2010

Analysis avoidance techniques of malicious software

Murray Brand

Edith Cowan University

Analysis Avoidance Techniques of Malicious Software

Murray Brand
BEng (Hons) (Electronics and Communications)
MEngSc (Electrical Engineering)
GradCert (Computer Security)

A thesis submitted for the Award of

Doctor of Philosophy

At the Faculty of Computing, Health and Science
Edith Cowan University, Mount Lawley Campus

Principal Supervisor Professor Craig Valli
Associate Supervisor Doctor Andrew Woodward

Submitted 30 November 2010

PUBLICATIONS ARISING FROM THIS RESEARCH

Brand, M. (2007). *Forensic Analysis Avoidance Techniques of Malware.* Paper presented at the 5th Australian Digital Forensics Conference, Edith Cowan University, Mount Lawley Campus, Western Australia.

Szewczyk, P., Brand, M. (2008). *Malware Detection and Removal: An Examination of Personal Anti-Virus Software.* Paper presented at the 6th Australian Digital Forensics Conference, Edith Cowan University, Mount Lawley Campus, Western Australia.

Valli, C., Brand, M. (2008). *Malware Analysis Body of Knowledge*. Paper presented at the 6th Australian Digital Forensics Conference, Edith Cowan University, Mount Lawley Campus, Western Australia.

Brand, M., Valli, C., Woodward, A. (2010). *Lessons Learned from an Investigation into the Analysis Avoidance Techniques of Malicious Software*. Paper presented at the 8th Australian Digital Forensics Conference, Duxton Hotel, Perth Western Australia.

Brand, M., Valli, C., Woodward, A. (2010). *Malware Forensics: Discovery of the intent of Deception*. Paper presented at the 8th Australian Digital Forensics Conference, Duxton Hotel, Perth Western Australia.

Brand, M., Valli, C., Woodward, A. (2010). *Malware Forensics: Discovery of the intent of Deception*. The Journal of Digital Forensics, Security and Law, 5(4), 31-42.

ABSTRACT

Anti Virus (AV) software generally employs signature matching and heuristics to detect the presence of malicious software (malware). The generation of signatures and determination of heuristics is dependent upon an AV analyst having successfully determined the nature of the malware, not only for recognition purposes, but also for the determination of infected files and startup mechanisms that need to be removed as part of the disinfection process. If a specimen of malware has not been previously extensively analyzed, it is unlikely to be detected by AV software. In addition, malware is becoming increasingly profit driven and more likely to incorporate stealth and deception techniques to avoid detection and analysis to remain on infected systems for a myriad of nefarious purposes.

Malware extends beyond the commonly thought of virus or worm, to customized malware that has been developed for specific and targeted miscreant purposes. Such customized malware is highly unlikely to be detected by AV software because it will not have been previously analyzed and a signature will not exist. Analysis in such a case will have to be conducted by a digital forensics analyst to determine the functionality of the malware.

Malware can employ a plethora of techniques to hinder the analysis process conducted by AV and digital forensics analysts. The purpose of this research has been to answer three research questions directly related to the employment of these techniques as:

1. What techniques can malware use to avoid being analyzed?
2. How can the use of these techniques be detected?
3. How can the use of these techniques be mitigated?

These questions were effectively answered by validating anti-analysis techniques, showing how the techniques can be effectively detected and mitigated as well as by analyzing malware collected from the internet. This research contributes to the knowledge of malware analysis and digital forensics by:

- Demonstrating that anti-analysis techniques can be very effective at hindering analysis by the tools typically used by analysts.
- Showing that the use of anti-analysis techniques can be effectively detected and mitigated by the use of appropriate analysis techniques, scripts and plugins.
- Support of claims virus signature based detection by anti-virus software can be far less than ideal.
- Showing that extensive use of packers and protectors are employed by network based malware collected from the internet to obstruct signature based detection and to hinder analysis.
- Support of an alternate paradigm of malware detection that could use detection of deception and anti-analysis techniques to detect malicious software instead of using virus signatures and heuristics.
- Identification of a Malware Analysis Body of Knowledge (MABOK) that incorporates anti-analysis techniques as a core component.
- Identification of deficiencies in analysis tools given the extent of available anti-analysis techniques.
- Determination of an appropriate analysis methodology tailored for dealing with anti-analysis techniques.
- Development of a taxonomy of analysis avoidance techniques.

DECLARATION

I certify that this thesis does not, to the best of my knowledge and belief:

(i) incorporate without acknowledgement any material previously submitted for a degree or diploma in any institution of higher education;

(ii) contain any material previously published or written by another person except where due reference is made in the text; or

(iii) contain any defamatory material.
I also grant permission for the Library at Edith Cowan University to make duplicate copies of my thesis as required.

Signature ....

Date ....

ACKNOWLEDGEMENTS

This thesis would not have been possible without the guidance and encouragement provided by my Principal Supervisor Professor Craig Valli and Associate Supervior Doctor Andrew Woodward.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

1.1. OVERVIEW

This thesis analyses techniques malicious software (malware) incorporates into its code to prevent and/or hinder the malware forensic analyst from conducting an analysis of the malware. The effectiveness of these techniques was validated in this research. A variety of procedures were developed and examined to determine if these anti-analysis techniques could be detected and mitigated. Malware collected from the internet was also analyzed to partially corroborate these techniques. This research found that a plethora of techniques are available to hinder the malware analyst and all of the techniques that were implemented in the course of this research were found to be effective at hindering analysis. Equally, detection and mitigation techniques were uncovered and also found to be effective at detecting and mitigating the anti-analysis techniques. Malware and forensic analysts and researchers will be the primary users of this research.

Aycock (2006, pp. 1-12) defines malware as “software whose intent is malicious, or whose effect is malicious”. Analysis of malicious software is essential for computer security management and is emerging as an important field of research. This is because malware is often targeted at organizations and is increasingly using anti-analysis techniques to prevent detection and analysis (Masood, 2004).

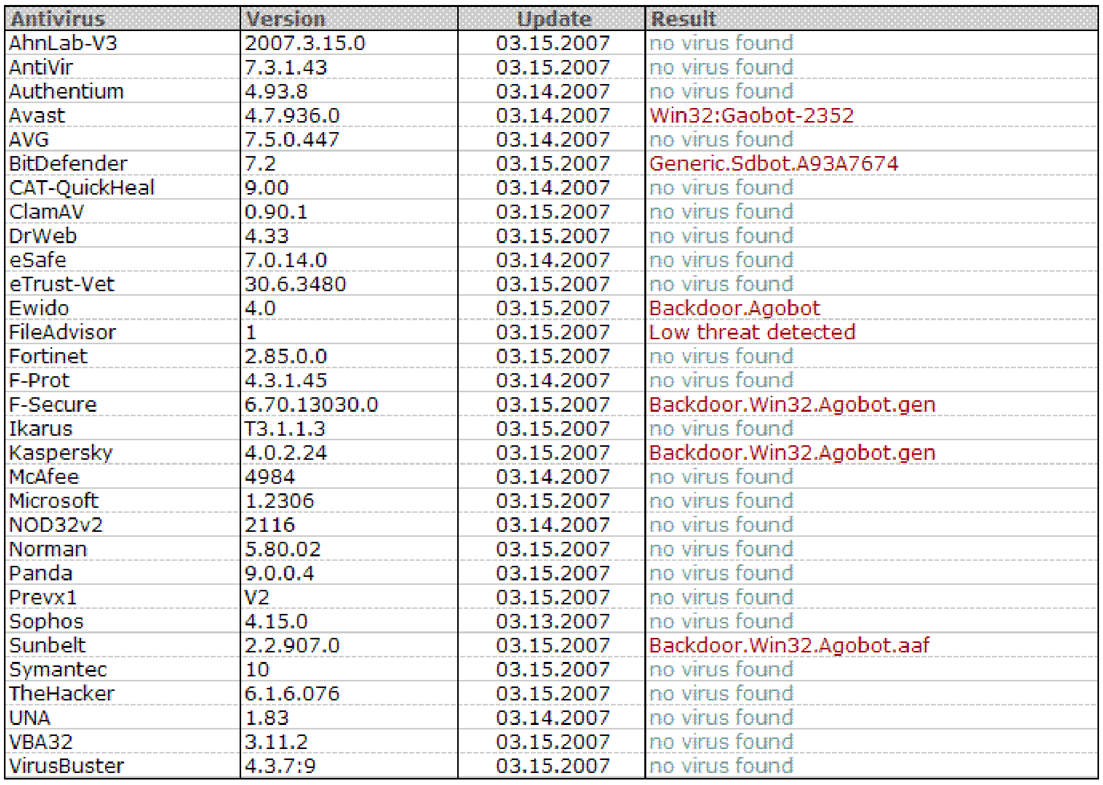
Anti-Forensics is described by Rogers (2006) as “attempts to negatively affect the existence, amount, and/or quality of evidence from a crime scene, or make the examination of evidence difficult or impossible to extract”. Kessler (2007) extends this definition in a practical sense by saying “anti-forensics, then, is that set of tools, methods, and processes that hinder such analysis”. The movement towards the employment of anti-forensic techniques in malware could be attributed to the substantial illicit financial gain that can now be achieved from employing malware nefariously (Larsson, 2007; Newman, 2006; Sukhai, 2004; Team Cymru, 2006).

Commercial Anti-Virus (AV) software is often limited in its ability to detect and remove malware (Chouchane, Walenstein, & Lakhotia, 2007; Mila Dalla, Mihai, Somesh, & Saumya, 2008; Xuxian, Xinyuan, & Dongyan, 2007; Yin, Song, Egele, Kruegel, & Kirda, 2007; Zhang, Reeves, Ning, & Purushothaman Iyer, 2007; Zhou & Meador Inge, 2008). This is essentially because AV software relies on an analyst having already analyzed collected malware, extracted a signature and made computer virus signature files available to the users of the AV software through very regular updates. Hence, AV software is highly unlikely to detect new malware that is unleashed on the internet, corporate intranet or that has been customized to target specific networks because it has not been previously analyzed and had a signature extracted (Masood, 2004).

