteria.

1. if **(size < 200K)** → malware
2. **Suspicious Timestamp i.e.**
   a. year 1970
   b. future Year like 2022
   c. No Creation Date at all
3. **Console based Subsystem**

Figure 24 – Three Added Features

The graph for the occurrence of these features in our sample test set for both benign and malware samples is given below:-

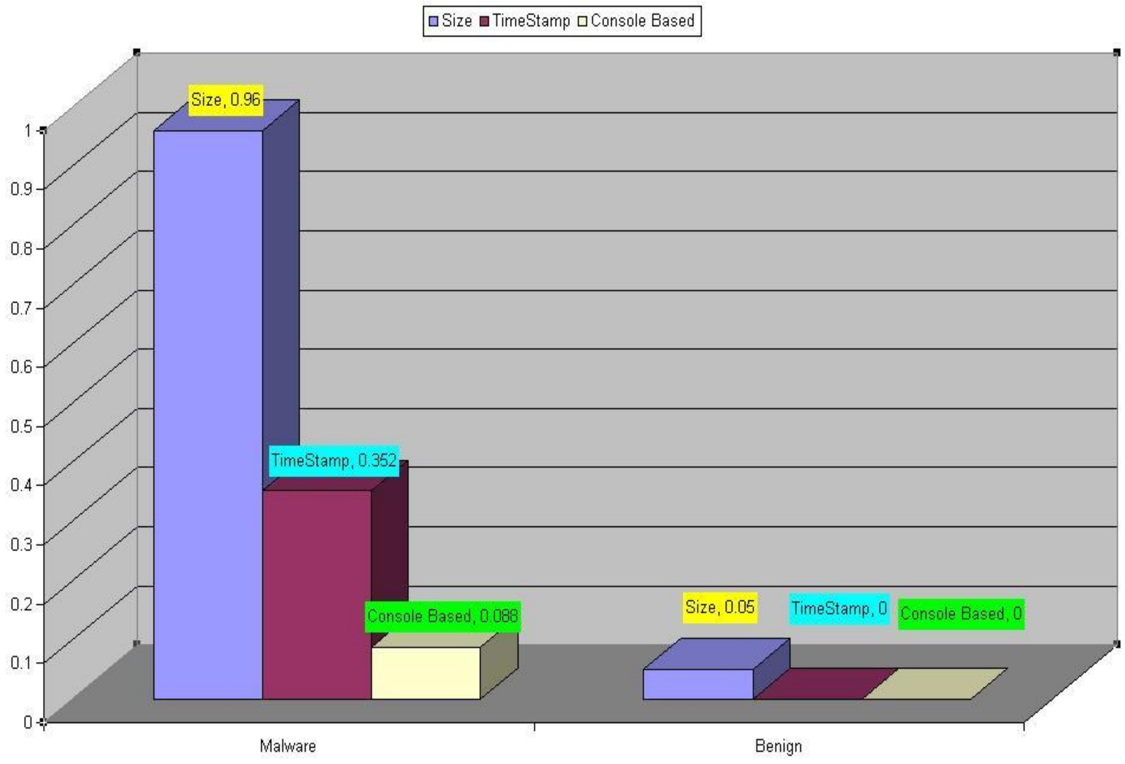


Figure 25 – Percentages of Other Features

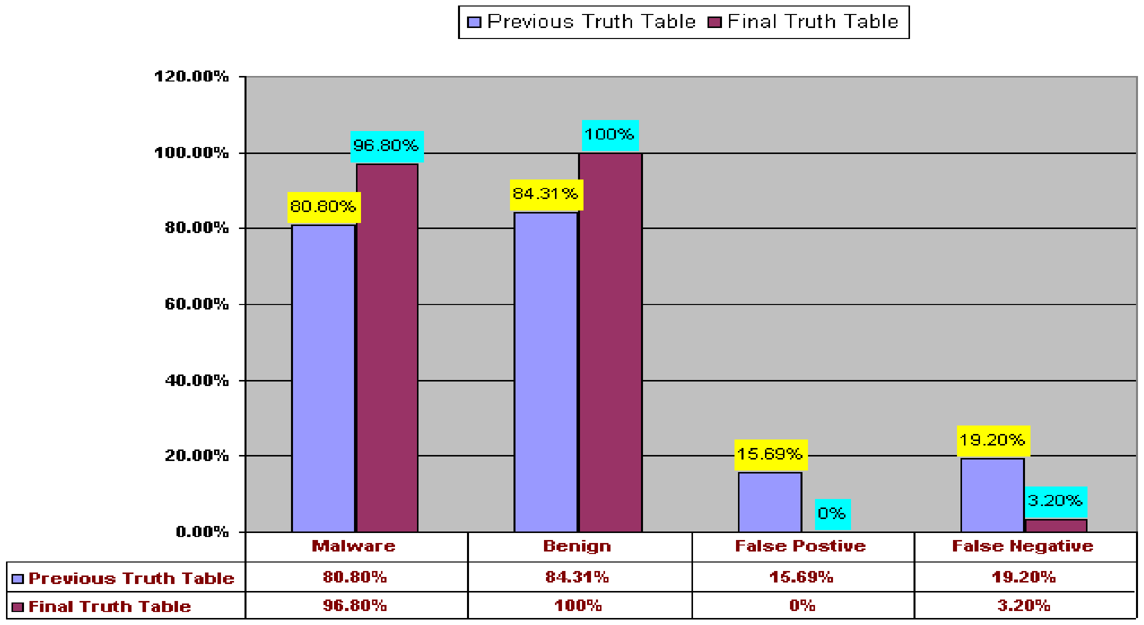
Based on the above results the final detection criteria were made which is explained in next section.

4.3 FINAL DETECTION CRITERIA:-

Approach 1	**Approach 2**	**Size<200K || Suspicious Timestamp** **Console based**	**Output**
B	B	**Yes**	**M else B**
B	M	**Yes**	**M else B**
B	X	**Yes**	**M else B**
M	B	**Yes**	**M else B**
M	M	**Yes**	**M else B**
M	X	**Yes**	**M else B**

Table 5- Final Truth Table

By applying the above decision criteria, the detection rate has increased amazingly as shown follows:-



	Malware	Benign	False Postive	False Negative
Previous Truth Table	80.80%	84.31%	15.69%	19.20%
Final Truth Table	96.80%	100%	0%	3.20%

Figure 26 – Final Result

Chapter 5

CONCLUSION

Our research aimed at detecting malware based on the features that are common in malware by analyzing and disassembling a software system in order understand its design, components, and inner-workings. We began our investigation by developing a program that dumps the structure of PE files which gives important implementation information of an executable without having a source file. Our research on malware and benign samples shows that reverse code engineering can be used to better analyze malware and provide us with better techniques to protect against them.

Malware Detection Rate = 96.8%

Benign Detection Rate = 100 %

False positive Rate = (0%)

Figure 27 – Final Detection Rate

