PROTECTING ANTI-VIRUS SOFTWARE UNDER VIRAL ATTACKS

by

Raghunathan Srinivasan

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

August 2007

PROTECTING ANTI-VIRUS SOFTWARE UNDER VIRAL ATTACKS

by

Raghunathan Srinivasan

has been approved

July 2007

Graduate Supervisory Committee:

Partha Dasgupta, Chair
Charles Colbourn
Aviral Shrivastava

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

Computers are an important part of everyday life to many people across the world. Computers in the hands of consumers who lack the knowledge of protection tools and who have limited administrator skills are vulnerable to virus attacks. These systems are extremely valuable to intruders as they have lot of secret personal information about the users. Attackers exploit vulnerabilities in the software layers to install malicious programs on user machines to steal secret data for financial gains.

Security protocols have been in place for some time to counter the threat posed by the attacks. However, despite the presence of such measures, the number of attacks on consumer computers is growing rapidly. A recent trend in attacks has been the attempt to disable security protocols in place at the host machine. This type of attack leaves the host computer completely defenseless and vulnerable to many further exploits through the Internet.

To ensure the continuous functioning of the security protocols, a software-based solution is proposed in this thesis. The solution involves camouflaging the security processes to avoid being detected and disabled by malicious programs. To protect the program in the memory from being tampered or altered with, various modules are employed in this solution. The modules provide obscurity, diversity, randomization and migration of code to hide the location and presence of the security processes.

In memory of my parents

who helped me fight my viruses

ACKNOWLEDGMENTS

I have had the chance to spend two years at Arizona State University pursuing my MS degree. Now that I look back to see how far I have come since completing my undergraduate studies in India, I feel I have learnt so many things towards my career and life, and yet, I feel that there is so much more to learn. I have finally reached the point where I wanted to be two years back. As I stand here hours from the fulfillment of my dreams, I would like to take time and thank the individuals who helped me along my way.

I would like to express my gratitude to Dr. Partha Dasgupta for giving me an opportunity to work on this novel topic and for providing me with valuable guidance, encouragement and support. I remain highly obliged to Dr. Charles Colbourn and Dr. Aviral Shrivastava for the useful ideas and feedback they gave as part of my thesis committee. I would also like to thank Dr. Dijiang Huang for agreeing to attend my thesis defense. I would like to thank my friends Lifu Wang and Satyajayant Mishra for helping me with programs and ideas.

It has been a wonderful experience working with all my colleagues in the Distributed Operating Systems group at ASU and I would like to thank them for helping and making the whole process of research so much fun. I would like to thank all my friends and roommates for making it all worthwhile. Finally, I am ever grateful to my family, especially my sister and my uncle for their unconditional support, inspiration and love I received from them.

TABLE OF CONTENTS

Page

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTER 1 INTRODUCTION 1

1.1. Security in Consumer Computing 2

1.2. Malware Threats 4

1.3. Background on Malware 5

1.3.1. Types of Virus 5

1.3.2. Other Malware 8

1.4. Types of Attacks 9

1.4.1. Social Engineering 9

1.4.2. Exploit on Software Vulnerabilities 10

1.4.3. Phishing 11

1.4.4. Pharming 11

1.5. Anti-Virus 12

1.5.1. Signature detection or Pattern Matching 13

1.5.2. X - Raying 13

1.5.3. Emulation 13

1.5.4. Frequency Analysis 13

1.5.5. Heuristics 14

Page

1.6. Recent trend in malware . . . 14
1.6.1. SpamThru Trojan . . . 14
1.6.2. Beast Trojan . . . 15
1.6.3. Win32.Glieder.AF . . . 16
1.6.4. Winevar . . . 16
1.7. Organization . . . 18
CHAPTER 2 RELATED WORK . . . 20
2.1. Secret data protection . . . 20
2.1.1. Smart cards . . . 20
2.1.2. HD-DVD encryption . . . 22
2.1.3. Distributed software for secret protection . . . 24
2.1.4. Software based approach for secret management . . . 25
2.2. Code Hiding . . . 26
2.2.1. Code Obfuscation . . . 26
2.2.2. Code hiding by malicious programs . . . 27
2.3. Code Injection . . . 31
CHAPTER 3 MOTIVATION AND THREAT MODEL . . . 33
3.1. Motivation . . . 33
3.2. Threat Model . . . 34
3.2.1. Internet Threat Model . . . 35

Page

3.2.2. Shortcomings of ITM 36

3.2.3. Viral Threat Model 36

3.2.4. Threat Model used in this research 37

3.3. Types of attacks 38

CHAPTER 4 BUILDING BLOCKS 40

4.1. Injecting Code in Logon process 41

4.1.1. Shortcomings 41

4.2. Watch Processes 42

4.2.1. Shortcomings 43

4.3. Install as a Different Process 44

4.3.1. Shortcomings 45

4.4. Summary 46

CHAPTER 5 DESIGN 48

5.1. Installing the Program 48

5.2. Starting the Process 49

5.3. Execution of the Process 51

5.4. Watch Processes 52

5.4.1. Shut Down Events 53

5.5. Virus Definition Files 54

5.5.1. Whitelists 54

Page

5.5.2. Storage of definition files 55

5.6. Summary 57

CHAPTER 6 PROTOTYPE 59

6.1. Placing start up information in Windows system process 59

6.2. Code Injection 60

6.2.1. Injecting Libraries 60

6.2.2. Injecting Code 61

6.3. Watch Processes 62

6.4. Overhead and Performance 62

CHAPTER 7 CONCLUSION AND FUTURE WORK 64

REFERENCES 65

LIST OF TABLES

Table — Page

1. System Overhead 63

LIST OF FIGURES

Figure Page

1. Working of a Cryptographic Smart Card 21

2. Software Based Solution for Secret Storage 25

3. Protection Rings in the Pentium Architecture 28

4. Working Mechanism of Shadow Walker 29

5. System before infection 30

6. System after infection 31

7. Internet Threat Model 35

8. Threats posed by malware 37

9. 3 Watch processes to monitor the anti-virus 43

10. Query results before camouflaging the anti-virus software 44

11. Query results after camouflaging the anti-virus software 45

12. Starting the anti-virus 50

13. Starting and migration of anti-virus program 53

14. Receiving and storing image files 56

15. Splitting and placing of definition files 57

CHAPTER 1

INTRODUCTION

Internet is a collection of interconnected computers that are accessible over a protocol like the Internet Protocol. Today's Internet has evolved from the United States Defense Project ARPANET [65, 30]. Internet has revolutionized the way people go about their everyday routines. People rely on the Internet to communicate, share files, for news, and most importantly for financial transactions. Internet apart from bringing in comfort and convenience has brought an ugly side of computers with it, a plethora of Malware. Recent studies and researches show that a computer connected to the Internet may experience an attack every 39 seconds [31]. The home computers are the most vulnerable to attacks by malicious programs and hackers, because most home users do not have the required skills to use tools to prevent or counter an infection. Even if the user is skilled at detecting intrusions, a well designed virus that does not reveal itself in the process table entry or hides the files it has infected can evade detection. This is made possible by the malicious program patching various operating system services and routines. Security software rely on system calls and the view of the system it receives to detect malicious code. If the system calls are patched to return fake values, and the system appears perfect to the anti-virus, the virus can become very difficult to detect.

A virus does not rely on a single method of infecting computers. Methodologies used for attacking the system vary from exploiting vulnerabilities of the operating system/software, social engineering, buffer overflow, spam, code injection. New vulnerabilities in the system are discovered every few days and are successfully exploited by hackers. These vulnerabilities are fixed by the software vendors who provide

patches and updates for the system. However, these updates may not be available to all users, due to reasons such as Internet connectivity. A virus may also disable updates from occurring on an infected machine. In such situations, it is very difficult to prevent every machine from being compromised by malware.

1.1. Security in Consumer Computing

Consumer computers are very attractive to intruders. These machines are fairly easy to gain access into with the aid of numerous online resources on hacking. Many home PC users are not aware of the fact that their machines have been compromised after an infection. This enables hacker to use the compromised machine to easily steal secret data such as passwords to bank accounts, credit card numbers and social security numbers. Furthermore an attacker can also install additional software that can enable the use of the compromised machine as some sort of central server to launch similar attacks on other machines. The compromised machines can also be made a part of a huge botnet that can be used to launch Denial of Service attacks on servers, or be used in an attempt to intrude the computers of government agencies. In such cases the blame for the attack ends up on the owner of the compromised machine and not the on the hacker who initiated the attack.

Software such as anti-virus solutions and firewalls offer some protection to users against attacks, however, they are not completely effective. Researchers have shown that there is no algorithm that can perfectly detect the presence of malicious code [39, 37]. The reason for this is that anti-virus relies on virus definitions and known behavioral patterns to identify malicious code. A code that is completely new in

design is bound to effectively use the *zero day exploit* [67]. That is the virus writers take advantage of the fact that all the machines around the world will be vulnerable to a security threat on the day the threat is created. This is because the anti-virus software will have no definitions describing the code as malicious in nature.

Any anti-virus software must perform three functions: *detection*, *identification* and *removal* of malicious code. The goal of any virus writer is to design a virus that can evade detection. Virus writers have employed a variety of techniques described in section 1.3.1 to achieve this. Till now, the absence of a *perfect virus* has allowed security software to detect the presence of malicious logic. A perfect virus would not provide the user or the security software any knowledge of its presence. This type of virus can evade detection from naive users as well as system administrators who are trained to detect any malicious activity. As mentioned above, anti-virus software use definitions to identify malicious code. The definitions are based on signatures or known behavioral patterns. The theoretical possibility of creating a virus that has a changing signature or behavioral pattern has been demonstrated in one research[76]. This is made possible due to the fact that all security software rely on detecting known patterns by means of *blacklists*. This problem can be solved by using a combination of blacklists and white-lists that contain a list of all known good programs in the system. This is discussed in detail in 5.5.

Anti-virus and other security software also suffer from other vulnerabilities. The anti-virus process can be killed by any process in the system with administrator privileges, or it can also be infected by viruses, due to which the virus detection

engine is rendered useless. A malicious program that obtains administrator privilege and manipulates other processes in the system is known as a *rootkit*. The rest of this chapter describes threats posed by malware, types of malware, recent trends in malware behavior and finally the problem statement of this research.

1.2. Malware Threats

Malware is a general term given to any malicious software that infects a computer without the knowledge of the user. Malware can be classified into Viruses, Spyware, Rootkits, Trojan horse, etc. Malicious programs infect a computer and then travel through the network connections to infect other computers. The Internet serves to increase the vulnerability as the infection spreads not only across a local network, but to any computer across the world [36].

Motivations behind malware attacks have been changing constantly over time. In the early days they were designed to cause disruptions. Early viruses attempted to delete files or wipe out hard disks. Recent malware are aimed at stealing secret information such as passwords, credit card numbers and social security numbers for providing some sort of financial gains for their developers. Apart from the change in motivation, malware writers have changed the method of creating the malicious code. A malware may contain a virus, a root kit and a password logger. The programs are written through software tools that are available on the internet. The code produced is very complex, made more difficult by the fact that most viruses generated nowadays are *polymorphic* [1] in nature. Malware are a very big threat in today's computing world.

1.3. Background on Malware

Computer viruses are programs that are deliberately designed to interfere with computer operation, record, corrupt, or delete data, or spread themselves to other computers and throughout the Internet [2]. One of the first well known malware was the Internet worm which was released in 1988 [69]. It broke into the system by means of flaws in the system software. It was not designed to do any damage, but was simply made to estimate the size of the internet. However, each computer started getting infected multiple times; the consequence was that the machines became very slow to the point that they could not be used.

Personal Computer Viruses first started appearing in the 1980's. In 1990 it was estimated that there were 500 virus programs. By 1996 the estimate rose to around 10000. It was estimated that by 2002 there were 60,000 viruses [58]. Today the number of known computer virus is estimated to be around 103,000 [3].

The first few viruses were simple machine language programs [61]. The viruses had the ability to infect other executable code. When a user would execute such code, the virus would take control of the system and infect more files. These viruses are known as parasites as they would not totally destroy the running machine. These viruses are easy to detect and remove, as administrators only had to search for patterns called virus signatures to identify infections. Viruses have become more and more sophisticated over the years. The following section describes the types of viruses and the methodology employed by virus writers.