1.2. A STATEMENT OF THE PROBLEM

There is a positive feedback loop between malware developers and malware researchers. As soon as a strategy is developed by one side, the other side implements a counter measure. Security professionals in the field need to know how to determine if they are the target of an attack, what the functionality of malware infections is and how to eradicate infections from their systems. This is especially true if a signature does not exist and the forensic analyst is required to analyse the instance of malware. The analysis process can be assisted if the analyst has up to date methodologies and skill sets at their disposal.

Virus Total provides a web-based, free and independent service that uses multiple anti-virus engines to analyze suspicious files that have been uploaded to their site. Virus Total (2007), on their website, state that “Currently, there is not any solution that offers a 100% effectiveness rate for detecting viruses and malware”. In support of this statement, Figure 1-1 shows the results that were captured from submitting a potentially harmful web robot (bot) that was collected from the ECU *Nepenthes* sensor network, to the Virus Total service.



Antivirus	Version	Update	Result
AhnLab-V3	2007.3.15.0	03.15.2007	no virus found
AntiVir	7.3.1.43	03.15.2007	no virus found
Authentium	4.93.8	03.14.2007	no virus found
Avast	4.7.936.0	03.14.2007	Win32:Gaobot-2352
AVG	7.5.0.447	03.14.2007	no virus found
BitDefender	7.2	03.15.2007	Generic.Sdbot.A93A7674
CAT-QuickHeal	9.00	03.14.2007	no virus found
ClamAV	0.90.1	03.15.2007	no virus found
DrWeb	4.33	03.15.2007	no virus found
eSafe	7.0.14.0	03.14.2007	no virus found
eTrust-Vet	30.6.3480	03.15.2007	no virus found
Ewido	4.0	03.15.2007	Backdoor.Agobot
FileAdvisor	1	03.15.2007	Low threat detected
Fortinet	2.85.0.0	03.15.2007	no virus found
F-Prot	4.3.1.45	03.14.2007	no virus found
F-Secure	6.70.13030.0	03.15.2007	Backdoor.Win32.Agobot.gen
Ikarus	T3.1.1.3	03.15.2007	no virus found
Kaspersky	4.0.2.24	03.15.2007	Backdoor.Win32.Agobot.gen
McAfee	4984	03.14.2007	no virus found
Microsoft	1.2306	03.15.2007	no virus found
NOD32v2	2116	03.14.2007	no virus found
Norman	5.80.02	03.15.2007	no virus found
Panda	9.0.0.4	03.15.2007	no virus found
Prevx1	V2	03.15.2007	no virus found
Sophos	4.15.0	03.13.2007	no virus found
Sunbelt	2.2.907.0	03.15.2007	Backdoor.Win32.Agobot.aaf
Symantec	10	03.15.2007	no virus found
TheHacker	6.1.6.076	03.15.2007	no virus found
UNA	1.83	03.14.2007	no virus found
VBA32	3.11.2	03.15.2007	no virus found
VirusBuster	4.3.7:9	03.15.2007	no virus found

Figure 1-1 Screen shot from Virus Total showing low detection rate of submitted bot after examination by thirty one different AV engines.

Out of the thirty-one antivirus programs that had the bot submitted to them, only six detected the bot, whilst one detected that there was a low threat present. This is not a particularly unusual result, evidenced from this research and supported by other researchers (Bilar, 2005; Masood, 2004; Mohandas, n.d.; Skoudis & Zeltser, 2004; Szewczyk & Brand, 2008; Wysopal, 2009). An analysis of the *Win32.Qucan.a* worm, by Mohandas (n.d., p. 20), found that only 50% of the antivirus detection engines were able to detect the worm that he submitted. This finding is well supported by the researchers with indication that the situation is deteriorating. Masood (2004) claims that the percentage of malware that avoids automated detection is growing every day and “manages to wreak havoc on networks”. Skoudis and Zeltser (2004, p. 108) emphasize that with new and fast spreading malware, most computer users would not be able to download a virus definition fast enough to stop them. Rubenking (2007) says that a solution needs to be found where malware can be recognized and cleaned up whilst not interfering with legitimate programs or interfere with the normal operation of the computer.

Part of the problem is that AV software relies on detecting signatures of malware that has already been analyzed by AV researchers and that the user has already downloaded the latest AV signatures to protect their computers. If newly released and unanalyzed malware is loaded onto a computer, it is highly unlikely that the malware will be detected because a signature will not exist. This undoubtedly can be classified as an incident. "An incident can be thought of as a violation or imminent threat of violation of computer security policies, acceptable use policies, or standard security practices" (NIST, 2004, pp. 2-1). An appropriate strategy and priority must be assigned for the incident. NIST (2004, pp. 3-17) lists a number of criteria for the determination of a suitable strategy, which include consideration of:

- Potential damage to, and theft of resources.
- Need for evidence preservation.

This information is important if the incident is reportable to the appropriate authorities and to assist in risk mitigation. However, the difficulty of obtaining this information must be taken into account. Malware uses a variety of techniques to avoid analysis. This is because there is an increasing profit motive for malware authors whose intention is to keep their malware undetected on computers (Dunham, 2006; Holt, 2007; Schiller et al., 2007; Sukhai, 2004; Team Cymru, 2006).

A significant body of knowledge is required to obtain this information from manual analysis, to either develop an AV signature or to determine the functionality of the malware (Valli & Brand, 2008). A short, non-exhaustive, requisite skills list for Windows-based malware analysis indicated by Valli & Brand could include:

- Assembly language programming.
- Program debugging skills.
- Static analysis techniques.
- Dynamic analysis techniques.

- Windows Applications Programming Interface (API) programming.
- Windows Operating System.
- Computer networking skills.
- Malware techniques.
- Reverse engineering skills.

Automated and semi-automated tools exist to assist in this analysis, but malware can detect these tools and alter its behaviour to hide its presence and/or modify its behaviour to not show its true intentions. This is exemplified by the research of Lau and Svajcer (2008). These researchers found that families of malware will adapt its behaviour if it detects it is running in a virtual machine by stopping execution or will run an alternate payload to deceive the forensic analyst. This is because forensic analysis of malware is often performed from within virtual machines. The advantage of using virtual machines for analysis is that they can be reverted to a known state very quickly. This is especially useful for analyzing malware that employs deception. If a deceptive path is executed that adds no value to the analysis but corrupts the host that is running the malware, the host can be reverted back to a known state and analysis continued down an alternate path of execution.

There is evidence that malware writers are targeting specific organizations such as banks. Larsson (2007) claims to have interviewed the creator of the Haxdoor Trojan, which was purportedly used to steal eight million Swedish Kronor from the Nordea bank. The significant issue raised in the article is that the creator of the virus is offering to create and sell customized versions of his malware so that users can steal money from accounts from the bank of their choice. He also offers support to achieve this, such as provision of servers for saving the stolen account information, in a non-traceable way. This sort of supported, targeted attack is not unprecedented. Dunham (2006, p. 11) reports that, in May 2005, an Israeli programmer was arrested for customizing and selling a Trojan horse, called *Hotworld*, to steal proprietary data from specified targets. At least eighty companies were implicated, including private investigation firms. This is significant because it is highly unlikely that such customized malware will be detected by AV software and in addition, that companies are prepared to pay for stolen information.

1.3. RESEARCH QUESTIONS

The objective of this research has been to find answers to the following research questions:

1. What techniques can malware use to avoid being analyzed?
2. How can the use of these techniques be detected?
3. How can the use of these techniques be mitigated?

Research question one seeks to find out what techniques malware can use to hinder the forensic analyst from fully analyzing it. The objective of the malware is to prevent full discovery of its malicious intent by using deception and obfuscation.

Research question two seeks to determine how the use of these techniques by malware can be detected by an analyst. This information is of value to the forensic analyst so that an appropriate strategy or methodology can be employed to counter the use of the technique. This information could also be of value to the forensic analyst to find evidence of intent to deceive or hide malicious intent.

Research question three seeks to ascertain how the use of these techniques can be mitigated so that analysis can proceed beyond the engagement of the analysis avoidance technique in the code so that discovery of the true intent of the malicious program can be determined.

1.4. SIGNIFICANCE OF RESEARCH

This research contributes to the body of knowledge for malware forensic analysis with particular emphasis given to the advancement of the analysis of the anti-analysis capability of malware.

The conduct of this research shows that there is a very large variety of anti-analysis techniques malware can incorporate to hinder analysis and avoid detection. This was determined primarily from a search of the literature and from validating the techniques in small, standalone programs and observing the effect on common analysis tools such as debuggers using quasi-experimentation. A taxonomy of anti-analysis techniques was developed during the course of this research that amalgamates the classification of techniques from key papers.

The detection of anti-analysis techniques feature far less in the literature than does the discussion of the incorporation of anti-analysis techniques. Detection of anti-analysis techniques in code would not only assist the analyst in investigation of malicious intent and the attempt at deception, it also appears that detection of anti-analysis techniques may be a very good indicator that the code has a malicious intent. This research supports other researcher's claims that existing AV software, that uses signatures and heuristics, is less than ideal at detecting malware, especially malware that has not been analyzed before. Analysis of network based malware collected from the internet for the purposes of this research is shown to nearly all contain a measure of anti-analysis techniques. Hence this research supports a new paradigm for AV software to rely less on signature detection and to be focused more on the detection of anti-analysis techniques as a good indicator that program under investigation is malicious.

Plugins exist for popular debuggers to hide its presence from discovery by malware that can incorporate anti-analysis techniques. These plugins focus primarily on hiding the presence of the debugger and ordinarily do not log or notify the analyst of the presence of anti-analysis techniques in the code that is being analyzed. This is a significant omission if malicious intent is being investigated, because it will simply not be logged. This research shows that the coverage of mitigation techniques of plugins is much less than the number of anti-analysis techniques that are available. This is significant because a false sense of security from using the plugins may lead to the analyst not conducting a thorough analysis of the malware and being the subject of deception. This suggests a deficiency in existing tools.

A variety of scripting languages and Application Programming Interfaces (API) exist to extend popular debuggers. This research shows how they can be incorporated to successfully detect and mitigate the use of anti-analysis techniques. Given the claim by this research that existing plugins have severe limitations due to their lack of coverage of anti-analysis techniques and lack of logging functionality, scripting of debuggers is an essential skill required for analyzing malicious software. In addition, this research shows that the extent of knowledge required to analyze malware is extensive. A proposed Malware Analysis Body of Knowledge was initiated by the conduct of this research where the treatment of anti-analysis techniques is a key and vital component.