Chapter 6

FUTURE RECOMMENDATIONS

Our Methodology leads to the investigation of another technique that has a great research potential. Our program gives the percentage of each Dll used in an executable by reading a binary code and following a decode sequence (that we were able to find out in our research) through which the sequence of API calls can be retrieved.

The sequence of API calls that we retrieved for a malware sample by decoding its binary is shown below.

```
malware\virus2\SVOHOST.exe,CloseHandle,CreateFileA,GetFileType,Get
SystemTime,GetFileSize,GetStdHandle,RaiseException,ReadFile,RtlUnw
ind,SetEndOfFile,SetFilePointer,UnhandledExceptionFilter,WriteFile
,CharNextA,ExitProcess,MessageBoxA,FreeLibrary,GetCommandLineA,Get
LastError,GetModuleFileNameA,GetStartupInfoA,RegCloseKey,RegOpenKe
yExA,RegQueryValueExA,GetCurrentThreadId,GetStartupInfoA,LocalAllo
c,LocalFree,VirtualAlloc,VirtualFree,InitializeCriticalSection,Ent
erCriticalSection,LeaveCriticalSection,DeleteCriticalSection,Local
Alloc,VirtualAlloc,VirtualFree,VirtualAlloc,VirtualAlloc,VirtualFr
ee,VirtualFree,VirtualAlloc,VirtualFree,InitializeCriticalSection,
EnterCriticalSection,LocalAlloc,LeaveCriticalSection,EnterCritical
Section,LocalFree,VirtualFree,LocalFree,LeaveCriticalSection,Delet
eCriticalSection,EnterCriticalSection,LeaveCriticalSection,EnterCr
iticalSection,LeaveCriticalSection,EnterCriticalSection,LeaveCriti
calSection,GetLastError,CharNextA,CharNextA,CharNextA,CharNextA,Ch
```

We could see that there can be a certain pattern in the sequence in which the API functions are called both in malware and benign samples .Based on these patterns that are distinct in malware a finite state machine can be developed to detect malicious softwares. A **finite state machine (FSM)** is a model of behavior composed of a finite number of states.