1.3.1. Types of Virus.

1.3.1.1. *Boot Virus.* Boot sector viruses account for about 5 percent of known PC viruses [51]. These types of viruses operate by infecting the Master Boot Record (MBR) of a PC. The MBR is a program that resides on the first sector of a hard disk; it runs every time a PC starts up and is responsible for loading the rest of the Operating System. This makes a Boot sector virus extremely dangerous, although they are very few in number compared to viruses that infect files. Once loaded a boot sector virus resides in memory and is capable of infecting any drive placed on the system. They are very difficult to remove and it requires a bootable anti-virus disk to properly remove the virus. A few examples of boot sector viruses are 'Brain', 'Crazy Boot', 'Polyboot.B' and 'AntiEXE'.

1.3.1.2. *Parasitic viruses/ File Infectors.* This type of virus attaches itself onto files or executables, leaving the contents of the file unchanged. About 85 percent of all known viruses are of this type [51]. When a user runs the infected application or file, the virus code executes and copies itself into the memory. The code then attempts to spread itself onto other applications, and also any removable disks attached to the machine.

1.3.1.3. *Date viruses/ Logic Bombs/ Time Bomb.* These are types of viruses that reside in a machine and get triggered by some event such as a particular date or a day of the week [33]. This technique is very useful for a virus or worm to gain momentum and spread before being noticed. Examples of such viruses are: 'Michaelangelo', 'Sunday', and 'Century'.

1.3.1.4. *Macro Virus.* These are programs that take advantage of the macro utilities that are built into programs like Excel and Word. They are fairly easy to write and target documents that save macro code within the body of the document. The 'Concept' virus was the first macro virus written for Word 95.

1.3.1.5. *Encrypted Virus.* This is a type of virus whose body is encrypted. The virus itself contains the key for decryption and a decryption engine within itself. The encryption key varies from infection to infection causing the encrypted body to appear differently in every instance [60]. This methodology was used by virus writers to hide the virus from signature scanning techniques. 'Cascade' was one of the first viruses that used this methodology [66].

1.3.1.6. *Polymorphic virus.* A polymorphic virus contains a encrypted body and a decryption engine like the encrypted virus. In addition to this, it also has a mutation engine that creates new encryption schemes for every infection. If a user runs a program that contains the virus, the decryption engine first executes and places the virus body in memory. The virus then starts the mutation engine which generates an encryption - decryption routine for the next infection. The virus finally encrypts a copy of itself and the mutation engine using the new encryption engine and places itself in a new file. This creates a virus that does not have a fixed signature to scan for as no infections look the same. The first known polymorphic virus, *1260*, was written in the U.S. by Mark Washburn in 1990 [25].

1.3.1.7. *Stealth Virus.* This type of virus attempts to hide its presence by hooking onto some system calls. The first stealth virus was 'Brain' which attempted to

hook Interrupt 13 and certain other system calls that detect viruses in DOS. This is a trend that is prevalent even today. A recent worm called the 'Lion' installs a rootkit and then makes various hooks and system modifications to prevent any scanner from capturing its presence.

1.3.2. Other Malware.

1.3.2.1. *Trojan Horse.* This is a program that enters a machine disguised or embedded inside legitimate software. The Trojan looks harmless or something interesting to a user, but is actually very harmful when executed. Each Trojan has its own characteristic that is dependent on what the designer intended it to do. The Trojan depends on successful implementation of social engineering concepts as it has to fool a user into installing the code. It does not depend on security flaws or loopholes present in the system. Once inside a system, it can exploit any resource or use the machine as a zombie, or use the infected system as a launch pad for further attacks.

1.3.2.2. *Worms.* A worm is a self replicating program. Unlike a virus, it does not attach itself to any existing program. It uses the network resources to infect other machines in the network. Worms always harm the network whereas viruses always infect or corrupt files on a targeted computer.

1.3.2.3. *Rootkit.* Rootkit is term derived from the UNIX term root. It was designed to give administrator privileges to the attacker. A well written rootkit can hook on the Operating System's Page fault handler and Virtual Memory controller to conceal its presence, and that of its files [71]. In recent years root kits have been used increasingly by malware, helping an intruder to maintain access to a system whilst

avoiding detection. One of the most famous rootkit is the SONY's Digital Rights Management root kit. Sony intended to install a program on the consumer's PC that would not let the user rip any music files. This program hid files, registry keys and processes starting with the string sys. This led to many attackers use files starting with the above string.

1.4. Types of Attacks

As described earlier, a malware may choose one or a combination of several attack methods to compromise a system. Some of these techniques are discussed below

1.4.1. Social Engineering.

Social Engineering is a common method of attack by viruses and worms, especially those ones that spread by means of e-mail and messenger. It is used to manipulate people into performing certain actions or divulging information [24]. In most cases, the virus arrives in the e-mail of a user as part of a picture or an executable, which when viewed launches a back-door program or a Trojan. The *Happy99\Ska* worm was a malware that arrived at a user machine by e-mail and contained an attachment **Happy99.exe** [18]. When the attachment is executed, it starts a fireworks display that provides New Year wishes to the user. In the background, the program modifies the library WSOCK32.DLL; it also creates files SKA.EXE and SKA.DLL in the system directory and then creates an entry in the registry to start the executable. Other examples of viruses that employ social engineering are *PrettyPark*, *Anna Kournikova* and *Gibe*. The best way to prevent

social engineering attacks is not to trust e-mails or messenger texts coming from unknown sources. It is possible that a virus may use the e-mail id of known person to spread the attachment, however, these mails usually follow a set template in terms of the content, hence if a user learns how to recognize the template these attacks can be averted.

1.4.1.1. *Mass E-Mailers.* Mass E-Mailers are viruses that arrive by e-mail on a machine. If the user executes the attachment, the trojan executes in the background and obtains control over the machine. It then looks up at the address book of the user and sends itself out to everyone on the contacts list. Most mass mailers use social engineering tricks and concepts to trick users into opening the attachment. Examples of mass mailers are *ExploreZip*, *Love Letter*. By 2002, 90% of all known viruses were mass mailers [36].

1.4.2. Exploit on Software Vulnerabilities. Operating systems and system software contain many bugs or vulnerabilities that can be exploited to gain control of a machine. Exploit refers to a small section of code that can take advantage of the flaw present in the software. This code is generally reused in numerous trojans and viruses before the vulnerability is fixed by the software developers by means of a patch. Many exploits are designed to provide root access to a machine. It can be done by just one exploit, or by means of multiple exploit, each providing escalated level of privileges to the attacker. A single exploit takes advantage of a specific software vulnerability.

Exploits are categorized by the vulnerability that they exploit. A few types of exploits are:

- Buffer Overflow
- Heap Overflow
- Integer Overflow
- Code Injection
- SQL Injection
- Cross-site Scripting

It is very difficult to remove software vulnerabilities as it is difficult to foresee what can go wrong with a piece of code. Programmers can be taught to code in a fashion such that the exploits are minimized, this process is also known as *secure coding*, however, as seen in section **??**, it is a complex problem.

1.4.3. Phishing. Phishing is an activity used to steal information from users using social engineering techniques. This is usually done by masquerading as some trusted entity during electronic communication. Phishing is typically carried out by means of e-mail. It directs users to websites that look identical to that of the trusted entities, where an attempt is made to trick the user into revealing the password or some other secret. It is very difficult to prevent phishing attacks by means of monitoring software, the user has to be careful to avoid getting tricked.

1.4.4. Pharming. Pharming is an attack aimed at redirecting a website's traffic to another fake website. It can be achieved by either changing the *Hosts file* on a

computer or by exploiting some vulnerability in the DNS server software. Compromised servers are known as having been *poisoned*. Pharming is used to steal secret information. Countering Pharming is a difficult problem. Anti-Virus and similar software cannot provide protection against this threat.

Elimination of software vulnerabilities requires the implementation of secure OS and secure coding. Both the issues have been researched heavily but have been ineffective in practice, mainly due to the abundance of legacy code. The OS kernel consists of millions of lines of code, and writing a secure OS would require that the entire kernel is bug free. Writing bug free code is a very complex problem. Creation of a completely secure OS is unlikely [34]. The problem of preventing infections is made difficult by the fact that most hackers rely on human error to compromise systems. It can be inferred from above that it is hard to prevent an infection since it is difficult to foresee the exact error a user may commit. Hence, security software rely on detection instead of prevention.

1.5. Anti-Virus

The appearance of computer virus in the 1980's, caused the emergence of the anti-virus. Anti-virus is a software that scans the system either continuously, or at specified times for the presence of malicious entities. The anti-virus software has constantly evolved with the virus. The methodologies used by anti-virus software are as below:

1.5.1. Signature detection or Pattern Matching. Signature detection involves the anti-virus application scanning the computer for files that contain a code that it recognizes as the virus. The initial anti-virus programs would scan the entire executable file to find the presence of the virus code. Later the programmers found out that most of the virus infections involve virus placing its code on the entry point of the program. Hence the scanners started checking the entry point of the program; this methodology became obsolete when encrypted and polymorphic viruses emerged.

1.5.2. X - Raying. As encrypted viruses arrived, anti-virus software started using brute force decryption of the code. This was possible since they had to do known plaintext attack on the encrypted code; the plaintext was the known virus definitions. The virus writers used random encryption algorithms; hence the protocols were easy to crack. This type of scanning was known as X-raying [59].

1.5.3. Emulation. Another technique that emerged due to the appearance of polymorphic viruses was Emulation. This involved starting a CPU emulator and loading the executable file on it. The virus would not realize that it was being run on an emulation environment. This allowed the software to observe the behavior of the program in a closed environment where no damage would be done to the user machine.

1.5.4. Frequency Analysis. This was not a stand alone method. Frequency analysis involved scanning a code for the presence or absence of particular opcodes. Programmers found that most legitimate programs would use DOS interrupts like

21h in their code. This interrupt was hardly visible in malicious codes. Hence they would examine frequency of such opcodes, and programs having them would not be scanned.

1.5.5. Heuristics. Virus writers slowly started using techniques such as entry point obfuscations, by which they would not keep the virus code at the entry point of the executable. The emergence of viruses that could change their properties required a change in virus detection technology. Programmers came up with decision support systems, where a scan would have weight system. Points were allotted to behavior such as memory access, file access, page faults. After scanning, the number of points accumulated determines if a file would be flagged for a virus from a known set of behavioral trend.

1.6. Recent trend in malware

As anti-virus products made it tougher for the viruses to escape detection, the virus writers started a new trend. The malicious programs started disabling the anti-virus and other related security software in the system. The increasing number of such attacks in recent months shows how vulnerable the anti-virus software is to such attacks. Few such Trojans are discussed below.

1.6.1. SpamThru Trojan. The SpamThru Trojan [4] demonstrates all sorts of loopholes that can be exploited in our current security architecture. It gets installed on a machine by some means of social engineering, then it patches the running anti-virus by blocking updates by changing the anti-virus update sites to point to the

localhost address. After this it downloads and installs a pirated copy of Kaspersky Anti-Virus into a concealed directory on the infected system. Once this is done, it patches the license signature check file of the anti-virus so that the user does not receive any notification of the expired license. It then runs the anti-virus to find the presence of any other malware and removes them - this is done so that there is no competitor for system resources. It conceals its own files from the scanner. It also installs a P2P engine to keep track of IP addresses of all infected machines. It runs a control server from any of the infected system, and transfers control if the control server is shut down.

1.6.2. Beast Trojan. Beast [17, 16] is a backdoor Trojan horse. It works as a RAT (Remote Administration Tool). This was first found in 2002. Beast employs client/server architecture. This was one of the first Trojan to establish a reverse connection in client/server architecture. The server does not require the IP address of the client, it would just connect to a DNS that would in turn connect to the client. The first version of the Trojan injected its DLL's into explorer.exe. The latest version of the Trojan injects its DLL's into winlogon.exe, a process in Windows 2000 and XP that is always active. Winlogon.exe also handles the logon process for windows. Its ability to inject its DLL's in a critical system process demonstrates the loopholes present in the current system. Once it infects a system it shuts off the anti-virus and Firewall and the attacker essentially obtains complete control of the system.