This research examines some of the more well known methodologies for analyzing malware. A suitable methodology that detects the presence of anti-forensic techniques during the analysis process and then mitigates the technique has been identified through the conduct of this research.

This research could also prove to be of benefit to software engineering where requirements dictate that the Intellectual Property (IP) of the software has to be protected from reverse engineering. An understanding of the anti-analysis techniques discussed in this thesis that can be used to hinder the reverse engineering of code could assist in validating such a requirement through Test and Evaluation (T&E).

1.5. STRUCTURE OF THIS THESIS

Chapter 1 of this thesis presents an overview of this thesis, a statement of the problem, the research questions this thesis addresses and highlights the significance of this research for the digital forensic investigator. Malware invariably incorporates anti-forensic techniques and AV software cannot be relied upon to detect the presence of malicious code. This necessitates the development of an appropriate methodology to reveal the true intent of malware.

Chapter 2 provides a review of the literature. It establishes the foundation for this research by defining key terminology, models, classifications, anti-forensic techniques, previous studies and models to reveal and support lines of enquiry not discussed in the literature.

Chapter 3 justifies the selection of the most appropriate research method, the conceptual framework and research design to address the research questions. The selected research method to address the research questions is positivist, empirical and quasi-experimental. Two lines of experimentation were identified. The first was to implement a number of anti-forensic techniques in small, standalone programs to determine their effectiveness against the software tools likely to be employed by a digital forensic analyst. The second line was to analyze network based malware using anti-virus software.

Chapter 4 presents the results from conducting experiments with anti-forensic techniques. All of the techniques were found to be effective, and that the use of these techniques can be detected and mitigated.

Chapter 5 presents the results of having analyzed network based malware. The results support claims that anti-virus software is much less than ideal at detecting malicious software.

Chapter 6 provides discussion of the results and why the results are significant to the digital forensic analyst. Claims of contribution to knowledge are discussed together with an appropriate methodology that can be employed by analysts when anti-forensic techniques are encountered during their investigations and highlights the limitations of existing tools and anti-virus software. Further lines onf investigation are also identified.

Chapter 7 concludes the thesis by linking the claims of contribution to knowledge to the implications of this research.

CHAPTER 2 LITERATURE REVIEW

Code that protects itself from being analyzed is a significant hindrance, not only to automated malware detection tools such as AV software, but also to the manual analysis performed by malware analysts. The purpose of this literature review is to examine the literature related to:

- Characterization of network based malware
- Existing malware analysis methodologies.
- Anti forensic techniques used by malware to avoid analysis.
- Malware detection techniques.
- Packers and protectors.
- New paradigms for malware detection.

These lines of enquiry trace directly to the research questions.

2.1. CHARACTERISATION OF NETWORK BASED MALWARE

Malware presents itself as a significant threat to computer users. Various attack vectors exist as well as the number of malicious payloads that they can contain. Network based malware, such as worms, propogate autonomously via networks and do not propogate in the same fashion as viruses do. Network based malware was collected for the basis of this research, as discussed in the Conceptual Framework section of this thesis, subsection 3.6.2. For this reason, viruses are not included directly in the following discussion. Hence, this subsection introduces worms and how they propogate. It also discusses the various payloads of worms that can include, but not limited to, Trojans, Rootkits, Backdoors and Bots. It is important to note that the payload of worms, such as Bots, have evolved to incorporate anti-forensic techniques. The anatomy of a worm is presented to provide a greater insight into how they function, and how their payloads have evolved to include multiple threats and have become more stealthy to avoid detection. Current detection methods are also discussed.

2.1.1. Worms

Network worms can propagate to victim computers using a variety of methods. A summarized description of the propagation categories listed by Kaspersky Labs (2007b) is presented in Table 2-1.

Table 2-1 Summary of the propagation methods of worms.

Worm	Propagation Method
Email Worms	The worm could be an attachment to an email, and the worm is activated when the attachment is opened, or the email contains a link to an infected site. These worms spread though: • Windows Mail API (MAPI) functions • Microsoft Outlook Services • Directly to SMTP servers using code in the worm.
Instant Messaging (ICQ and MSN) Worms	Propagate using instant messaging applications to send links to entries in the contact list to infected sites.
Internet Worms	Spread by: • Copying to network resources • Exploitation of Operating System vulnerabilities • Penetration of services such as FTP and Web servers. • Take advantage of malware already installed to install the worm
IRC Worms	Utilizes contacts from the infected user to use Internet Relay Chat (IRC) channels to send links to infected websites or send infected files.
File-sharing Networks or P2P Worms	Uses the P2P network to download and execute infected files.

2.1.2. Trojans

A summarized description of the categories listed by Kaspersky Labs (2007a) is listed in Table 2-2.

Table 2-2 Summary of the malicious functionality of trojans.

Trojan	Functionality
PSW Trojan	Steal passwords and confidential information and send this information to a remote computer.
Trojan Clickers	Redirect infected machines to web sites to : • Increment the hit count of a site for the purposes advertising. • For organizing a Denial of Service (DoS) attack. • To redirect the victim to an infected site where the victims machine will be attacked by other malware.
Trojan Downloaders	Downloads and installs malware on the victim machine and most likely registers it to auto run without the consent or knowledge of the user.
Trojan Droppers	Consist of multiple payload components to install other malware onto the victim machine so that the installation of the additional components is hidden from the user, and perhaps to trick anti virus software which may not analyse the other components.
Trojan Proxies	Uses the infected machine to give the attacker anonymous access to the intenet. These machines can also be used by an attacker for mass mailing of spam.
Trojan Spies	Spy on user activity through the use of spy programs such as key loggers and forward the collected information to the attacker. Can be used to steal banking details and financial information for the purposes of fraud.
Trojan Notifiers	Notify the attacker that the machine has been infected via email, ICQ, or IRC.

2.1.3. Rootkits

Rootkits are used by an attacker to evade detection by replacing system files. Hoglund and Butler (2005, p. 4) say “a rootkit is a set of programs and code that allow permanent or consistent, undetectable presence on a computer”.

2.1.4. Backdoors

Contain a remote administration capability so that infected machines can be controlled remotely via a network connection, and may not be visible in the list of currently active programs (VirusList.com, 2009). Activities may include all the functionality listed above and may also

- Send and/or receive files
- Launch and/or delete files

2.1.5. Bots

The original intention of a robot (bot) was to perform some useful action on an IRC channel whilst the operator or user were engaged in some other task (Schiller et al., 2007, p.7). Bots are capable of taking action on a client machine without a hacker having to have logged onto the infected machine. A collection of Bots is known as a botnet. The botnet is typically under the command of a botherder who can dictate the actions of the botnet through a bot server. The botnet can be divided into divisions which can each be performing different actions, or if the communication channel to one division is lost, the other divisions can continue the mission. Bot clients are modular and adaptive and can be updated with new software, or commanded to perform a malicious action such as a DDoS against a target. The attacker may be distanced from the infected machine by many layers within the hierarchy. The attacker can send commands to an IRC channel through an obfuscating proxy and through multiple hops (Schiller et al., 2007, p.30).

A bot typically consists of a module to exploit a vulnerability to gain access to a target, another module to stop AV software and firewalls, a module to scan for other vulnerable systems, a module to exploit the system it is installed on, such as collecting passwords or keylogging and a module that communicates with a Command and Control Center (C&C). Not only is it important to remove the bot from an infected system, it is important to work out how the bot got onto the system in the first place so the vulnerability can be rectified.

Schiller *et al*. (2007, p. 24) describes botnet technology as the “next killer Web application” because organized crime have used it as a force multiplier to attack the non-computer literate, including the young and the elderly to derive money. Their discussion continues to say that these criminal organizations have grown large enough to become a “threat to major corporations and even nations”.

The evolution of botnets is important to understand. It shows how they have become more modularized and stealthy as time has progressed. Stealth is a critical component of anti-analysis techniques. The following sub sections discuss this evolution.

2.1.5.1. Evolution of Bots

PrettyPark (Anonymous, 1999) was the first bot client that made use of the IRC bot for the purpose of remote control over the internet, and emerged in June 1999 (Canavan, 2005, p. 6). It allowed the attacker to retrieve information from the compromised system and had a basic mechanism for updating itself by downloading and executing new files from IRC. Features of *PrettyPark* are still evident in IRC bots seen today. Features discussed by Canavan (2005, p. 6) include:

- The capability to determine system information such as the version of the operating system as well as the user and computer name.
- The ability to retrieve email addresses and login names to applications such as ICQ.
- The ability to retrieve network settings, user names and passwords.
- The capability of being able to download updates to increase its functionality.

Global Threat (GT) bots began to appear in late 2000 and made use of a Windows shareware IRC client called *mIRC* (Mardam-Bey, 1995) that include scripting capabilities that allowed hackers to put together their own scripts to connect to remote servers and await commands. GT bots also made use of tools such as HideWindow (Anonymous, n.d.-f) to conceal its presence on infected machines and used PsExec (Microsoft, 2008c) to spread itself over the local network. They also used FireDaemon (FireDaemon Technologies Ltd, 2009) to install and run as a service and IrOffer (iroffer.org, n.d.) to perform as a fileserver. These bots were launched as a service by altering the system startup files (Canavan, 2005, p.7). GT bot also had the capability to conduct a DDoS attack by flooding. It could spread itself also by using social engineering ploys including sending an email that claimed to be from a security vendor and if the user clicked on an embedded link they downloaded the bot client from a malicious website. GT bots were not modular, they were all contained within a single package (Schiller et al., 2007, p. 9).

SDBot (sd, 2002) appeared in 2002 and added the feature of a remote control backdoor (Schiller et al., 2007). The source code was made available by the author, as well as a Web page and contact information through email and ICQ. This made it easy for hackers to modify and maintain. Variants of *SDBot* can exploit the backdoors of other malware such as *SubSeven (Sub7Crew, n.d.)*, *Mydoom (Anonymous, n.d.-k)*, *Bagle (Anonymous, n.d.-b)*, *Kuang* (Anonymous, n.d.-h) and many others. When these backdoors are found *SDBot* downloads itself onto the client and infects it.