6.1 FINITE STATE MACHINE

e.g if we take the example of sequence of Api calls for accessing and modifying registry keys both for malware and benign samples we can develope a mehtolody to declare a file as malware or benign based on certain pattern as showm below

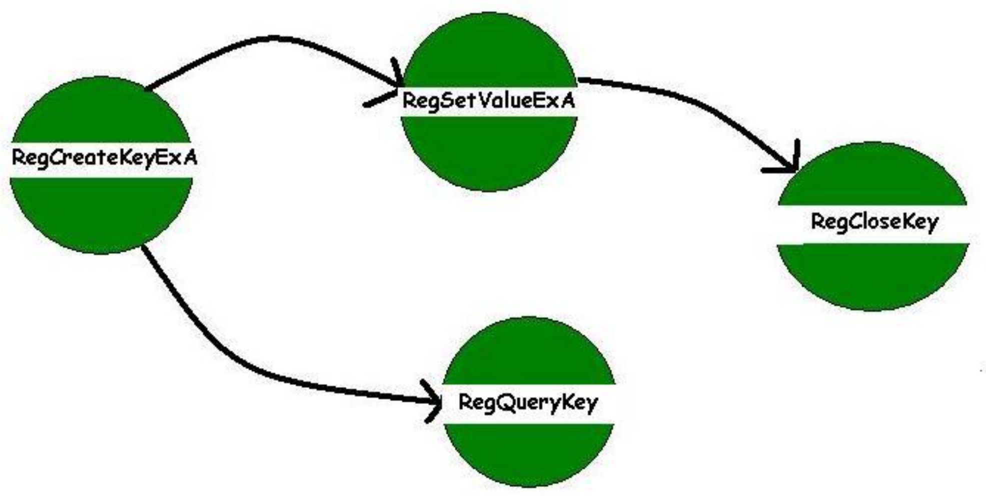


Figure 28 – Finite State Machine

REFERENCES

[1]- Steve R. White , (1998) , Open Problems in Computer Virus Research, Presented at Virus Bulletin Conference, Munich, Germany.

[2]-Explanation of Malware,December 2006 <http://en.wikipedia.org/wiki/Malware>

[3]- Ronald B. Standler, (2002), Examples of Malicious Computer Programs , Decemeber 2006 <http://www.rbs2.com/cvirus.htm>

[4]- William Stallings , (4th Edition), Cryptography and network security Principles and Practices , Prentice Hall, 680

[5]- IBM Anti-Virus Research , Jan 2007 <http://www.research.ibm.com/antivirus/SciPapers.htm>

[6]- Dmitry Gryaznov, (2006), Malware in Popular Networks, McAfee AVERT, Network Associates, Inc., Beaverton, OR 97006, USA

[7]- Arini Balakrishnan, Chloe Schulze , (December 19th, 2005), Code Obfuscation Literature Survey ,University of Wisconsin, Madison

[8]- VMware, Inc. “VMware Workstation FAQs.”,March 2007 < http://www.sol-tec.com/wkstnfaqs.asp>

[9]- Lenny Zeltser , (May 2001), Reverse Engineering Malware,Global information Assurance Certificate,USA <www.zeltser.com>

[10]- VirusList.com All about Internet Security, Jan 2007 http://www.viruslist.com/en/trends

[11]- Konstantin Rozinov Bell Labs, Reverse code engineering: an in depth analysis of the bagle virus, government communication laboratory, internet research, systems and software group

[12]- The Portable Executable File Format, March 2007 <http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile.html>

[13]- OllyDbg , March 2007 < http://www.ollydbg.de/ >

[14]-Download PE Explorer, March 2007

< http://3d2f.com/programs/11-286-pe-explorer-download.shtml>

[15]- UltraEdit A Text Editor , April 2007

<http://en.wikipedia.org/wiki/UltraEdit>