1.6.3. Win32.Glieder.AF. Win32.Glieder.AF [5] is a Trojan that downloads and executes files from a list of URL's. This Trojan is spammed to users by another Trojan - Win32.Bagle.BG. The Trojan arrives in a mail having a zip attachment. When the executable gets launched, the Trojan copies itself to %System%\winshost.exe
It then adds winshost.exe to the startup process by adding the registry item:

- HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\winshost.exe= %System%\winshost.exe

After doing this Glieder.AF deletes registry items if found, belonging to security software such as Firewall, anti-virus, etc.

1.6.4. Winevar. Winevar [77] is a Windows Portable Executable of size about 91 kb. When the file is executed it copies itself as WINxxxx.PIF (the last 4 x denote random characters) to windows system directory. It also creates a startup entry for itself as below:

- HKEY_Current_User\Software\Microsoft\Windows\CurrentVersion\Run
- HKEY_Local_Machine\Software\Microsoft\Windows\CurrentVersion\Run
- HKEY_Local_Machine\Software\Microsoft\Windows\CurrentVersion\ RunServices

The worm creates subkeys under the Run key with names that correspond to the name of the worm in the system. The virus remains continuously active in the system and terminates processes and services that contain the following text:

- view, debu, scan, mon, vir, iom, ice, anti, fir, prot, secu, dbg, avk, pcc, spy

The worm then scans through the hard disk(s) to locate any folders that contain any file with the following names, if any such folders are found then the virus deletes all the files in the folders.

- Antivirus, cillin, nlab, vacc

The list is not limited to only these viruses, Klez, Bugbear, Fizzer and Lirva are examples of other viruses that attempt to disable anti-virus programs. This process is known as *Armoring* [36]. Armoring marks a significant change in the tug of war between the viruses and the anti-virus software. The methodologies used by anti-virus companies have undergone changes, till now any infection could be contained and cleaned by the anti-virus after the arrival of the update. However, the latest trend of killing the anti-virus process threatens to make their presence inconsequential, as there would not be any further updates. This means that there is an urgent necessity to address the issue and protect anti-virus and security software from such rogue programs.

This thesis presents a software based solution to prevent malware from disabling security software. This problem is similar to that of preventing infections and also similar to the problems faced by virus writers in hiding their programs from the anti-virus. As seen earlier, preventing infection is hard due to the numerous approaches used by attackers to exploit software. Hiding programs inside a system from the anti-virus is also a difficult problem. Some approaches are seen in 2.2.2. However,

any vulnerability or technique used by attackers to gain control over machines becomes obsolete soon as system administrators and software developers provide patches to fix the bug or shortcoming in the code. Due to this reason, many hackers sometimes do not publish the details of how the exploit was carried out.

It is not possible to provide a fool-proof solution that will hide the anti-virus from a malware completely; however, this solution makes it much more difficult for malicious programs to locate the anti-virus program and kill the process by hiding the following:

- process table information
- directory structure
- files used
- registry information

1.7. Organization

The rest of this thesis is organized as follows:

Chapter 2 outlines the work done in the field of data and secret protection by means of smart cards, and software based solutions. It also outlines related work done in the field of rootkit detection and prevention by means of software, trusted computing hardware and VMM.

Chapter 3 presents motivation and threat model.

Chapter 4 presents software based approaches for hiding process information inside the Operating Systems and discusses their shortcomings.

Chapter 5 presents the design and framework for the solution to the problem statement.

Chapter 6 summarizes this thesis.

CHAPTER 2

RELATED WORK

Hiding information is a technique used both for benevolent and malicious purposes. The benevolent purposes are to hide secret information like passwords, credit card information, etc. The malicious purposes are typically to hide the presence of programs such as Trojans and Viruses from the users and system administrator. This chapter discusses the methods used for hiding code, data, and process information.

2.1. Secret data protection

2.1.1. Smart cards. Protection of secret data was first proposed around 20 years ago by means of a hardware device [35]. Since then many applications and architectures have been proposed to protect data using *smart cards* or a secure hardware device that is tamper resistant. Smart card is a hardware device that is embedded with a microprocessor capable of performing certain limited operations. It is also embedded with a memory chip that can store some data. The main kinds of smart cards are described below. The first is a memory card; it contains a memory chip that is updated by an external device. This type of card is used in the *store cash* card employed by pay phones, laundry utilities, gift cards, etc. The second is a microprocessor card that contains a processor, memory chip and a mini operating system that can process and store data. Example of this type of card is a cryptographic coprocessor card that is used for secure transactions. The general working of a cryptographic smart card is shown in Fig. 1. The card contains the private key or the secret identifier of the person who owns the card. It also contains a signature algorithm (RSA) by which

the card encrypts the incoming messages. The messages may be hash of a document, or a challenge response sequence.

A coprocessor card has to be designed carefully so that it does not expose its secrets under attacks. For example if the card contains the private key of a consumer, it should never reveal the key. If any value is to be encrypted (signed) the card should receive the value and return the encrypted value. If the design for the co processor is not carefully thought through, there is a good chance of the stored data being misused or *phished*.

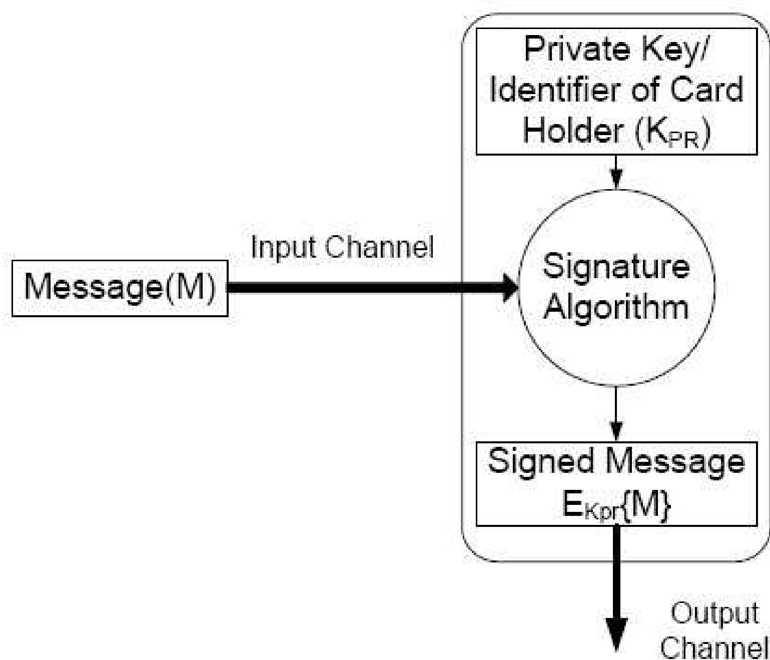

FIGURE 1. Working of a Cryptographic Smart Card

2.1.1.1. *Common Access Card.* The common access card [57] issued by US department of defense is a smart card used for identification and authentication. It allows users to access controlled facilities, sign electronic documents, and encrypt information. The card has a 32 bit RISC processor that can provide on board com-

puting for challenge response. The card can also store digital certificates. The card generates its own public - private key pair. It can perform encryption operations without revealing its key to the host machine. The card is connected to a host machine using a USB connector. The reader to card interface uses an industry standard smartcard connector. To prevent identity thefts, the card is protected by means of one or more PIN(s). The card is currently deployed heavily among government and military personnel; it is not for use by consumer computers.

2.1.1.2. *IBM's PCI Crypto Card.* The IBM's PCI crypto card [49] is a programmable PCI board with specialized electronics required for providing a secure system where data processing and cryptography can be performed. It provides data protection by PKCS #11 library or extended IBM's Common Cryptography Architecture (CCA). The card is used by banks for financial applications and identification purposes but is not used in Personal Computers.

2.1.1.3. *SET.* The Secure Electronic Transaction (SET) [72] is a protocol that deploys the use of software and hardware. It was developed by Microsoft, Visa and others for performing secure transactions over the internet. SET requires every client to install a special hardware and software. Due to this reason, it failed to win any significant market share. It also does not have any means of guaranteeing secure transactions if the host computer is compromised or infected with malicious code.

2.1.2. HD-DVD encryption. HD-DVD (High-Definition DVD) is a high density optical disc designed for storage of data and high definition video. It is designed to be the successor of the DVD format. The data stored inside the disc is encrypted by

means of AACS (Advanced Access Content System), which uses the AES (Advanced Encryption Standard) as the algorithm for encryption. The difference in AACS and CSS (Content Scrambling System) is in the organization of the decryption keys. In CSS, the content of the DVD is encrypted by means of a content key. The content key is then encrypted by means of numerous other keys and each player (software or hardware) is given a key for decrypting the content key. Due to inclusion of so many encrypted values of the same key it was possible to compromise the keys of a player to illegally decrypt the content.

Under AACS the decryption key is derived from a combination of a media key and other elements such as a physical serial number on a DVD and the cryptographic hash of the title's usage rules. To view a movie, the player has to decrypt the content on the hard disk. The disk contains 4 items:

- MKB (Master Key Block)
- Volume ID
- Encrypted Key
- Encrypted Content

The MKB is different for every disc. This is combined with the device key of a particular player, to generate Km (Master Key). Km is then combined with the Volume ID (which the program obtains by presenting a cryptographic certificate to the drive) in a one-way encryption scheme (AES) to generate Kvu (Volume Unique

Key). Kvu is used to decrypt the encrypted title key, and that is used to decrypt the encrypted content [50].

However, AACS has been broken by crackers [21] despite AES being employed to protect the decryption keys. This was achieved by examining the contents of the physical memory for the presence of the decryption keys. This attack uses the principal that encrypted data has to be decrypted before it can be processed by the display device. This attack demonstrates that hiding information is a very difficult problem and that any complex solution can be broken down by means of memory analysis.

2.1.3. Distributed software for secret protection. Distributed software approaches protect secret information using remote servers. In [45, 46, 48, 62] one or more remote credential servers are used to construct and distribute secrets. N pieces of a secret are distributed into N servers in a way such that the recovery of data is possible only in the presence of 't' trusted servers. These protocols stop outsider attacks by exchanging hidden information between clients and servers, and insider attacks by using multiple servers, each of which holds a share of the secret, hence the secret cannot be compromised by an attacker unless enough servers are attacked simultaneously.

Splitting a secret into multiple shares is not very useful in consumer computing. The first reason is that it is not very practical to set up distributed storage for home users. Secondly, the protocols assume that data can be compromised but security algorithm cannot be altered. This means that a malicious code can attack the system

by compromising the client side security software instead of attacking any server nodes. Many rootkits are capable of performing such binary modifications.

2.1.4. Software based approach for secret management. The issue of software based secret management is addressed by means of two components [75]. It consists of a front end that interacts with software running on the system, and a back end that hides the secrets. The front end provides the API's for encrypting secrets such as passwords, keys.

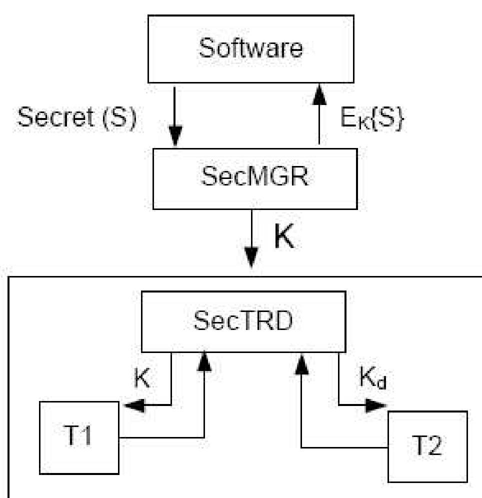

FIGURE 2. Software Based Solution for Secret Storage

The back end protects the key in memory by generating a number of worker threads. All but one of the threads receive fake keys. The back end stores the hash of the real and fake keys in its memory. The worker threads can choose to exchange their keys in memory. By means of a few other protocols, the system ensures that even if a key gets stolen from the worker threads, the incident is notified.