Agobot (Gembe, 2002) made use of modular design and appeared in 2002. It uses IRC for C&C, but is spread using P2P file sharing applications (Schiller et al., 2007, p. 11). It has three modules which retrieves the next module once the primary task of the module has completed. The sequence of events is as follows:

1. Delivers the IRC bot client and installs a remote access back door.
2. Attacks and shuts down AV processes.

3. Prevents the user from accessing Web sites including AV vendor sites.

Capabilities of *Agobot* discussed by (Schiller et al., 2007) include:

- Able to scan other computers for vulnerabilities.
- Capable of being able to launch DDoS attacks.
- Ability to scan for CD keys for games and software.
- Can terminate AV software and security monitoring processes.
- Can modify the host file so that updates will not be downloaded from AV software sites.
- Can install a rootkit to hide itself.
- Incorporates anti reverse engineering techniques to make analysis difficult.

Related bots include *Phatbot* (Gembe, 2002) which uses public key cryptography for communication with the C&C over P2P, *Polybot (Anonymous, 2004)*, *XtremBot* (Anonymous, n.d.-d) and *Forbot (Anonymous, n.d.-e)*. It is also worth noting that this is when the family lines of bots began to blur and variants appeared which took the best components of other bots and incorporated those features. It became harder to determine from which family a particular bot had evolved from (Canavan, 2005, p. 14). There are reports that AV vendors are becoming less concerned about identifying the particular bot because of the number of variants which have different capabilities (Schiller et al., 2007, p. 12). Instead they are looking at the malicious components of the bot as the source of identification.

Spybot (Anonymous, 2003) is a derivative of *SDBot* and appeared in 2003 as open source. It adds Spyware capabilities and collects email addresses, lists of visited web sites and logs of activities. Variants can also capture screen shots of the screen, send spam, install a rootkit, control webcams, kill security processes and other malicious acts. It spreads via file sharing applications, exploitation of known vulnerabilities and backdoors left by other malware.

RBot (Anonymous, n.d.-l) appeared in 2003 and is a backdoor Trojan which uses IRC to communicate with the C&C. It introduced the use of packers and protectors to compress and/or protect the malware. It can scan for shares on networks with Windows machines and attempts enumerate users and attempts to guess weak passwords.

Polybot is derived from the source code of *Agobot* and appeared in March 2004. It uses polymorphism to change its appearance of the packed and or protected binary for each infection by using a different key each time.

The *MyTob* (Diabl0, 2005) bot appeared in February 2005 and is a hybrid that uses its own SMTP engine for sending mass e-email to addresses in the Address Book of the infected computer and has capabilities similar to *Spybot*.

(Schiller et al., 2007, p. 15) lists a number of new features appearing as components for bots. These are summarised as follows:
GpCoder (Anonymous, 2005) – Encrypts a user's files and then offers to sell the user a decoder.
Serv-U – An FTP server that enables botherders to store stolen software, games, movies and illegal material on the botnets under their control. The data is stored in hidden directories, and the FTP server appears as Windows Explorer in Task Manager.
SPIM – Spam for Instant Messaging. Can be used for phishing attacks which provide links to Web sites that download malicious code to victim machines. An example SPIM message presented by Schiller et.al. (2007, p. 16) is reproduced in Figure 2-1.

```
ATTENTION...Windows.has.found.55.Critical.System.Errors...
To fix the errors please do the following:..
1 Download Registry Update from: www.regfixit.com.
2 Install Registry Update
3 Run Registry Update.
4 Reboot your computer
FAILURE TO ACT NOW MAY LEAD TO SYSTEM FAILURE!
```

Figure 2-1 Example Spam for Instant Messaging (SPIM) message to trick the user into downloading malware.

2.1.6. Blended Threats

It is very important to realize that modern malware combines numerous attack vectors and malicious payloads, such that simple classification of malware to an individual type or family is becoming more difficult. Such a combination of threats in an individual instantiation of a malware specimen is known as a blended threat. Virus Bulletin (2008) describes a blended threat as “a sophisticated attack using multiple malware types and vectors to carry out penetration and control of a system”. An example of a blended threat discussed by Virus Bulletin could be initiated be the receipt of a spammed email that contains a link to a hijacked web site that uses *`iframes`* running malicious javascript. The malicious javascript exploits vulnerabilities in the browser of the user which can then execute code on the users computer to disable security software and download additional malware. Functionality of the downloaded malware could be to run a spam e-mail server or to launch attacks against new victims.

2.1.7. Anatomy of a Worm

Skoudis and Zeltser (2004) describe the anatomy of a worm with an analogy to a rocket with the following components:

Warhead – Contains exploit(s) to take advantage of vulnerabilities in software to penetrate a target.
Propagation Engine – Mechanism(s) to propagate itself to other vulnerable machines.
Target Selection Algorithm – An algorithm to select or search for vulnerable machines.
Scanning Engine – An algorithm and code that searches for machines that run software that is known to be exploitable, using the code available in the warhead.
Payload – Contains the individual malicious packages that are installed on the target machine such as a keylogger, web server, backdoor, firewall and AV disabler and so on.

Figure 2-2 is adapted from Skoudis *et al.* (2004). and depicts the components of the BugBear.B worm.

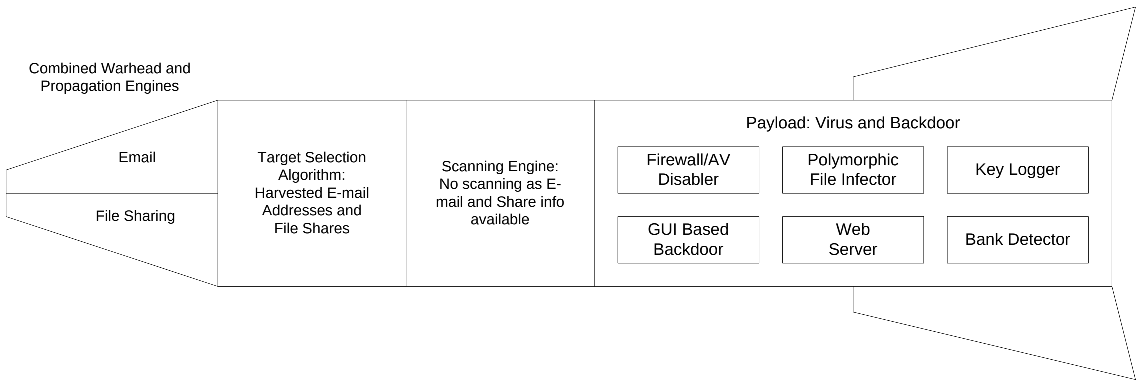

Combined Warhead and Propagation Engines
Email
File Sharing
Target Selection Algorithm: Harvested E-mail Addresses and File Shares
Scanning Engine: No scanning as E-mail and Share info available
Payload: Virus and Backdoor
Firewall/AV Disabler
Polymorphic File Infector
Key Logger
GUI Based Backdoor
Web Server
Bank Detector

Figure 2-2 Anatomy of BugBear.B showing modular nature of the worm.

The representation of the BugBear.B worm emphasizes the modular nature of modern malware. Different components and sub components can be plugged in or out depending upon the requirements and intention of the attacker. Trend Micro Incorporated, a major AV software vendor, recognizes that blended threats are increasingly being seen on the internet and predicts that malware will increasingly use tricks to avoid detection (Trend Micro Incorporated, 2007).

2.1.8. Defence Methods

Defence methods against malware are typically based on some combination of the following methods (Farwell, 2004):

Signatures - recognition of signatures of known and previously analyzed malware.

Heuristics – flagging of anything outside the normal operating parameters of the system.

Integrity – detection of changes to the integrity of known files.

The typical computer user runs a signature based virus checker that should download new signature files every day to help protect them from compromise. However, this is far from a complete solution as it relies on the signature of the malware being present in the updated signature file. If the malware that is attacking a computer system is new to the internet, or custom written to attack identified targets, it is highly likely that the updated signature file will not protect the target (Masood, 2004).

Before a signature is extracted and uploaded to the client machines and added to the virus signature database, the malware must be analyzed by a malware analyst to determine what the functionality of the malware is, what changes it will make to system files and how it will change the normal behaviour of the machine. The extent of infection must be determined to ensure that infected files are removed or repaired. If the malware has detected it is being analyzed and has not shown its true intent by not unpacking and installing all of the files it was going to install, then the full extent of the infection will not be determined to the detriment of the end user who requires protection. The first step in this analysis process is referred to as profiling.

2.2. PROFILING

Initial examination of collected malware is called profiling (Aquilina, Casey, & Malin, 2008, p. 286). Profiling of malware is conducted from a high level of perspective to determine the purpose and functionality of the malware. This assists in making an informed decision on how to proceed with a more detailed analysis. There are two general types of file profiling that can be conducted, namely static analysis and dynamic analysis.

2.2.1. Static Analysis

Static analysis extracts information about the binary code without actually running the code. It can include examination of disassembly listings, extraction of strings, obtaining a virus signature, determination of the target architecture and compiler used, as well as many other characteristics

Static analysis of disassembly listings of binary code can be technically difficult. A disassembly of binary code is a textual file that represents the assembly language code of a program. A program is a series of instructions and data that a computer executes to perform some series of functions. The series of instructions and structures of data can be analysed without executing the program. This gives the analyst the ability to explore various possible paths of execution that can take place when a program runs. These different paths of execution are referred to as control flow graphs and consist of nodes and edges, where nodes consist of basic blocks of code and edges interconnect the nodes as potential control flow paths. Control flow can be dictated by constructs including conditional blocks, switch blocks and loops. Dataflow analysis examines the way data is moved and changed throughout the execution of a program (Chess & West, 2007).

2.2.2. Dynamic Analysis

Dynamic analysis extracts information about the code by observing what it does whilst it is running. This can include network communications, file and registry access and modification, interaction with services and other behavioral activities. Dynamic analysis gives consideration to the services to provide or emulate for the network based malware to interact with so that its dynamic behaviour can be observed. The malware that arrives on the system may simply be the first stage in a process that attempts to download the real payload in a second stage. This is known as a dropper. Arnold, Chess, Morar, Segal, & Swimmer (2000) recommend that the following services may need to be provided through emulation, or via a real service, to give the network based malware the opportunity to behave in the environment it would expect on a real network.

HTTP – Malware may try to transfer files from HTTP, through javascript, or some other scripting language. Typically this is port 80.