When the software requires the secret, it sends the encrypted secret to the front

end. The front end then requests the key from the back end. The worker threads are then requested to return the values stored by them. The real key is identified from the fake ones by means of comparing the previously stored hash values. Fig. 2 shows a 2 worker thread model for the solution.

A conceptually similar approach to this is implemented in this research work for protecting anti-virus software from malicious entities. In this thesis instead of creating multiple threads, multiple processes are created as one of the steps to protect the anti-virus software.

2.2. Code Hiding

2.2.1. Code Obfuscation.

Obfuscation is the process of transforming code such that it becomes hard to understand and read. Obfuscation may involve source code obfuscation and/or object code obfuscation, the latter is also referred to as code morphing. There are several methods and automated tools available to transform code [40, 43, 42, 41]. These techniques use language based and tools supported by the compiler to transform code into another code that is functionally equivalent to the original code. The methods hide program logic and code from attackers so that it is harder to reverse engineer the program. There are many other techniques present that perform data, control and layout obfuscation [38, 74, 55]. Another technique used to perform obfuscation is self modifying code [54]. This involves replacing the actual code with a template that contains some references to the program and in addition also contains random code. During runtime the code is generated on the fly

by means of the template and a stub that contains information about which portions of code to actually generate and which portions are to be omitted.

These transformations, however, are not necessarily fool proof in protecting code and program information [73, 78]. There are tools available for disassembly of programs. Code obfuscation can be broken by using static disassembly techniques [68]. These involve scanning the object code to construct the program control flow. Object code obfuscations can defeat techniques that include analysis of execution pattern with static disassembly techniques [53]. A study also showed that there are programs that cannot be transformed into an obfuscated form, hence proving that there is no general obfuscation methodology [32]. Another study showed that De obfuscation is NP - easy, proving that it is impossible to come up with a transformation that is guaranteed to hide program code functionality and control flow [19].

However, despite theoretical findings, obfuscation is a very popular method commercially. Software industry, DRM vendors, and hackers (Virus/Trojan developers) find obfuscation a convenient tool, even though it is not guaranteed to be perfect.

2.2.2. Code hiding by malicious programs. Control is a major goal of malware writers. To achieve this, they must be able to monitor and intercept the state and actions of a compromised system. This requires that the malware should be constantly active in the system while remaining unseen. Rootkits are a popular tool used by hackers to hide the presence of malicious code.

Fig. 3 [56] shows the protection rings in the Intel pentium architecture. The

mechanism provides four levels of privilege. Ring 0, the innermost ring, denotes the highest level of privilege, while ring 3, the outermost ring is the lowest level of privilege. The aim of smart rootkit writers is to obtain access to the inner rings if possible. This is because lower the ring, the tougher it is to detect a root kit. Three such rootkits are discussed in this section.

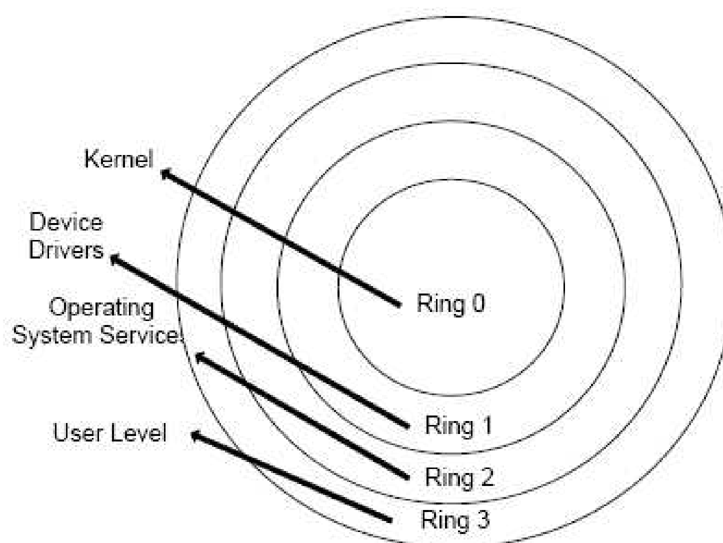

FIGURE 3. Protection Rings in the Pentium Architecture

2.2.2.1. *Shadow Walker.* Shadow Walker [70] is not exactly a rootkit, but it is a hooking technique that can be used by other malicious programs to hide their presence in the system. It is designed to deceive in memory signature scanners. It takes advantage of the Windows architecture where the mapping between the virtual memory used by a process and the actual physical memory is done by means of a Page table entry and a hardware device called Translation Lookaside Buffer (TLB). There are two buffers in hardware, the Instruction TLB (ITLB) and the Data TLB (DTLB). Both the buffers are normally synchronous; this rootkit makes them asynchronous,

i.e., makes the buffers hold addresses of different locations for the same entry. Memory scanners rely on the integrity of the view of memory they have. Therefore, Shadow Walker attempts to control a scanner's memory reads. To do this, the rootkit hooks onto the Page Fault Handler and the Page table entry. This enables the rootkit to have knowledge about all read/write or execute accesses on memory. An access where the program counter is the same as the address being accessed is an execute access, if the two are different, then the access is a read/write access. After doing this, the rootkit writes its code in certain section of the memory and marks it as not present in the page table entry. It then flushes the TLB so that no access can occur to the infected page except through the hooked Page Fault Handler. When a scanner attempts to read a certain section in memory, a fault is generated, which causes the page fault handler to fetch the page into memory and fill the contents of the DTLB with the address of the fetched page. The hooked page fault handler does not give access to the requested page, but gives access to a page that is a fake version of the malicious page. This is shown in Fig. 4. The scanner never sees the malicious code, hence does not detect any infection.

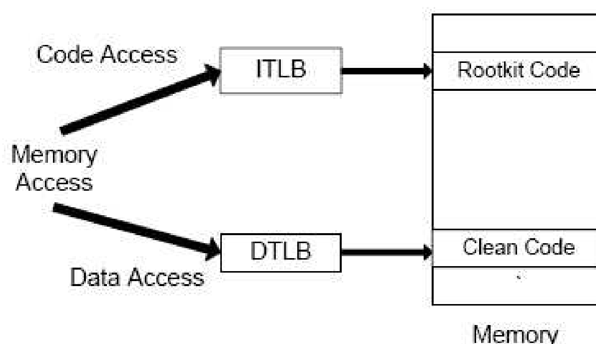

FIGURE 4. Working Mechanism of Shadow Walker

2.2.2.2. *SubVirt.* SubVirt [52] is a Virtual Machine based rootkit. This method takes advantage of the fact that in a system the lower layers can effectively control the upper layers. In a system, the hardware is at the lowest level, which is followed by the operating system, and then the applications. Fig. 5 shows a target system that is in a clean state. SubVirt uses Virtual Machine technology and installs itself between the Operating System and the Hardware. The attacker first gains access to the system with sufficient privileges in order to change the boot sequence and to install applications, this is typically done in *Root* mode in Linux/Unix and *Administrator* mode in Windows. Once sufficient privileges are obtained, the attacker installs the virtual machine based rootkit (VMBR) in the system inside the persistent storage. The attacker then proceeds to modify the boot sequence to ensure that the VMBR loads before the host OS. Once the system reboots, the machine wakes up inside the *hypervisor* [6] and all accesses made by the host OS are through the VMBR. Fig. 6 shows the machine after infection. This type of rootkit cannot be detected by processes running within the system, but requires either a Virtual Machine based detection scheme running below the VMBR or a hardware based mechanism.

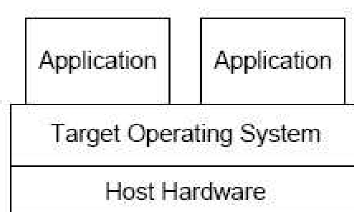

FIGURE 5. System before infection

2.2.2.3. *Blue Pill.* Blue Pill is also a Virtual Machine based rootkit designed for Windows Vista OS running on AMD Athlon 64 Processors. Windows Vista does

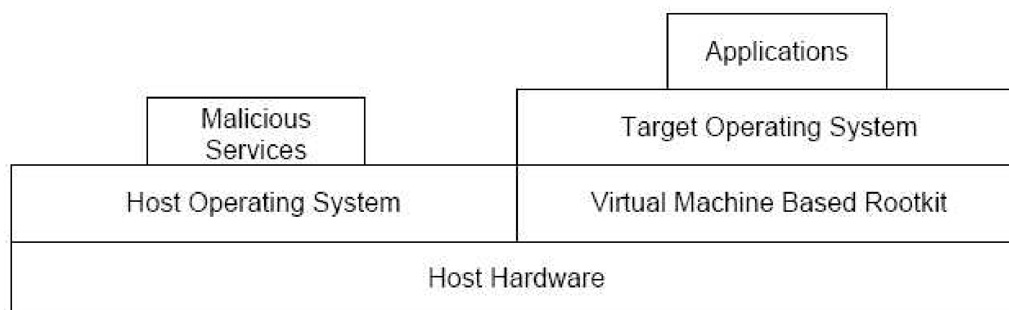

FIGURE 6. System after infection

not give kernel level access to many applications to prevent malware from obtaining escalated privileges, however, it allows user mode applications to get raw write access to the disk drive [64]. This rootkit initially executes code in the user mode and allocates lot of virtual memory for its process. Eventually when the memory capacity is overloaded, the kernel starts paging out the memory and code sections to the disk. The rootkit then identifies a driver to be written on and writes binary code on the pages out code. The code then calls a Windows API to load the driver back in memory. The written code then executes.

After this the injected code turns on the hardware virtualization feature [44], and forces the Vista Operating System to go into the guest environment. This is done in a manner such that the transition is not known to the host OS. Once this is done, complete control over the machine is achieved. The rootkit does not survive system reboot.

2.3. Code Injection

Code Injection is a technique to introduce code into a process from an outside source during execution. Code injection may be used for malicious or benevolent

purposes. Typically code injection is seen as a malicious action. These techniques are very popular in system hacking or cracking. They are also used to gain privileged access to a system. Examples of methods the attackers use are injection vectors [27]. Code Injection is not limited to any particular language, injection methods are present for the C language, web CGI scripts, SQL and unix command line [28, 29]. A benevolent use of code injection is a user changing the behavior of a program or the system to meet the certain requirements. This is done when the functionalities of the software have to be altered and the cost of modifying the software is very costly or a cumbersome process, and it is cheaper and convenient to inject some code in the program to achieve the desired functionality.

Code Injection is a debatable topic which has certain benevolent uses, but is typically used for hacking purposes. In this thesis, code injection is used as the predominant means of hiding the presence of the anti-virus process in the system.

CHAPTER 3

MOTIVATION AND THREAT MODEL

3.1. Motivation

The financial impact of malware has increased from 500 million to 14.2 billion dollars between 1995 and 2005 [23]. This can be attributed to the increase in the usage of Personal Computers and the growing popularity of the internet. The widespread usage of Internet has directly contributed to the extensive and rapid spreading of malware. Personal Computers are the most vulnerable point for the hackers as many home PC users lack knowledge of how to protect themselves against malware. The consumers are also careless about the data they store on their machines, hence hacking them is easier for the attackers compared to hacking corporate servers and networks, even though the fiscal benefit gained from hacking a corporate server may be far greater.

There is no perfect recipe for detecting malicious code as discussed earlier. Hence the problem of determining malicious logic is *turing undecidable.* Malware are getting stronger and sneakier with high penetrative power. Rootkits are very effective in hiding their presence in a system. Each malware is different in essence (genetic diversity), and hence their detection requires that anti-virus products use definition files. All systems remain vulnerable to a new virus until the definitions arrive. However, recent trends in malware attacks show that many trojans attempt to switch off anti-virus and other security programs. This brings up an important issue that has not been addressed till now, and that is, how to protect the anti-virus from such attacks?

The anti-virus solutions in most user machines are identical (genetic unifor-

mity). Due to this, an attacker can use one machine for carrying out experiments to find out ways to exploit vulnerabilities, and use the information to carry out an attack on other machines. By making programs dissimilar on every machine the complexity and cost of an attack can be increased.

Some tricks from the virus world can be used to achieve this goal, however the use of rootkits has to be avoided. The reason for this is obvious: if a rootkit that is used to hide the presence of a software gets infected by a trojan, the entire system would collapse. Instead techniques like polymorphism and code injection can be used to achieve this goal. Even though their usage is frowned upon as malicious, they can serve as effective tools to hide the presence of the anti-virus software on a system. Polymorphic viruses are very difficult to detect as they keep on changing form. If the anti-virus software can change its process name, file name, directory name, executable, registry entries, code and checksum from time to time, it will be very difficult for virus writers to write a program to disable the anti-virus.

This research aims to create a process that can evade detection by such programs, but still stay visible to the system administrator and the user. The latter requirement is important as it gives a user the option of terminating the process if necessary. This is also necessary because in the eventuality that a virus patches onto to the hidden anti-virus, the user can uninstall the software.

3.2. Threat Model

There is no perfect solution to solve the computer security issues. Every solution has its limitations and is made keeping in mind the threats it must address.

In this section two conventional threat models are described followed by the threat model used in this thesis.

3.2.1. Internet Threat Model. The Internet Threat Model (ITM) is defined for client server architecture over the Internet. It describes the resources that an attacker can possess and the attacks that can be mounted to compromise a system. The Internet Threat Model assumes that end systems (client and server machines) are secure while the communication network is not, this is illustrated in Fig. 7. An attacker can have complete control over the communication channel between two machines. An attacker can also inject packets in the network and set their destination to any machine over the network. The attacker can read any packet and modify its contents. It also assumes that websites are not to be trusted and that passwords are not safe on servers [63].

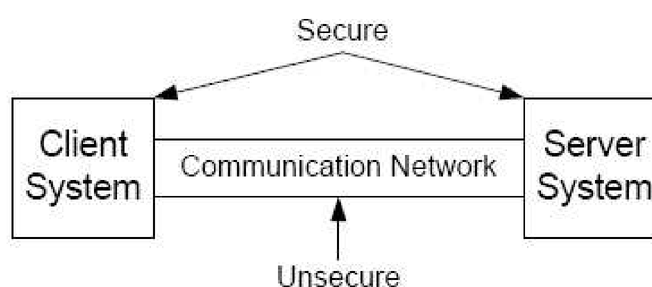

FIGURE 7. Internet Threat Model

To counter the Internet Threat Model several network security protocols have been implemented in different network stacks. These are SSL (Secure Sockets Layer), IPSec (IP Security), VPN(Virtual Private Network), TLS (Transport Layer Security) and HTTPS (Secure HTTP).

3.2.2. Shortcomings of ITM. ITM assumes that end systems are secure and the communication network is vulnerable to attacks. This type of threat model works in developing network protocols, but will hardly help in developing a protocol for developing security protocols within a PC. This is because the end systems are not secure, and the communication channel is safer than assumed [47].

On host machines, every activity is software driven. Many end systems are infected with trojans, spyware, rootkits, etc. No amount of source level verification or scrutiny can protect the machines from untrusted code. Vulnerabilities exist in the entire system as seen earlier, if the bug is in the lower layers of the system, it becomes more and more difficult to discover the vulnerability. There is a stricter threat model called the *Thompson Threat Model*. Under this threat model, no software is to be trusted. This model is highly strict, and is very impractical since it is not possible for anyone to design an entire software system from operating system, assembler, compiler, microprocessors and other system components.

3.2.3. Viral Threat Model. The viral threat model assumes that there are malicious programs in a host system. A malicious program potentially has the capacity to do anything that it wishes in the system as represented in Fig. 8. The threat model is constructed around the second type of virus attack mentioned in the preceding section. The virus will not allow anti-virus programs to function correctly, but it will allow applications to connect to the internet. It will also allow most of the system applications to run unhindered. This threat model also assumes that there are some trusted software in the system. These are the software that are provided by

the manufacturers.

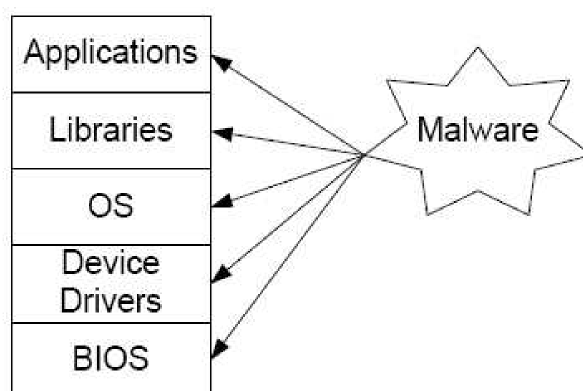

FIGURE 8. Threats posed by malware

3.2.4. Threat Model used in this research. This research aims at protecting an anti-virus from getting disabled or shut down by an attacking virus. A few assumptions have been made while tackling the problem of malware shutting or incapacitating the anti-virus. The assumptions made in this thesis were:

1. The anti-virus will get installed on a clean machine, but a virus may attempt to infect software on the system at a later date

2. The virus will not attempt to shut down all the processes

3. The virus will not attempt to delete all files

4. The virus will allow applications to upgrade to newer versions

The assumptions have been made for the following reasons. If there is already the presence of a virus or a rootkit on the machine, then it may hinder the installation

process, or make a patch on the software being installed. This will render the software ineffective.

If the virus shuts down all processes or attempts to delete all files, then the user will certainly get some indication that there is some virus residing on the machine, it is also not possible to protect against such viruses that will kill all running processes. The last assumption is made so that the anti-virus can request for upgrades and receive new definitions.

3.3. Types of attacks

Virus attacks can be classified in two broad categories. The first is where the malware attempts to delete every file on the system, leading to a system crash. It is not possible to deal with an attack like this; the only solution is to re install the operating system and all the required software. The developer of such an attack does not get any profit or return from this type of attack, this kind of attack is aimed just to have fun.

The second type of attack is more complex, and ironically, can be dealt with compared to the first type of attack. This type of attack includes patching operating system files, installing keystroke loggers, spywares and rootkits to convert the machine into a zombie or part of a botnet. These viruses will somehow attempt to stay concealed within the machine or run in the background. A big motive of these attacks might be to steal private information. Generally this type of attack will not involve deleting system files, it will only attempt to shut down or patch a few programs in an attempt to conceal itself. Hence a virus will more often than not try to disable

the anti-virus, or not allow update of virus definitions to happen so that it can stay hidden, or patch the definitions to make the anti-virus think that the virus is a legitimate program. The anti-virus programs need to overcome these attacks. This thesis is aimed at countering this exact problem.

CHAPTER 4

BUILDING BLOCKS

A hardware based anti-virus would possibly be the best solution to the problem of preventing malware from disabling it. However, this would require a change in the architecture of consumer computers. This involves lot of expenditure on part of the manufacturers, which would in turn reflect on the price of the machines, and it is likely that many consumers would prefer not to incur the extra cost.

The next best mode of implementation of an anti-virus that cannot be disabled is a virtual machine. A virtual machine based anti-virus would reside in between the OS and the hardware. This methodology would not require any new hardware and would not incur any additional cost to the manufacturers. An implementation of this kind would ensure that a malware will not have access to kill the anti-virus since it resides in the layer below the OS. However, this solution is a complex solution and it would be very difficult to address issues like user interaction. A virtual machine based anti-virus should not provide the operating system any means to stop it. This is because there is no way to differentiate between the function calls executed by the user and those executed by a malware. If user interaction is provided, then it must be done by means of a separate I/O channel. This again requires the installation of special hardware. Even if the special hardware is installed by the user, it is very likely that social engineering schemes may succeed in getting the user to turn off the virtual machine based monitor.

Integrating the anti-virus into the kernel is not an effective solution as rootkits are known to compromise the kernel, moreover updating the kernel is a complex issue. This means that the anti-virus has to be implemented as part of the user space.

In this chapter few solutions to countering the problem of a virus switching off the anti-virus process that were implemented during the course of this research are discussed along with their shortcomings. The designs described in the following sections were implemented in the Windows environment.

4.1. Injecting Code in Logon process

All OS have a logon process that is constantly running in the background. It monitors for any event that corresponds to any logon functions. Examples of such events may be logoff, system lock/unlock, shutdown, restart, etc.

In Windows the process that handles logon activities in Winlogon.EXE, this executable has mainly function calls written inside it. Most of these functions are present inside different DLL files. It is possible to modify an executable in windows environment without obtaining the source code by performing a few tricks that virus writers use: mainly changing the entry point of the executable [22]. The DLL files used by winlogon are also modified to insert the desired code there. One such DLL used by winlogon is Gina [26]. The source code for gina is available in Windows SDK. This code is modified to produce the desired functionality and the resulting DLL replaces the original stub library provided by Microsoft. This way the anti-virus runs as part of the windows logon process and does not appear in the system process list.

4.1.1. Shortcomings.

Replacing Gina DLL is a technique that has been used for quite some time now to provide third party logon methods. Anyone can replace

Gina library in the system. Winlogon does not know the exact name and location of the DLL. It merely looks it up in the system registry to locate the DLL. Since Windows registry is an entity that can be modified by anyone, this methodology does not guarantee whether the anti-virus will stay active in case of virus attacks. Any virus that obtains administrator rights can change the registry entries to point to a malicious DLL using Windows API calls. Once the DLL entry in the registry is replaced, the original DLL can be deleted, and hence the anti-virus process would be lost from the machine.

4.2. Watch Processes

Malicious programs use Windows API's similar to GetProcessId [20] to obtain the process id of any process by name and kill the process. The same technique can be used to keep a process alive, or restart any process that has been killed. A standalone process can monitor whether the anti-virus process is running or not. If the anti-virus is not running then this process can prompt the user that the anti-virus has been shut down. It can be left to the user to restart the anti-virus. This technique is combined with the technique used for hiding secret keys by means of multiple threads [75]. The anti-virus program can be monitored by 'n' different processes running in the system. There processes are provided with the name of the anti-virus process as a start up parameter, and each of these processes check whether or not the anti-virus process is running after specified intervals of time. In addition they are also provided with the name of the other n-1 watch processes. This way each of the watch process monitors whether the other watch processes are running or have been disabled by any malicious

entity. This design enables the revival of any security process that might be killed by a trojan.

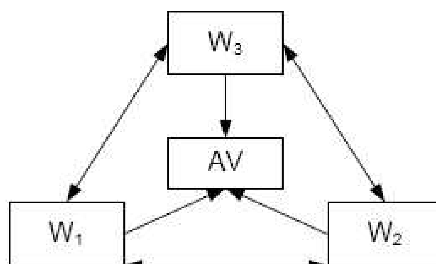

FIGURE 9. 3 Watch processes to monitor the anti-virus

Fig. 9 shows a 3 watch process model. AV is the anti-virus software to be monitored, W_1, W_2, W_3 and the three processes that watch over the anti-virus. W_1 monitors W_2 and W_3, W_2 monitors W_1 and W_3, while W_3 monitors W_1 and W_2. If any of the three processes detect that any of the watch processes is not running, it starts the killed process. If any of the three processes detect that the anti-virus is not running, the user is informed and prompted to decide whether or not to turn on the anti-virus. Once the user makes the decision to not turn on the anti-virus, the processes store the answer and do not prompt the user again.

4.2.1. Shortcomings. Although this design gives a certain amount of confidence in the system that a security process will get restarted after certain amount of time, the reliability of the system depends on running of at least one of the watch processes. Theoretically many processes can monitor the security software; however, there is a serious performance issue with this design. The time quantum between two checks has to be as small as possible. However, if all the watch processes execute within a

small time quantum, system performance will degrade. In addition it is not possible to make a claim that no Trojan can kill all the watch processes at once. A malicious entity can obtain the snapshot of a system at a given time, and then process the findings offline to gain some information about the names and process id's of the watch processes. Then by means of a remote command, kill all the processes at the same instant (or within the quantum of execution of two checks), and then proceed to kill the anti-virus process.