FTP – Malware may try to transfer files. Typically this is port 21.

IRC – Bots, in the past, typically used IRC for communications. P2P is becoming more popular for communications. Typically, IRC uses ports in the ranges of 6660-6669, but malware can use any unused port.

DNS – Malware may seek to look up an address in DNS. Typically this is port 53.

Drive sharing – Malware may look for shared drives. Typically this could include ports 135, 137 and 445.

Email – Malware may look for mail services, typically on port 25.

Packet routing – Malware may try to route packets through various network devices.

Skoudis (2004, p. 595) outlines a model where analysis tools are distributed on a local victim machine and on an external machine, to capture behavioral aspects of the malware on the local machine and its interaction with external services over a network. External services as outlined by Arnold *et al*. (2000) can be setup on the external monitoring segment. A possible model for malware monitoring is shown in Figure 2-3. It shows that the malware is installed on a local machine together with local file, registry and process monitoring tools, debugger and local network monitoring. Externally provided tools include a port scanner and vulnerability scanner to see if the malware has opened up ports, or exposes a particular vulnerability that may only be visible from an external computer. This is because malware can hide the presence of open ports on the victim machine and they can only be seen externally. A sniffer is a useful addition to the external network to detect the types of network communications that are initiated by the malware, including attempts to resolve names from a DNS server, attempts to establish connections to an IRC server, scans for computers that are sharing drives, or mail servers.

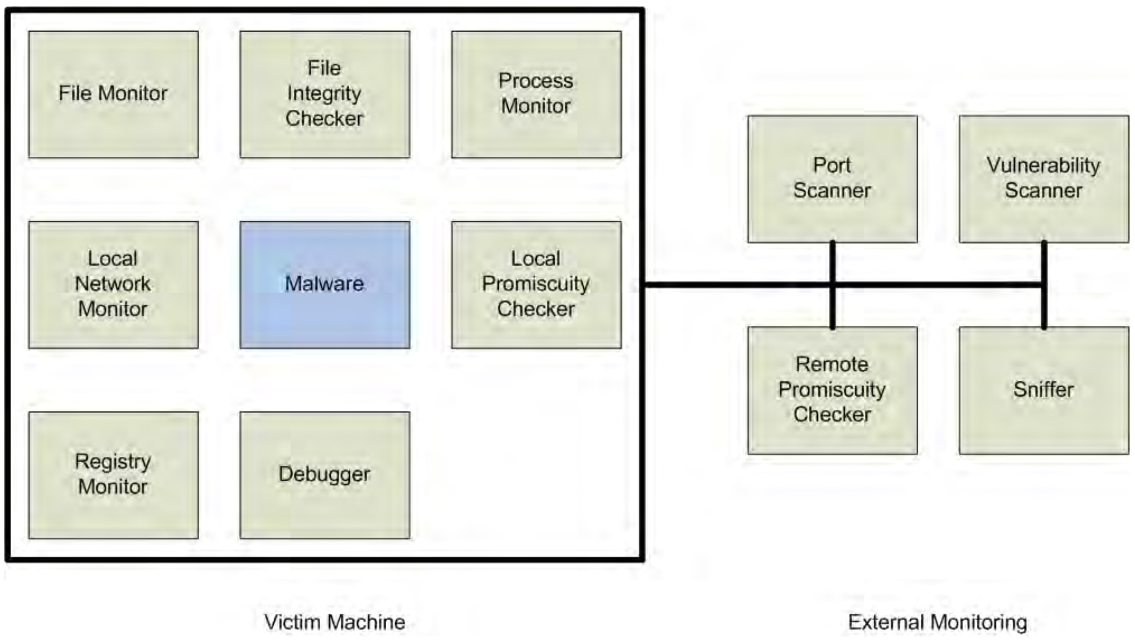


Figure 2-3 Possible model for deployment of analysis tools for monitoring malware on victim machine and via external monitoring.

A summarized list of the analysis tools recommended by Skoudis (2004, p.568) as well as their purpose and analysis type, is shown in Table 2-3.

Table 2-3 Summary of malware analysis tools showing analysis type, purpose and name of commonly used tool name.

Analysis Type	Purpose	Tools
Static analysis	Use as many antivirus detection engines as possible to assist classification.	*VirusTotal* (Virus Total, 2008)
Static analysis	Search the body of the malware for strings.	*Strings* (Microsoft, 2008c)
Dynamic analysis	File integrity check to record baseline configuration.	*Winalysis* (Winalysis.com, 2008)
Dynamic analysis	File monitoring. Find which tools are opening, reading and writing files.	*Filemon* (Microsoft, 2008c)
Dynamic analysis	Process monitoring. Determine resources that are being used such as DLL's and registry keys.	*Process explorer* (Microsoft, 2008c)
Dynamic analysis	Network monitoring. Uncover which ports are open, collect network traffic and find vulnerabilities.	*Fport* (Foundstone, 2008), *tcpview* (Microsoft, 2008c), *nessus* (Tenable Network Security, 2008), *nmap* (Insecure.org, 2008), *wireshark* (Combs, 2008), and *snort* (Sourcefire, 2008).
Dynamic analysis	Registry monitoring. Monitor registry activities as they occur.	*Regmon* (Microsoft, 2008c)
Code analysis	Disassembly, debugging	*IDA Pro* (Hex-Rays, 2008) , *OllyDbg* (Yuschuk, 2008) .

2.3. OVERVIEW OF COMMON MANUAL ANALYSIS METHODOLOGIES

A manual, step by step, analysis process suggested by Skoudis (2004, p.573) for analysis of malware that incorporates static and dynamic analysis techniques has been reproduced in the following list:

- Load specimen onto victim machine.
- Run antivirus program.
- Research antivirus results and filenames.
- Conduct strings analysis.
- Look for scripts.
- Conduct binary analysis.
- Disassemble code.
- Reverse compile code.
- Monitor files changes.
- Monitor files integrity.
- Monitor process integrity.
- Monitor local network activity.
- Scan for open ports remotely.
- Scan for vulnerabilities remotely.
- Sniff network activity.
- Check promiscuous mode remotely.
- Monitor registry activity.
- Run code with debugger.

The methodology of Skoudis (2004) is fairly linear in nature, after one step is completed, the next step is entered. It does not explicitly seek to mitigate the use of anti forensic techniques the malware may be using to hide its presence, alter the program flow, or detect the presence of analysis tools.

A generalized approach to profiling listed by Aquilina, Casey and Malin (2008, p.286) is listed and summarized as follows as a series of steps that may be conducted in a particular order:

Detail – Document the system details from which the suspect file was obtained.

Hash – Determine the cryptographic hash of the suspect file.

Compare – Conduct a similarity test against known samples.

Classify – Identify the target platform, high level language of the specimen and the compiler used.

Scan – Identify the language used to author the code as well as the compiler used, the type of file and target architecture.

Examine – Use executable file analysis tools to try to determine if the suspect file has malicious intent.

Extract and Analyze – Extract strings, file metadata and symbolic information.

Reveal – Identify armoring techniques that will protect the suspect file from examination.

Correlate – Determine if the file is statically or dynamically linked.

Research – Determine if the file has already been analyzed by conducting online research.

This list explicitly has a step to reveal armoring techniques that malware can use to hinder analysis which is not listed by Skoudis. The work by Skoudis (2004) precedes the list by Aquilina *et al*. (2008) by approximately four years and may indicate that the use of anti-analysis techniques employed by malware has become more prevalent during this time and that these techniques have to be mitigated before analysis can proceed.

A significant work by Zeltser (2007) is very much, a comprehensive, manual analysis treatise. It is in the form of a training course conducted by the SANS organization and is appropriately titled “Reverse-Engineering Malware: Tools and Techniques – Hands On”. Zeltser begins by setting up a safe, laboratory environment, using freely available software tools. The general methodology presented by Zeltser (2007, pp. 1-12) is listed as follows:

1. Run the malware in an isolated laboratory
2. Monitor the interactions between the system and the network from a behavioral sense.

3. Understand the program’s code
4. Repeat the process until enough information is gathered.

What becomes evident throughout Letzer’s (2007) notes and the practical exercises, is the iterative and recursive nature of this methodology. This is in contrast to the linear methodology of Skoudis (2004). Starting points, or clues, are extracted from the malware from static and dynamic analysis and these are used to focus on the aspects of the code that have malicious functionality. This approach is often referred to as “hit listing” in reversing and analysis literature because it is often infeasible to fully analyze a malware specimen from the perspective of time that can be expended to this endeavor. In fact, it is to the malware writers’ advantage to make the code as difficult and time consuming to analyse as possible. The analyst may not be able to spend as much time analyzing the code as they would like. This could lead to missing the opportunity to analyse important and relevant sections of code.

As this information is extracted, the investigative environment is adapted, such as adding entries to the hosts file, addition of an IRC client or server, mail server or whatever else the malware expects to connect to. Then the behavioral analysis can begin again, with the new information, to delve deeper into the malware to reveal its intentions and how it works. The iterative and recursive nature also lends itself to dealing with anti-analysis techniques as they arise and could be a superior methodology to adopt to detect and mitigate anti-analysis techniques, especially in the case where detection of the use of anti forensic techniques is an objective.

2.4. OVERVIEW OF ANTI FORENSIC TECHNIQUES

“Digital forensics includes preserving, collecting, confirming, identifying, analyzing, recording and presenting crime scene information” (Kleiman, 2007, p. 9). Malware is increasingly being used to commit cyber crime (Trend Micro Incorporated, 2007) and digital forensics are applied by investigators to achieve this objective. However, techniques to thwart the digital forensic analyst are employed by maware developers.

> Today as computer intruders become more cognizant of digital forensic techniques, malicious code is increasingly designed to obstruct meaningful analysis. By employing techniques that thwart reverse engineering, encode and conceal network traffic, and minimize the traces left on file system, malicious code developers are making both discovery and forensic analysis more difficult. This trend started with kernel loadable rootkits on UNIX and has evolved into similar concealment methods on Windows systems. Today, various forms of malware are proliferating, automatically spreading (worm behaviour), providing remote control access (Trojan horse/backdoor behaviour), and sometimes concealing their activities on the compromised host (rootkit behaviour). Furthermore, malware has evolved to undermine security measures, disabling AntiVirus tools and bypassing firewalls by connecting within the network to external command and control servers. (Aquilina et al., 2008, p. xxxv)

An important consideration in the analysis of malware is that anti forensic techniques are increasingly being employed by developers of malware to avoid detection and analysis of their code (Brand, 2007; Falliere, 2006, 2007; Ferrie, 2008; Grugq, n.d.; Harbour, 2007; Smith & Quist, 2006). It was reported in an online article that a speaker at the Australian IT Security in Government Conference claimed that 65% of new malware “uses some type of stealth or anti-forensic technology in an attempt to remain undetected before, during and after an attack” (Kotadia, 2006).