4.3. Install as a Different Process

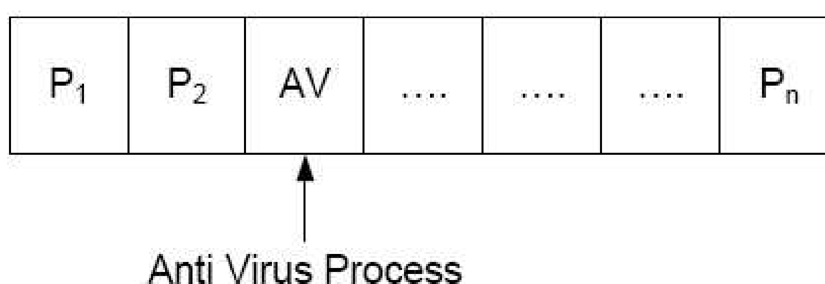

FIGURE 10. Query results before camouflaging the anti-virus software

Camouflaging is a technique that is used by many malicious programs. Viruses are known to install sections of their code in other programs so as to hide their presence. A similar trick can be used to hide the anti-virus. The anti-virus program can be installed as a completely different software. This involves replicating the entire directory structure and file names of that software and placing the anti-virus program's files in them. The installation suite contains a list of commonly used software, upon installation request, the suite finds out which software in its list have not been installed on the machine. The installation suite provides the truncated

list to the user to choose the software in whose name and structure the anti-virus should be installed. On getting the user choice, the suite proceeds to replicate the directory, file structure and registry entries of the chosen software. This methodology prevents many malware attacks that identify the presence of an anti-virus program by examining registry entries, file system and process table entries. Fig. 10 shows the results of a query that finds out the processes running in the system normally. The query that generates the list of processes running in the system would not reveal any information about which process is the anti-virus process.

Fig. 11 shows the query results after the anti-virus process is camouflaged as another process in the system. It can be inferred from the two scenarios that the query will not provide the malicious entity any information about the anti-virus running in the system. A scan of the file system on the machine would also not reveal the presence of the anti-virus. Hence many attacks that attempt to shut down the anti-virus can be overturned by this design.

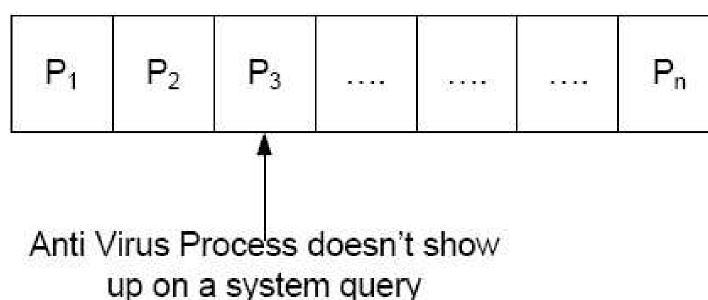

FIGURE 11. Query results after camouflaging the anti-virus software

4.3.1. Shortcomings. This design uses some tricks used by the virus world in masquerading as a different program. Although this offers some protection from

malware that kill the anti-virus software, there is no guarantee that the anti-virus process can survive long enough on the infected machine till the updated definitions for a new virus arrives. Even though the file structure and directory structure of some software can be replicated, it is not possible to replicate the exact hash value of the executable that is being faked. If a malicious program tries to find the hash value of every executable on the system and compares them with known hash values, then it is bound to find a hash value that does not match for one executable. Even though this attack would require heavy computations, it cannot be ruled out. An attempt can be made to trap any API call that attempts to fins the hash value of an executable on the system and provide a fake value. This would require installing some sort of a rootkit. The use of a rootkit has already been argued against in previous sections. Hence this design also has a few shortcomings that can be exploited by malware writers to kill the anti-virus process.

Another shortcoming in this design is that the anti-virus program should start executing on system start up. In this design, there is a possibility that the anti-virus might get installed as a software which under normal circumstances would not appear on the startup items list. This information can serve to be very useful to attackers in killing the anti-virus program even if it is installed under a different name.

4.4. Summary

In this chapter three possible solutions to protecting the anti-virus from malware that kill the security software, along with their shortcomings were discussed. In

the next chapter all the previous mentioned designs are combined to bring about a robust solution that survives virus attacks to a greater extent.

CHAPTER 5

DESIGN

To evade detection by malicious programs, the anti-virus should remain hidden from all processes in the system. The reason that it should be hidden from all processes is that software components running on the computer cannot be trusted, as any program might be infected by a malicious entity. To effectively hide a program, its file structure, registry entries and process table entries have to be hidden. These issues are addressed by a two fold process. The first step involves installing the program as a different program on the machine. This serves to hide the file structure and registry entries, and also ensures each copy of the anti-virus looks different. The next step involves using code injection to migrate the program code and library into other processes. Migration of code serves to hide process table entries from all other system components. By performing code injection and the subsequent migration after certain time intervals, another threat is addressed. It becomes difficult for malware to locate where the anti-virus resides currently even if it finds where the anti-virus resided previously.

5.1. Installing the Program

The anti-virus process must be installed in a way such that a malicious entity cannot identify it. In section 4.3 a methodology was described to install the anti-virus process as a different program (P'). The shortcomings of the designs were that a program may take the hash value of all executables in the system and identify a value that does not match for a popular software, and that the process may be identified by the startup entry. If these shortcomings are fixed then the solution can

be a fairly viable solution to the problem. It must be noted however, that it is very difficult to produce an absolute solution to the problem at hand, and that attempts can be made only to make the process of killing security software much harder than they currently are.

It is quite difficult to fix the first shortcoming mentioned above. However, storing and comparing hash values of various versions of different software requires high computational and storage complexity. This type of checking is done by the anti-virus software to identify malicious code. The anti-virus is a memory heavy application that has many definition files which occupy significant disk space. With these factors taken into count, it is very difficult for a malicious entity to identify a fake executable. In spite of this, a malicious entity may still compute hash values of all executables in the system and send it to some central server for identification of a fake binary. This issue is addressed in section 5.4

The second shortcoming of the design was that the process may be identified by looking at the start up entries in the system registry. This attack is very abstract in nature, as an attacker cannot be completely sure that a candidate program is the anti-virus process, nevertheless, this shortcoming is fixed in the following section.

5.2. Starting the Process

As seen earlier, malware attempt to search for registry entries to find any entry with the names of popular anti-viruses and delete the entries, including start up entries. Hence, the point where the start up information about the anti-virus is stored has to be cloaked. This can be achieved by forcing another process to start up

the anti-virus. If a randomly chosen process starts the anti-virus program at every consumer machine, then it would be very difficult for malware to delete the start up point for the anti-virus.

In theory any program can be chosen to act as the starter (S) for the anti-virus, however, since the anti-virus must load along with the events at system boot, the best choices for the starter process would be a program that loads on system start up. Due to this criterion, OS components that run during start up would be the best candidates for starting the anti-virus. In Windows, there are many services and processes that start up on system boot, any one of those can be used as the starter for the anti-virus. Any process can create another process in the system as long as it has sufficient privileges to do so. If the starting program is running with administrator privileges then it can start the execution of any number of programs.

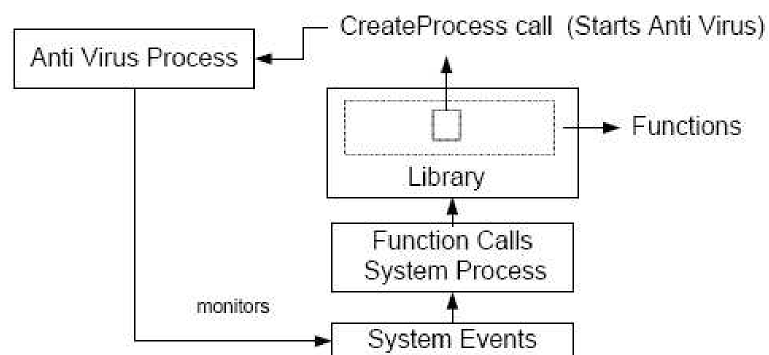

FIGURE 12. Starting the anti-virus

Fig. 12 illustrates the idea. S is also provided with a seed value (pw). The seed value is used in section 5.3 to perform code injection. An issue that still remains is that an attacker can replace system's DLL files, just like the anti-virus program does. This is addressed in section 5.4

5.3. Execution of the Process

The anti-virus process must run continuously in the system. In section 5.1 the program was installed under a different name and directory structure. In section 5.2, P' was started by means of another program creating some amount of obfuscation over its location. In section 1.6 it was seen that most trojans identify the location of the anti-virus by executing a system query on the process table or the registry or by means of a search in the process table and the file system. In the previous two sections, it has been made fairly difficult for virus writers to kill the anti-virus process by simply scanning and searching. However, it still cannot be claimed that this process will ensure that the anti-virus is not identified by a malware. A malware can attempt to identify the files a process reads. The anti-virus would read all files on the system during execution and in addition be memory intensive. Information like these can give access to the name of the anti-virus process in the system. To make it even tougher for a malware to identify the anti-virus, we can include the technique mentioned in section 2.3. It is possible to inject the arbitrary code and library files into any running process [7]. Code injection can serve to move the anti-virus program in the system from one process space to another.

Once P' is loaded by S in the system, it has to migrate the library files and function calls to another process space. This is done by using the user provided seed 'pw' and computing the process id (pid) of a target process (T) from it. Once T is identified, P' injects the DLL's and code into it using the technique described above. Once T is injected with the required code, P' (the anti-virus process) stops. The anti-

virus code however, continues executing under the address space of T. After a certain time period the injected code within T identifies another process (T_1) under which the entire anti-virus code is to be migrated. The process of migrating the code from T to T_1 is different from P' to T in the aspect that the libraries and code loaded into T have to be cleaned. The anti-virus code continues migrating from T_1 to different programs until the system is rebooted or the user disables the anti-virus program.

5.4. Watch Processes

In section 4.2 the concept of running multiple processes to monitor the anti-virus program was introduced. It is a useful technique to determine whether any program is running or not in the system. In this design P' starts a number of watch processes and provides them with pw, the seed for determining the target process. Since the migration of program is deterministic, the result computed by P' or any other T while identifying the next target process, and the results computed by the watch processes will be the same. Furthermore, since the watch processes also know the names of the library files and the exact code that is migrated, it can do a memory scan on the process that it computed to check if these libraries are present in the process' address space. If it finds that the process does not have the libraries, which would happen only if the anti-virus program was stopped, it notifies the user that the anti-virus program is not on. If the user chooses to start the anti-virus, then one of the watch processes load P'. At this point the user is requested to provide a fresh pw. This way the anti-virus program is kept running on the machine, even if an attacker identifies and kills the process.

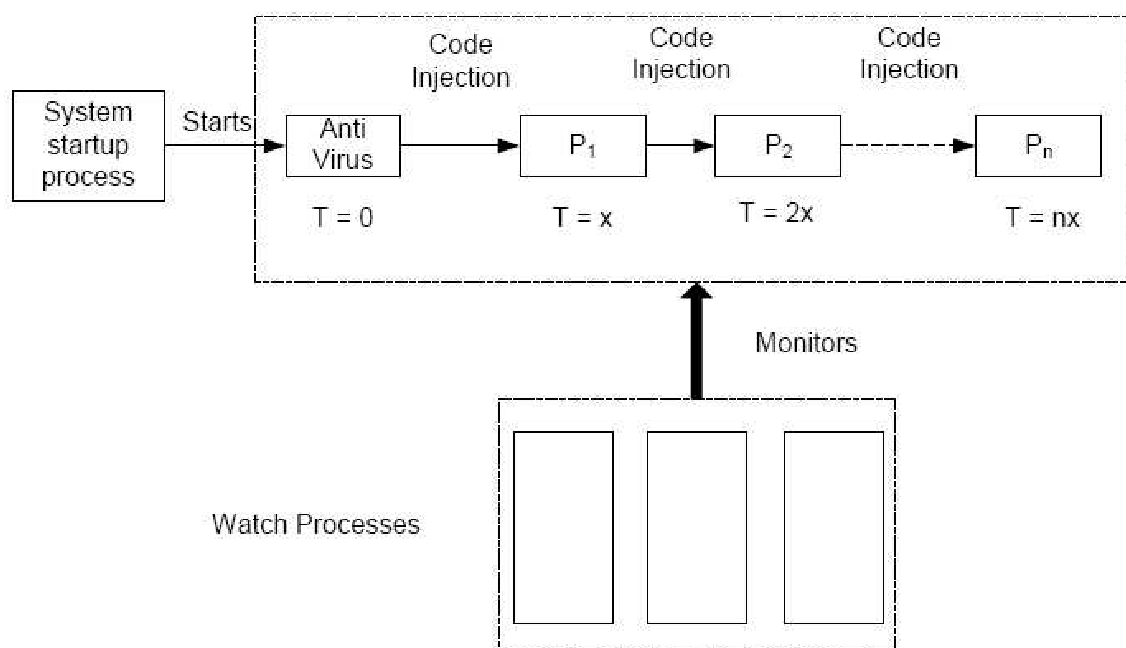

FIGURE 13. Starting and migration of anti-virus program

Fig. 13 shows the system of hiding the anti-virus. A random system process, loads the anti-virus program. The anti-virus program after executing for some time - 'x', selects a process P_1 running in the system to inject the code and libraries into. P_1 executes for some time in the system, and injects the code and libraries into P_2 and so on, the entire process is monitored by the watch processes.

5.4.1. Shut Down Events. In spite of the measures taken to protect the anti-virus program, a major threat remains: a malware modifying the process S. If a malware manages to change S, then it can prevent the start of the anti-virus program. Although it can be argued that a user can start the process manually, not many users would notice that a startup program failed to load. To ensure that a malicious program does not alter the files of S, the watch processes can compute the hash

value of the known good copy of S, store it, and compare it with the hash value of S every time before system reboot. This checking can also be extended to P' and virus definition files. This would ensure that even if an attacker tampers with the file system, it gets trapped and fixed before system reboot.

5.5. Virus Definition Files

A virus definition file contains information that contain patterns which are used to identify viruses. Most anti-virus software use definition files to separate the virus patterns from the executable so that updating the software remains relatively easy. This section is not implemented in this thesis, it outlines future work that if implemented can augment the virus detection process.

5.5.1. Whitelists.

In chapter 1 it was seen that the anti-virus software identifies malware by means of definition files. These definition files are currently *blacklists*, i.e., they contain the list of all known malicious entities and patterns. Due to this, new viruses and trojans are able to successfully exploit the zero day attack. The zero day attack can cause major financial losses. The solution to this problem is simple: the anti-virus and other security suited must move from using plain blacklists to a combination of *blacklists* and *whitelists*. A Whitelist would be the list of all known good programs. This list would be similar to that used by a Firewall. A firewall initially does not have any trusted outgoing connections; it gradually builds trusted connections while interacting with the user. Similarly, the anti-virus software can store the hash values of all known executables in the system while it gets installed.

On subsequent scans the anti-virus software can check whether any executables have changed, or any new executables have been installed. If there are any changes in the hash values of executables, then those executables can be flagged for user review. For the user scanning this list may be a difficult and arduous task, but some attacks like patching on to the explorer or to certain other system files can easily be uncovered by this. If the user knows that no updates were performed, then most likely the changes are part of a virus attack.

5.5.2. Storage of definition files. The anti-virus software is designed to store additional information like whitelists, hash values of libraries, apart from blacklists in this thesis. If all this information is stored in one location, it can give rise to a large definition file. A malware can monitor this file to identify which process obtains read or write locks on it. This way the anti-virus program and other watch processes can be identified by a malware. The solution to this problem is to hide the presence of definitions in a file and scatter the files across the system by means of a scatter function S' as shown in Fig. 14.

The definition files can be embedded in files by means of Steganography. Fig. 15 shows a scheme to store definitions. The definitions are replicated and grouped together in random order. After this, each group is placed inside the image files. Apart from the definitions, the image files should also contain a section for storing the checksum for all the definitions present in the file. The checksums must be encrypted with the private key of the software provider. This is done so that anyone can decrypt the value using the software provider's public key to read the checksum, but no one

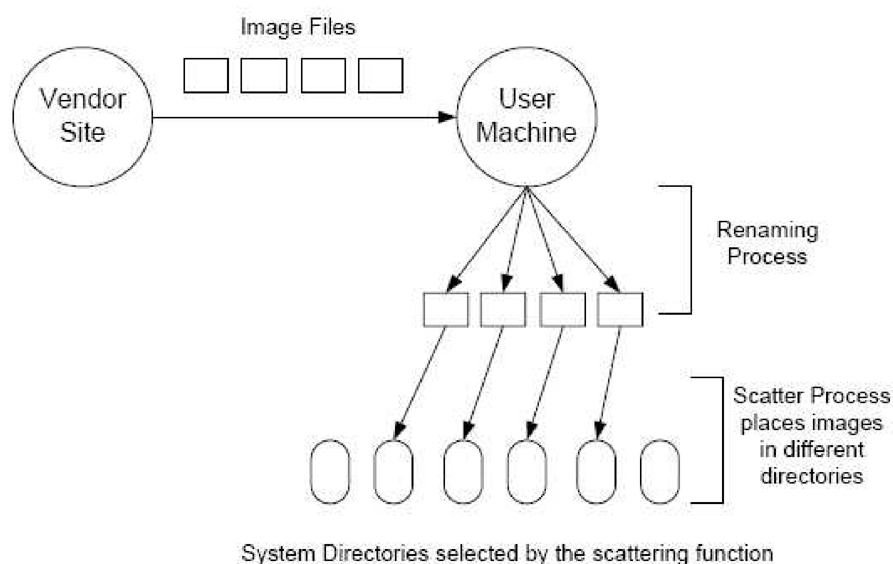

FIGURE 14. Receiving and storing image files

can encrypt and place a false checksum. This takes out an attack where a malicious code might remove itself or change some definition in an attempt to confuse the software. There is no way that malicious code can place a correct checksum without knowing the private key of the software provider.

The individual virus definitions themselves should not be stored at only one place. They can be stored at multiple places, and grouped randomly. The storage files need to be renamed randomly at every user machine. This needs to be done so that an attacker doing reverse engineering at his machine cannot use that knowledge to attack the definition files on other user machines. To do this, a renaming function - R can be created, which will be placed inside the binary by the installation suite. The function R can be a pseudo random number generator, whose seed can be placed in the executable, the seed must be different on every executable. R must generate a name for each of the image files. The names provided by the software vendor can be

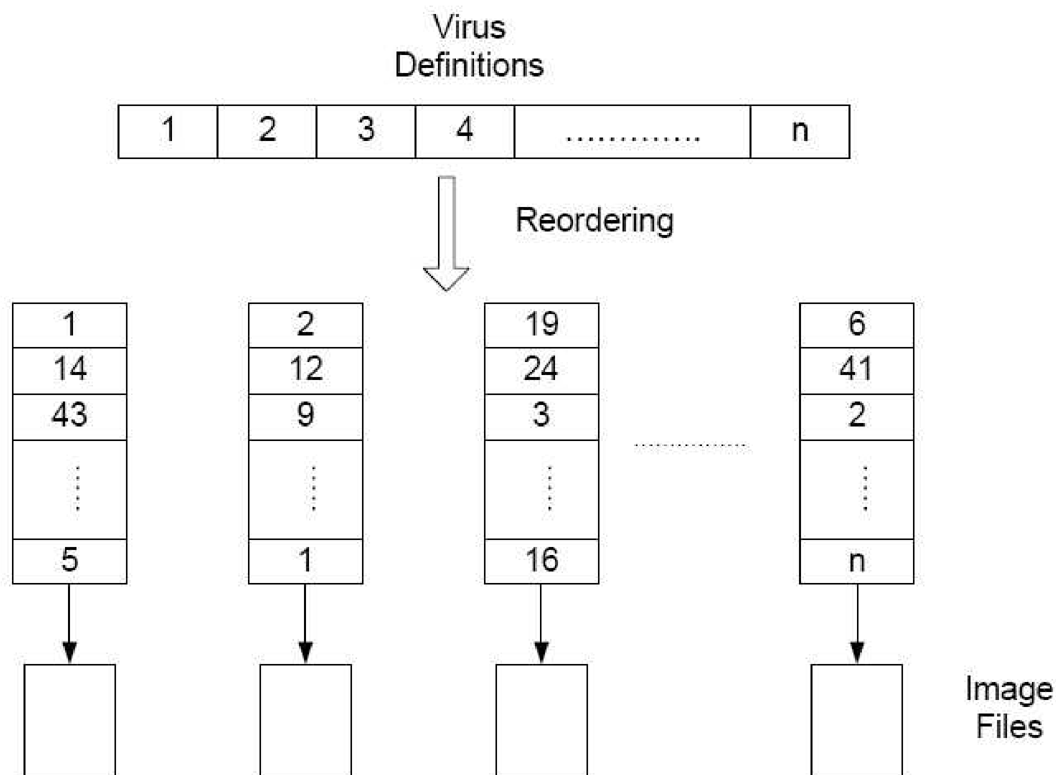

FIGURE 15. Splitting and placing of definition files

treated as base names, and the list of base names can be present inside the executable to easily locate which files to load definitions from.

5.6. Summary

This chapter presented a software based approach for hiding the anti-virus program from malware. This problem is essentially the problem of managing a secret in the memory. The technique in this thesis attempts to obscure vulnerable locations of the anti-virus process. A software based solution is not guaranteed to be fool proof in hiding the anti-virus program, however, the migrating of code from one process space to another offers effective protection that makes it much tougher for malware to

identify and kill the anti-virus. The solution comes with self check scheme to protect the startup point of the process. It also contains methods to monitor whether the anti-virus program is running or not, by means of multiple watch processes and user supervision. Finally, certain enhancements to the virus detection methodologies were proposed.

CHAPTER 6

PROTOTYPE

This chapter describes the skeletal implementation of the system described in chapter 5. The system was implemented in the Windows environment. Insertion of start up information into a system process and code injection were implemented using Visual C++ 6.0, while watch processes were implemented under the .NET framework. Section 6.1 describes the implementation of placing the start up call inside a windows system process. Section 6.2 describes the implementation for the execution and migration of code. Section 6.3 describes the implementation of watch processes.

6.1. Placing start up information in Windows system process

As seen in section 5.2, any Windows system process that executes on system boot may be used as the program to start the anti-virus program. In this thesis, Winlogon.Exe has been used as the starter process S. Winlogon also receives the user provided seed value (pw) to perform injection and migration of code.

In Windows, there is an API *CreateProcess* [8] that can be called by any program to start a process. CreateProcess takes in the following parameters - name of the module to be executed, command line arguments and a few other optional parameters. The name and location of the module to be started is passed as the first parameter, and the user provided seed value is passed as the second parameter to the API call.

To start the anti-virus process, CreateProcess has to be called by S. To do this, the executable of S has to be modified to explicitly call the API, or the libraries it

uses can be patched on, to call the API. In this implementation, the library used by Winlogon - gina.dll has been modified to call the API.

When the system boots up, Winlogon.exe is called to authenticate the user and provide an environment for the user to work on the system. Winlogon calls some routines that are stored in Gina.dll. Gina contains many functions, one (or more) of them can be patched to contain a call to start the anti-virus process. When the patched function is called by Winlogon, the call CreateProcess executes, starting the anti-virus.

In commercial practice, the call can be placed inside any of the windows services on every machine. This will ensure that even if an attacker uses some analyzing technique to find the location of the API call on one machine, the information will not be useful in another machine.

6.2. Code Injection

The technique and API's used to inject libraries is described next, followed by the techniques used to inject program code in a process.

6.2.1. Injecting Libraries. Any process can load any library on the system using the API call *LoadLibrary* [9] that is contained in the library 'kernel32.dll'. However, the API cannot be used to directly load DLL files into any arbitrary process in the system. To load the required library files into the address space of the process the following steps are followed. First, the process allocates virtual memory for the library to be written into. Then the reference to the libraries are written on the allocated

memory. Finally, the process is forced to execute at the memory location of the library or the function.

The first step is achieved by using the API *VirtualAllocEx* [10]. The function takes in the handle to the process in which virtual memory has to be allocated and the size of memory to be allocated. If the function succeeds, it returns the base address of the allocated region of pages. The returned address is used to write the reference to the Library in the target process using the API *WriteProcessMemory* [11]. Once this is done the process has to be forced to load the library, this is done by means of another windows API *CreateRemoteThread* [8]. This function causes a new thread of execution to begin inside the address space of the target process. This way any DLL can be injected into any process in the system as long as the injecting program has sufficient privileges.