Malware employs anti forensic techniques to prevent the forensic analysis of its behaviour and its underlying code. This is achieved by detecting the use of popular analysis tools and debuggers. Once detected, the malware can modify its behaviour so that it does not perform its malicious action from a dynamic analysis point of view. From a static analysis point of view, it can use numerous techniques to make the static analysis difficult and hide its true nature.

An example presented by Yason (2007, p. 12) has been adapted and modified by the researcher in Figure 2-4 with comments. It uses the `FindWindow()` function from the `user32` Dynamic Link Library (DLL) to identify if the popular debuggers, *`WinDbg`* (Microsoft, 2008b) or *`OllyDbg`* are running. If malware detects the presence of a debugger, it can amend its behaviour so that it does not perform malicious activities, remove itself from the system, or, with appropriate privileges, damage the system.

```
; set up the call to FindWindow to find OllyDbg
push  NULL
push  .szWindowClassOllyDbg
call  [FindWindowA]

; check the result of the call
test  eax,eax

; if the result is non zero, the debugger was found,
; so jump to the section of code to display a message box
; note that this is not in this snippet of code
jnz   .debugger_found

; set up the call to FindWindow to find WinDbg
push  NULL
push  .szWindowClassWinDbg
call  [FindWindowA]

; check the result of the call
test  eax,eax

; if the result is non zero, the debugger was found,
; so jump to the section of code to display a message box
; note that this is not in this snippet of code
jnz   .debugger_found

; data
.szWindowClassOllyDbg db    “OLLYDBG”, 0
.szWindowClassWinDbg  db    “WinDbgFrameClass”, 0
```

Figure 2-4 Partial implementation of FindWindowA function to find popular debuggers (Yason, 2007, p. 12)

Another example by Yason (2007, p. 14), reproduced in Figure 2-5 checks for the presence of breakpoints by scanning for the byte *0xCC* (which represents a breakpoint) in a region of protected code as defined by the region:

```
Protected_Code_End – Protected_Code_Start
```

The protected code could be within a region of packed code that is unpacked by a runtime packer. A packer compresses and/or encrypts an executable program (which may or may not be malware) and creates a new executable binary file. The packed program includes a runtime unpacking stub which unpacks the original program into its original state and transfers control to the original program. Packers may use software protection mechanisms such as anti debugging, anti virtual machine, exception handling and control flow handling to hinder analysis (Sun, Ebringer, & Boztas, 2008).

```
cld
mov       edi,Protected_Code_Start
mov       ecx,Protected_Code_End – Protected_Code_Start
mov       al,0xCC
repne     scasb
jz        .breakpoint_found
```

Figure 2-5 Partial implementation of code to detect breakpoints (Yason, 2007, p.14)

Most of the literature that discusses anti-analysis techniques only provides code snippets to accompany explanatory text. These snippets can be incorporated into working code for validation purposes to assess the effectiveness of the technique. Work on validation of a subset of these techniques has been conducted by the researcher and the results are documented in Chapter 4 of this thesis. A very general, overarching taxonomy of anti-analysis techniques, revealed through a search of the literature (Aquilina et al., 2008; Brand, 2007; Grugq, n.d.; Skoudis & Zeltser, 2004; Zeltser, 2007), includes the following:

- Anti virtual machine
- Anti online analysis engines
- Anti unpacking
- Process injection techniques
- Code execution from memory
- Checksum checks
- Process camouflage
- Structured exception handling
- Import Address Table
- Rootkits
- Packers and Protectors

These techniques are discussed in the following sections.

2.5. ANTI VIRTUAL MACHINE

Analysis of malware is recommended by Zeltser (2007, pp. 1-20) to be performed on Virtual Machines such as *VMWare* (VMware, 2008) or *Virtual PC* (Microsoft, 2007). This allows multiple virtual machines to be run on the one physical machine, all of which can be networked and can each be running a different operating system. These virtual machines can also be backed up and restored very quickly and easily. This makes an ideal environment for the analysis of malware where a known state or checkpoint can be returned to, and the analysis restarted if required.

However, malware can use techniques to determine if it is running in a virtual machine as demonstrated by the logic of the following pseudo code reproduced from a presentation by Smith and Quist (2006) as Figure 2-6.

```
IF detect_vmware
        THEN do nothing, destroy self, destroy system
ELSE
        Continue with malware payload
```

Figure 2-6 VMWare detection pseudo code showing that if VMWare is detected, the machine could be damaged (Smith & Quist, 2006).

Eagle (n.d.) reports that *VMware* uses a registry key for the installation location of *Vmware* as:
`HKLM\Software\VMware, Inc.\VMware Tools\InstallPath`

Malware can look for the presence of this key to indicate that it could be running in a virtual machine. Another technique Eagle points out, is to use the Windows Management Instrumentation (WMI) to iterate though the network interfaces to see if any of the MAC addresses used belongs to VMware. Eagle suggests the following to mitigate this technique:

- Uninstall *VMware* tools.
- Change the MAC address of the virtual adapter in the guest OS.

Innes and Valli (2006) point out that VMWare, in its default configuration, is very easy to detect through a listing of the hardware and its reported type. The types listed by Innes *et al*. are reproduced in Table 2-4.

Table 2-4 Default Hardware Configurations used to find presence of VMWare (Innes & Valli, 2006)

Hardware	Reported Type
Video Card	VMWare Inc [VMWare SVGA II
Network Interface Card	Advanced Micro Devices [AMD] 79c970 [PCnet 32 LANCE] (rev 10)
Hard Disk	VMWare Virtual IDE Hard Drive
CD Drive	NECVMWar VMWare IDE CDR10
SCSI Controller	VMWare SCSI Controller

Innes *et al*. (2006) also lists the three MAC addresses assigned to the virtual network cards as one of the following three values and this can be detected by running either `ipconfig /all` or by running the command arp –a and scanning the result.

```
00-05-69-xx-xx-xx
00-0C-29-xx-xx-xx
00-50-56-xx-xx-xx
```

Innes *et al*. (2006) also point out that VMWare developers left a backdoor open for the configuration of the virtual machine during runtime with the following lines of assembly code that have been reproduced from their paper as follows in Figure 2-7.

```
mov eax, VMWARE_MAGIC ; 0x564D5868
mov ebx, b ; <parameter of command>
mov ecx, c ; <number of command>
mov edx, VMWARE_PORT ; 0x5658
in eax, dx
```

Figure 2-7 Code snippet used to detect the presence of VMWare (Innes & Valli, 2006)

A sample of the commands listed by Innes *et al*. are reproduced in Figure 2-8.

```
04h Get current mouse cursor position.
05h Set current mouse cursor position
06h Get data length in host's clipboard.
07h Read data from host’s clipboard
08h Set data length to send to host's clipboard.
09h Send data to host’s clipboard
0Ah Get VMware version
0Bh Get device information
```

Figure 2-8 Commands that can be used to detect the presence of VMWare (Innes & Valli, 2006)

Innes *et al*. (2006) point out that a VMWare machine could be detected if running this code was successful and a result was returned from the function call. Smith et.al. (2006) provide additional techniques to detect Vmware and Virtual PC.

Porras, Saidi & Yegneswaran (2007, p.7) note that recent versions of *Storm* appear to have stopped checking to see if it is running inside a virtual machine and is instead focusing on hiding themselves from monitoring software. The significance of this comment is that the developers of *Storm* have evolved their malware beyond detecting the presence of a virtual environment. Possibly, this could be because of the trend for organizations to use virtualization to host their servers. If the simple approach of the malware is to not install itself on a virtual machine, an opportunity may be lost to it if it tries to install itself on a virtual machine that is not an analysis environment, but a real, business orientated, virtual machine. By loading their own drivers (sys files), they can be notified when a program or driver in an undesired list is launched. This takes the malware to a lower layer, underneath the radar, beneath where the virtual machine runs. This is done via a call to the Windows API function `PsSetLoadImageNotifyRoutine()`. The list of executables disabled by *Storm* is quite extensive and listed in the Appendix of the paper by Porras et. al. (2007). The list includes spyware detection programs, virus scanners and anti spyware programs. This is a problem because it provides a vector to detect and mitigate the tools of a forensic analyst as well.

2.6. ANTI ONLINE ANALYSIS ENGINES

Anubis (International Secure Systems Lab, Vienna University of Technology, Eurecom France, & UC Santa Barbara, 2008) is an online malware behavioral analysis service. Online analysis engines automate the dynamic analysis process as discussed above. Malware can be uploaded to the site, and a report is generated that includes extensive information on:
General information such as the MD5 hash and file size

- Load time DLLs
- Run time DLLs
- Packer signature
- Virus signature
- Registry activities
- File activities
- Process activities

This information provides a high level over view of the actions malware can conduct on a system and assists in determination of any possible threats. A post by Xc (2007) to a forum, reported that all of the analyzed files on Anubis were being executed from the directory `C:\InsideTM`. This makes it easy for the malware to check if it is being run from this directory. An *Anubis* detection routine was written by OG (2007).

Sandboxie (Sandboxie, 2008) is an application where suspicious programs can be run in an environment that uses a transient storage area, known as a sand box, so that data is not written to the hard drive. This allows the analyst to observe what an unknown program is going to do. However, *Sandboxie* can be defeated by "a DLL (`SbieDLL.dll`) being injected into the process run under *SandBoxie*" (Thrasher, 2007). Anti sandbox code was written by OG (2007).

Norman Sandbox (Norman, 2008) also provides an online service to analyze malware, but this also can be detected. Krack (2006) notes that the presence of the sandbox can be detected by "reading it's memory, and comparing it to that of a standard computer". Then, upon detection of the sandbox, the malware can halt its execution, resulting in nothing being logged and detected.