6.2.2. Injecting Code. The difference between this method and the method described above is that instead of making the process load a library, the program writes the code directly to the address space of the process. This method is not as stable as compared to the previous one and is restricted to calling functions from the loaded libraries [7]. The steps involved in writing code on an executing process are similar to the ones described for injecting the DLL.

It must be noted that the above methods are not the only ones used for code injection. Another popular technique used by hackers is the usage of windows debugging functions to suspend the execution of a process; this also saves the context of the process. Then the EIP register (it contains offset to next instruction to be executed)

is modified to point to the location of the code to be executed. When the process resumes, it begins execution at the location that is pointed by EIP register [12]. Since this methodology modifies the EIP register the old code cannot run simultaneously with the new code. This means that if the anti-virus program migrates in the system at regular intervals of time, many system events such as storing the context of a process will get generated, these can cause a malware to identify the anti-virus process. Therefore, in this thesis, the first two methods are used for moving the anti-virus program.

6.3. Watch Processes

A scheme similar to that described in section 4.2 is implemented in this thesis. Three watch processes were installed as Windows services under the .NET framework. The class *ServiceController* [13] was used to determine if the other two watch processes were executing. The time quantum between two executions of every process was specified as 3 milliseconds. The anti-virus process is monitored by calling *Process.GetProcesses* [14] on the system. The name of the initial process is passed to the watch processes as a parameter, the later processes are found by computing a function on the seed provided by the user, the libraries used by the process were identified using the scheme provided in MSDN [15].

6.4. Overhead and Performance

The system performance in terms of processor time and memory usage was measured using the 'Perfmon' tool provided by Microsoft. The results of the analysis

	Normal System	Modified System
Processor Time	.78%	7.7%
Unused Memory	281 Mb	279 Mb

Table 1. System Overhead

are displayed in table 1. The machine was first tested with the installed Anti-Virus components running, and then the components were disabled to obtain performance under normal circumstances. The processor overhead on the system was approximately 7% and 2 Mb of extra memory was utilized by the modules in this design as compared to a normal system.

The time taken to inject code was measured using the MFC call *GetSystemTime*. The time taken to inject 200 Kb of code was 20 milliseconds, while time taken to inject 300 Kb of code was 45 milliseconds. Most Anti-Virus software are between 1 - 1.5 Mb in size. If we take the current readings into account, it can be concluded that it would take less than a second to migrate the entire Anti-Virus process.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis, an approach was presented to improve the reliability of the anti-virus process by hiding its presence from other processes on the machine. The reason that the anti-virus program has to be hidden from all other processes is that a malware may infect any process on the system, in such a situation no component of a consumer computer can be trusted.

The first step in solving this problem was changing the names of the files and changing the registry entries by installing the process under a different name. This step helps in working around attacks that scan the registry entries and the file system to identify the anti-virus program. Next, the program was started by means of a different process, by this; the starting point of the anti-virus program was obscured. After this, the process was continuously migrated to different address spaces to avoid detection by any malware. This particular component enables the anti-virus process to overcome attacks that take a system snapshot, and identify the anti-virus process by finding the files used by each process. By moving the code at regular intervals of time, such a snapshot would not be very useful in killing the anti-virus process as it would have migrated to another process space while the results of the snapshot are calculated. After this, multiple watch processes were installed to detect if the anti-virus program is shut down at any point of time. The design also included checking of library files at system shutdown to detect any unauthorized changes. Finally, the implementation of whitelists and scattering of definition files was left as future work.

REFERENCES

[1] http://www.webopedia.com/TERM/P/polymorphic_virus.html.

[2] http://www.microsoft.com/athome/security/viruses/intro_viruses_what.mspx.

[3] http://www.cknow.com/vtutor/NumberofViruses.html.

[4] http://www.eweek.com/article2/0,1895,2034680,00.asp.

[5] http://www3.ca.com/securityadvisor/virusinfo/virus.aspx?id=42627.

[6] http://searchservervirtualization.techtarget.com/sDefinition/0,,sid94_gci1083767,00.html.

[7] http://www.codeproject.com/threads/winspy.asp.

[8] http://msdn2.microsoft.com/en-us/library/ms682425.aspx.

[9] http://msdn2.microsoft.com/en-us/library/ms684175.aspx.

[10] msdn2.microsoft.com/En-US/library/aa366890.aspx.

[11] http://msdn2.microsoft.com/en-us/library/ms681674.aspx.

[12] http://www.codeproject.com/dll/DLL_Injection_tutorial.asp.

[13] http://msdn2.microsoft.com/en-us/library/system.serviceprocess.servicecontroller.aspx.

[14] http://msdn2.microsoft.com/en-us/library/1f3ys1f9.aspx.

[15] http://msdn.microsoft.com/msdnmag/issues/02/06/debug/.

[16] The beast. http://lists.virus.org/dshield-0310/msg00337.html.

[17] Beast trojan. http://www.answers.com/topic/beast-trojan.

[18] Cert incident note in-99-02.
http://www.cert.org/incident_notes/IN-99-02.html.

[19] Deobfuscation is in np.

[20] Getprocessid. http://msdn2.microsoft.com/en-us/library/ms683215.aspx.

[21] Hi-def dvd security is bypassed.
http://news.bbc.co.uk/1/hi/technology/6301301.stm.

[22] Inject your code to a portable executable file.
http://www.codeproject.com/system/inject2exe.asp.

[23] Malware report: The impact of malicious code attacks.
http://www.computereconomics.com/article.cfm?id=1090.

[24] Social engineering fundamentals. http://www.securityfocus.com/infocus/1527.

[25] Virus list.
http://www.viruslist.com/en/viruses/encyclopedia?chapter=153311150.

[26] Winlogon and gina. http://msdn2.microsoft.com/en-US/library/aa380543.aspx.

[27] Cert advisory ca-2002-12: Format string vulnerability in isc dhcpd, 2002.
http://www.cert.org/advisories/CA-2002-12.html.

[28] Cert vulnerability note vu#282403, 2002.
http://www.kb.cert.org/vuls/id/282403.

[29] Cert vulnerability note vu#496064, 2002.
http://www.kb.cert.org/vuls/id/496064.

[30] C. ARPANET and A. NSFNET. Chapter 6 inventing the internet.

[31] N.C.F.L. Ban. Study documents rate, nature of hacker attacks. *IT Pro, News Brief*, march 2007.
http://csdl2.computer.org/comp/mags/it/2007/02/f2004.pdf.

[32] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139:19–23, 2001.

[33] Richard Barnhart. Notes on computer viruses. http://courses.cs.vt.edu/professionalism/Viruses/viruses.html.

[34] Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation0. *Commun. ACM*, 27(1):42–52, 1984.

[35] R.M. Best. Preventing Software Piracy with Crypto-Microprocessors. *Proceedings of IEEE Spring COMPCON*, 80:466–469.

[36] T.M. Chen. Trends in Viruses and Worms. *The Internet Protocol Journal*, 6(3):23–33, 2003.

[37] D.M. Chess and S.R. White. An undetectable computer virus. *Proceedings of the 2000 Virus Bulletin Conference (VB2000)*, 2000.

[38] S.T. Chow, H.J. Johnson, and Y. Gu. Tamper resistant software-control flow encoding, aug 2004. US Patent 6,779,114.

[39] Frederick B. Cohen. Operating system protection through program evolution. *Comput. Secur.*, 12(6):565–584, 1993.

[40] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. *Department of Computer Science, The University of Auckland*, 148, 1997.

[41] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 28–38, 1998.

[42] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, 1998.

[43] CS Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *Software Engineering, IEEE Transactions on*, 28(8):735–746, 2002.

[44] A.M. Devices. Inc., amd64 architecture programmer's manual. vol. 2. *System Programming, Sept*, 2003.

[45] W. Ford and BS Kaliski Jr. Server-assisted generation of a strong secret from a password. *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2000.(WET ICE 2000). Proeedings. IEEE 9th International Workshops on*, pages 176–180, 2000.

[46] J.A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2):363–389, 2000.

[47] I. Griggs. What's your threat model, 2003. Website: http://iang.org/ssl/wytm.html.

[48] S. HALEVI and H. KRAWCZYK. Public-Key Cryptography and Password Protocols. *ACM Transactions on Information and System Security*, 2(3):230–268, 1999.

[49] IBM Inc. Version history of CCA Version 2.41, IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide for the IBM 4758-002. *IBM, pg xv (Feb 2002).*

[50] Matsushita Electric Industrial Co. Ltd Microsoft Corporation Sony Corporation Toshiba Corporation The Walt Disney Corporation Warner Bros. Intel Corporation, IBM Corporation. Advanced access content system: Introduction and common cryptographic elements, feb 2006.

[51] J.O. Kephart, G.B. Sorkin, D.M. Chess, and S.R. White. Fighting computer viruses. *Scientific American*, 277(5):56–61, 1997.

[52] ST King and PM Chen. SubVirt: implementing malware with virtual machines. *Security and Privacy, 2006 IEEE Symposium on Security and Privacy*, page 14, 2006.

[53] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. *Proceedings of the 5th ACM workshop on Digital rights management*, pages 75–82, 2005.

[54] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. *Proceedings of the 6th International Workshop on Information Security Applications (WISA)*, pages 194–206, 2005.

[55] M. Madou, L. Van Put, K. De Bosschere, and S. Pietersnieuwstraat. Understanding Obfuscated Code. *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference*, pages 268–274, 2006.

[56] P.P.F.D. Manual. Volume 3: Architecture and Programming Manual. *DevelopersInsight: Intels Web Site for Developers on CD-ROM products & tools*, 24142804.

[57] E. Messmer. Pentagon gets smart. *CNN, September*, 1999.

[58] J. Munro. Antivirus research and detection techniques, 2002. http://www.extremetech.com/article2/0%2C1558%2C325439%2C00.asp.

[59] I. Muttik. Stripping down an AVENGINE. *Virus Bulletin Conference*, 59, 2000.

[60] C. Nachenberg. Understanding and managing polymorphic viruses. *The Symantec Enterprise Papers*, 30, 1996.

[61] C. Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, 1997.

[62] R. Perlman and C. Kaufman. Secure Password-Based Protocol for Downloading a Private Key. *Proceedings of the 1999 Network and Distributed System Security*, 1999.

[63] E. Rescorla. *SSL and TLS*. Addison-Wesley, 2001.

[64] J. Rutkowska. Subverting Vista Kernel for Fun and Profit. *Black Hat USA*, August 2006.

[65] P.H. Salus. *Casting the Net: From ARPANET to Internet and Beyond.* Addison-Wesley, 1995.

[66] D.J. Sanok Jr. An analysis of how antivirus methodologies are utilized in protecting computers from malicious code. *Proceedings of the 2nd annual conference on Information security curriculum development*, pages 142–144, 2005.

[67] B. Schneier. Attack trends: 2004 and 2005. *Queue*, 3(5):52–53, 2005.

[68] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 45–54, 2002.

[69] E.H. Spafford. The internet worm program: an analysis. *ACM SIGCOMM Computer Communication Review*, 19(1):17–57, 1989.

[70] S. Sparks and J. Butler. Shadow Walker: Raising The Bar For Windows Rootkit Detection, 2005.

[71] P. Ször. *The Art of Computer Virus Research and Defense.* Addison Wesley for Symantec Press.

[72] S.E. Transaction. Specification. *Master-Card and VISA. Book*, 1.

[73] P.C. van Oorschot. Revisiting software protection. *6th International Information Security Conference (ISC).*

[74] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. *International Conference of Dependable Systems and Networks*, pages 193–202.

[75] L. Wang and P. Dasgupta. Software Based Approach for In-memory Secret Management. *In Proc.*

[76] Y. Wang, Faculty of Mathematics, Dept. of Combinatorics, Optimization, and University of Waterloo. *Using Mobile Agent Results to Create Hard-to-detect Computer Viruses.* Faculty of Mathematics, University of Waterloo, 2000.

[77] F. Witter, A.I. Handling, and H. Exploits. Responding to downloader-w.

[78] H. Yamauchi, Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Software obfuscation from crackers' viewpoint. *Proceedings of the 2nd IASTED international conference on Advances in computer science and technology*, pages 286–291, 2006.