A sample program was written by Stargazer (2006) that can detect the *Norman Sandbox*. This is a problem because analysts can submit suspicious files to online analysis engines such as *Norman Sandbox*. If the suspicious file detects that it is running on such an engine, it can alter its behaviour so that it appears to be benign and the report generated from the online engine does not reflect its real potential. The analyst could then allow the suspicious file to run on real systems, unaware of its real, malicious purpose.

Analysts need to be aware of the limitations of their tools and the limitations of virtual environments, online analysis engines and sandboxes as outlined in the discussion above in this section. This, in general, highlights a weakness in dynamic analysis techniques where the analyst may not be aware that malware has detected the environment it is in, and is using deception to mask its true capability. In contrast to detailed static analysis of code, dynamic analysis is faster and much easier to perform, but is arguably, easier to deceive. The following section addresses the techniques malware can use to hinder static analysis techniques.

2.7. ANTI REVERSING TECHNIQUES

Eilam (2005, pp. 327-356) devotes a chapter in his book, on anti-reversing techniques. Eilam's discussion of techniques is ordered into the following headings, and discussed in the following sections:

- Eliminating symbolic information
- Code encryption
- Active anti-debugger techniques
- Confusing disassemblers
- Code obfuscation
- Control flow transformations
- Data flow transformations

2.7.1. Eliminating symbolic information

Release builds that use C or C++ typically remove all symbolic information, but byte code languages such as Java and C# contain information that is useful to the analyst. This is because byte code languages utilize names instead of addresses for cross referencing. These meaningful names can be replaced by meaningless strings by byte code obfuscators. DLL imports can also use ordinals instead of names (Eilam, 2005, pp. 328-330). Ordinals are simply numbers and may appear far less meaningful than a function that is appropriately named according to its purpose. This can make it harder for the analyst because a list of the names of function calls can make it easier to assess the overall functionality of the malware. This could include identifying calls to modify the registry, startup programs or communicate over the internet to other computers.

2.7.2. Code encryption

Eilam (2005, p. 330) explains that this technique is commonly used to prevent static analysis and is performed after the program is compiled. It contains a decryption section in the code and the program is decrypted at run time. This means that the analyst will most likely have to run the program to let it decrypt itself. This gives the malware control and the opportunity to use deception to hide its true intent from the analyst.

2.7.3. Active anti-debugger techniques

Eilam (2005, pp. 331-336, p.331-336) discusses a few active techniques that are better described in other papers (Ferrie, 2008; Falliere, 2007; Yason, 2007). However, one technique worth discussion is the use of code checksums. This technique calculates a checksum for particular functions and then checks at runtime if the function has been modified by code patching, or by the setting of software breakpoints. This helps the malware determine if it is being analyzed if the code has been patched or a software breakpoint set in the region of code of interest.

2.7.4. Confusing disassemblers

Two methods used by disassemblers are linear sweep and recursive traversal. Linear sweep is used by the disassemblers/debuggers *SoftIce* (Compuware, 2008) and *WinDbg* (Microsoft, 2008b), which conducts a disassembly in a sequential manner. Recursive traversal (used by *`OllyDbg`* and *`IDA Pro`*) follows the flow of each branch and is the more reliable technique and tolerant to anti-disassembly tricks. Linear sweeps can be easily confused with junk bytes, but the recursive sweep technique can also be fooled with opaque predicates (Eilam, 2005, pp. 336-344). Opaque predicates are simply code that appears to make a decision that could alter program flow, but in reality, only one branch of execution is possible to follow.

2.7.5. Code obfuscation

Eilam (2005, p. 344) says "code obfuscation involves transforming the code in such a way that makes it significantly less human-readable, while still retaining its functionality". Transformation characteristics include potency, which is the level of complexity added to the code and can be measured by complexity metrics including the depth of nesting in a particular sequence and the number of predicates the code contains. Another characteristic is that the transformation must be resilient. A highly resilient transformation is hard to undo. Deobfuscators can conduct data-flow analysis to reverse the transformation. There is also a cost characteristic of the obfuscation transformation in terms of increased size of the resultant code and slower execution time (Eilam, 2005, pp. 344-345, p.344-345).

2.7.6. Control flow transformations

Control flow transformations are another way of reducing human readability of code by altering the order and flow of a program (Eilam, 2005, p. 346). Control flow transformations are categorized as computation transformations, aggregation transformations and ordering transformations (Collberg, Thomborson, & Low, 1998)

2.7.7. Data transformations

Eilam (2005, pp. 355-356) explains that data transformations obfuscate the data of a program rather than the structure of the code by encoding some, or all, of a program's variables and/or by restructuring the arrays of the program.

2.8. ANTI UNPACKING

The 2nd International Caro Workshop was held in the Netherlands in May 2008 that focused on the problems and technical aspects of packers, decryptors and obfuscators as the major theme of the conference. Ferrie, (2008) a Senior AV Researcher at Microsoft, presented a paper at the conference that (at the time of writing this thesis) extensively lists what he refers to as the most common anti-unpacking tricks, together with some countermeasures. Ferrie's taxonomy for these techniques is as follows:

- Anti unpacking by anti dumping
- Anti unpacking by anti debugging
- Anti unpacking by anti emulating
- Anti unpacking by anti intercepting
- Miscellaneous

The techniques Ferrie discusses in his paper are summarized in the following sections under the same headings as the taxonomy listed above. It should be noted that these techniques need not only be used during the unpacking process. They can be used within the body of the malware itself.

2.8.1. Anti Unpacking by Anti Dumping

Packed malware can be run until the OEP is reached, which generally means that the original code is now unpacked in memory. The analyst can then dump the code from memory and then analyze it. These tricks are used to prevent an accurate facsimile of the code being dumped (Ferrie, 2008, p. 1).

2.8.1.1. Size of Image

The `SizeOfImage` value in the Process Environment Block (PEB) can be changed so that process access is impeded, as well as stopping a debugger from attaching to the process. Ferrie (2008, p. 1) says that it breaks popular dumping tools such as *`LordPE`* (yoda, 2005a) in default mode, and continues by saying that this technique can be defeated by ignoring the `SizeOfImage` value in the PEB and call the `VirtualQuery()` function instead. This returns the number of sequential pages whose attributes are the same, and these pages can be enumerated. The first page begins with the

`ImageBase` page and sequential pages should return the `MEM_IMAGE` type. A page that did not come from the file is indicated by a page that is not of the `MEM_IMAGE` type.

2.8.1.2. Erasing the Header

Ferrie (2008, p. 1) reports that some dumpers such as *ProcDump* (G-RoM, Lorian, & Stone, 1999) rely on the section table in the PE header, and that altering or erasing the table can defeat such dumping tools. Ferrie (2008, p. 2) advises using the `VirtualQuery()` function to recover the image size and to determine the permissions of the pages, but that it is not possible to recover the section table once it has been erased.

2.8.1.3. Nanomites

As a more advanced form of anti-dumping, this technique replaces branch instructions with software breakpoints (`INT 3`), called nanomites. This technique was introduced by the packing tool *`Armadillo`* (Silicon Realms, 2008), now mostly known as *`SoftwarePassport`*. Tables in the unpacking code record details of the nanomite. Ferrie (2008, p. 2) relates that a process that is protected by nanomites uses self-debugging. This technique uses a copy of the process as a debugger which can then intercept the exceptions generated by the debuggee when the nanomite is reached. When this occurs and if the exception address is in an address table, the type information is retrieved from a type table. The branch is taken if the type matches the CPU flags and the destination address is retrieved from a destination table. Execution resumes from that address. If a match is not made, a size table is used to retrieve the size of the branch so that the instruction can be skipped.

2.8.1.4. Stolen Bytes

`ASProtect` (ASPack Software, 2008) introduced this technique. These are instructions taken from the original program and relocated into dynamically allocated memory. The original programs instructions are replaced with junk code except for a jump to the start of the relocated code (Ferrie, 2008, p. 2).

2.8.1.5. Guard Pages

The purpose of Guard Pages is to act as an alarm if they are accessed, by raising an `EXCEPTION_GUARD_PAGE` (*0x80000001*) exception. Then the exception can be intercepted and then checked to see if the page is within a particular range such as the process image space. Ferrie (2008, p. 2) reports that the packing tool called *Shrinker* (Blinkinc, 2003) uses this technique to perform on-demand compression. It uses this technique to reduce the committed memory requirements because pages that are not required do not need to be loaded into physical memory. It does this by hooking the `ntdll KiUserExceptionDispatcher()` function and looking for the `EXCEPTION_GUARD_PAGE` exception.

Armadillo uses a variation of this technique to perform on-demand decryption but requires the use of self-debugging. It loads the entire program into memory at once, in contrast to the way *Shrinker* loads pages only as required. The debugger intercepts the exceptions raised by the debuggee and if the exception is within the process image space, the individual page that is being accessed is decrypted and execution resumes. Ferrie (2008, p. 3) suggests a way of mitigating *Armadillo's* implementation by touching all the pages in the image which should make *Armadillo* decrypt all pages which can then be dumped from memory.

2.8.1.6. Imports

Because the list of imported functions of a binary give a good idea of the overall functionality of a program, most packers alter the Import Table after the imports have been resolved by erasing it and replacing it with a different access mechanism. This could be a private buffer that holds real function addresses that is not dumped by default (Ferrie, 2008, p. 3).

2.8.1.7. Virtual Machines

The executable code is never visible if a virtual machine is used to unpack the code. This technique is used by packers such as *themida* (Oreans Technologies, 2008), *neoGuard* (Seculab, 2008) and *VMProtect* (VMProtect, 2008). Seculab's Russian web page extols the virtues of *neoGuard* to include a very high level of protection against disassembling and debugging and that a custom disassembler and compiler would have to be written by the analyst to analyse the code that has been protected using *neoGuard* (Seculab, 2008).

A simple technique to analyse packed malware is to let it unpack itself into memory, halt execution and then dump the code from memory and analyse it. Rolles (2007) reports that new protectors are applying transformations to the original code so that dumping and analyzing code is much more difficult. Rolles says this is done by “converting portions of the code into proprietary byte-code formats which are executed by an embedded interpreter (so-called virtualization, virtual machines) and copying portions of the code elsewhere in the process' address space (so-called stolen bytes, stolen functions)”. This means that packers that use virtual machines run their unpacking routines from within a VM. The advantage to malware authors is that it negates the usefulness of existing, static analysis tools. Static analysis is broken because each different VM has a different instruction encoding format (and this can be polymorphic). Patching the VM program requires a familiarity with the instruction set that must be gained through analysis of the VM parser (Rolles, 2007).

2.8.1.8. Anti Unpacking by Anti Debugging

These techniques focus on preventing or hindering analysis when the malware is being run inside a debugger, or if a debugger tries to attach to a running process.

2.8.1.9. NtGlobalFlag

The `NtGlobalFlag` is a field in the PEB at offset *0x68* that is zero by default, but has a value stored in it when the process is running in a debugger. The value is comprised of a set of flags as follows:

```
FLG_HEAP_ENABLE_TAIL_CHECK(0x10)
FLG_HEAP_ENABLE_FREE_CHECK(0x10)
FLG_HEAP_VALIDATE_PARAMETERS(0x40)
```

Ferrie (2008, p. 3) emphasizes that other flags can be set in this value and it is a mistake to simply compare the value of this field with *0x70* to check for the presence of a debugger. Although these three flags are usually set for a debugger, they are not set for a debugger that attaches to a running process. Ferrie also points out three more exceptions. Additional flags can be set for all processes with the value of `GlobalFlag` by the registry key:

```
HKLM\System\CurrentControlSet\Control\SessionManager
```

The next exception is that all flags can be controlled on a per-process basis by the value of `GlobalFlag` by the registry key:

```
HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Image File
Execution Options\<filename>
```

Where `<filename>` is replaced by the name of the file being executed.

The third exception is all of the flags can be controlled by the Load Configuration Structure on a per-process basis and was introduced to support Safe Exception Handling in Windows XP. It also contains two fields called `GlobalFlagsClear` and `GlobalFlagsSet` and can be used to set or clear any flags in the `NtGlobalFlag` field in the PEB.

2.8.1.10. Heap Flags

The default heap of the process can give away the presence of a debugger. The pointer to the base of the heap can be determined by using the `kernel32 DLL GetProcessHeap()` function, or alternatively by directly accessing the PEB. The handle to the process heap is at offset *`0x18`* in the PEB from which there are two fields of interest, Flags at offset *`0x0c`* which shows the settings for the current heap block and `ForceFlags` at offset *`0x10c`* which shows the settings for how the heap will be manipulated. Ferrie (2008) says the presence of a debugger could be indicated by these values set in the Flag field as shown in Figure 2-9.

```
HEAP_GROWABLE(0x02)
HEAP_TAIL_CHECKING_ENABLED(0x20)
HEAP_FREE_CHECKING_ENABLED(0x40)
HEAP_SKIP_VALIDATION_CHECKS(0x10000000)
HEAP_VALIDATE_PARAMETERS_ENABLED(0x40000000)
```

Figure 2-9 Heap Flags that can be read and used to detect the presence of a debugger.

Ferrie (2008, p. 4) says that the presence of a debugger could be indicated by the setting of these flags in the `ForceFlags` field.

```
HEAP_TAIL_CHECKING_ENABLED(0x20)
HEAP_FREE_CHECKING_ENABLED(0x40)
HEAP_VALIDATE_PARAMETERS_ENABLED(0x40000000)
```

Figure 2-10 Force Flag fields that can be read and used to detect the presence of a debugger.

2.8.1.11. *The Heap*

Ferrie (2008, p. 5) reports that some artifacts can still be detected after the heap flags have been cleared, and that packers such as *Themida® (Oreans Technologies, 2008)* look for these. The following flag can cause the sequence *0xABABABAB* to appear twice at the end of the allocated block.

    HEAP_TAIL_CHECKING_ENABLED

Whilst the flag `HEAP_FREE_CHECKING_ENABLED` can cause the whole, or part sequence of *0xFEEEFEEE* to appear if bytes are required to fill the slack space between blocks.

2.8.1.12. *Special API's*

Various API's can be used to detect the presence of a debugger. These are presented in the following subsections.

2.8.1.12.1. *IsDebugger Present*

A call to the `kernel32` DLL `IsDebuggerPresent()` function returns TRUE if a debugger is found. Because it simply returns the value of the

BeingDebugged field of the PEB, the kernel32 call can be bypassed by directly looking at the PEB. This method can be defeated by setting the flag to FALSE (Yason, 2007).

2.8.1.12.2. Check Remote Debugger Present

This call has two parameters, a process handle, and a pointer to a BOOLEAN variable that will be set to TRUE if it is found that a debugger is attached to the process (Yason, 2007). The signature of this call is as follows:

```
BOOL CheckRemoteDebuggerPresent (
     HANDLE     hProcess,
     PBOOL      pbDebuggerPresent
)
```

Figure 2-11 Signature of the CheckRemoteDebuggerPresent function that can be used to detect the presence of a debugger.

2.8.1.12.3. NtQueryInformationProcess

The call chain for CheckRemoteDebuggerPresent is via ntdll NtQueryInformationProcess() which queries the DebugPort field of the EPROCESS kernel structure (Yason, 2007).

2.8.1.12.4. Debug Objects

Ferrie (2008, p. 6) explains that Windows XP introduced the idea of a “debug object” that is created when a debugging session commences. A handle is associated with this object and the ProcessDebugObjectHandle class can be used to query the value of the handle.

2.8.1.12.5. NtQuery Object

The number of debug objects can be obtained by using ntdll NtQueryObject() function call. This call returns a structure called OBJECT_ALL_INFORMATION which contains a field called NumberOfObjectsTypes which is a count of total object types. A mitigation strategy is to set a breakpoint when NtQueryObject returns and then patch the NumberOfObjectsTypes field to 0 (Ferrie, 2008, p. 7).

2.8.1.12.6. *Thread Hiding*

The `SetInformationThread()` call can be used to hide a thread using an information class called `HideThreadFromDebugger`. The thread will continue to run when the function is called, but a debugger will no longer receive any events related to that thread (Ferrie, 2008, p. 7).

2.8.1.12.7. *Open Process*

When a process is loaded into a debugger, the `SePrivilege` privilege in the access token is enabled. It is not enabled when not loaded into a debugger. "Some packers indirectly use `SeDebugPrivilege` to identify if the process is being debugged by attempting to open the `CSRSS.EXE` process" (Yason, 2007, p. 9). `CSRSS.EXE` (Client Server Runtime Server Subsystem) manages most of the graphical commands of Windows. The idea behind this is that the security descriptor of the `CSRSS.EXE` process only allows `SYSTEM` to access this process. A process that has the `SeDebugPrivilege` can access any process regardless of the security descriptor. Yason (2007, p. 10) says that this privilege is only granted to members of the Administrators group by default.

Packers may try to obtain the `PID` of `CSRSS.EXE` via process enumeration. A possible solution to this technique is to set a breakpoint where `ntdll NtOpenProcess()` returns and to set the value of EAX to *`0xC0000022`* (`STATUS_ACCESS_DENIED`) when the breakpoint is reached if the `PID` that is passed is that of `CSRSS.EXE` (Yason, 2007, p. 10)

2.8.1.12.8. *Close Handle*

The presence of a debugger can be detected by making use of the `ZwClose` system call. `CloseHandle` indirectly makes use of this call. Calling `ZwClose` with an invalid handle will generate a `STATUS_INVALID_HANDLE` exception. Falliere (2007, p. 6) says that "the only proper way to bypass the `CloseHandle` anti-debug is to either modify the `syscall` data from ring 0, before it is called, or set up a kernel hook."

2.8.1.12.9. *Output Debug String*

Falliere (2007, p. 7) says that if OutputDebugStringA is called with a valid ASCII string under the control of a debugger, the return value will the address of the string passed as a parameter. When not run in a debugger, the return value should be 1.

2.8.1.12.10. Read File

By reading file content into the code stream, the kernel32 ReadFile() function can be used as technique for self modification. It can also be used to remove the software breakpoints that a debugger may have placed. This technique can be defeated by using hardware breakpoints instead of software breakpoints (Ferrie, 2008, pp. 8-9).

2.8.1.12.11. Write Process Memory

The WriteProcessMemory() function of the kernel32 DLL can be used in a similar way to the ReadFile() function but requires that the data that is to be written is already in process memory space. Ferrie (2008, p. 9) says that the use of this technique can be defeated using hardware breakpoints.

2.8.1.12.12. Unhandled Exception Filter

Windows has a chained Structured Exception Handler (SEH) mechanism to pass exceptions to handlers instead of crashing the program if possible. Malware can take advantage of SEH to gain control of the malware to detect it is being debugged. The malware throws an exception deliberately, and if its own SEH does not handle the exception, it can deduce that it is being debugged (Yason, 2007, p. 25).

2.8.1.12.13. Block Input

Packers can use the user32 DLL BlockInput() function to prevent the analyst from using input devices such as the keyboard and mouse whilst the unpacking routine is being executed, and makes the system appear unresponsive during this time (Yason, 2007, p. 23).

2.8.1.12.14. Suspend Thread

User mode debuggers can be disabled by the use of the kernel32 SuspendThread() function. Ferrie (2008, p. 9) reports that *Yoda's*

Protector (yoda, 2005b) uses this technique which enumerates the process and then suspends the main thread of the parent process if it does not match *Explorer.exe* (Microsoft, 2008a) which is the parent process.

2.8.1.12.15. Guard Pages

This technique registers an exception handler, a page is dynamically allocated to it that is executable and writeable and the opcode RET is written to it. The page protection is changed to PAGE_GUARD and then an attempt is made to execute the RET instruction which will result in an EXCEPTION_GUARD_PAGE exception being raised. A debugger may intercept the exception and hence give away its presence. *PC Guard* (Sofpro, 2008) uses this technique (Ferrie, 2008, p. 10).

2.8.1.12.16. Alternative Desktop

An alternative desktop can be hidden by a technique described by Ferrie (2008, p. 10) and is used by the protector with its own VM, *HyperUnpackMe2* (Anonymous, n.d.-g).

2.8.2. Hardware Tricks

Various hardware related tricks can be utilized to determine if the process is being debugged. These techniques are presented in the following subsections.

2.8.2.1. Prefetch Queue

Prior to the Pentium and later CPU's, a variety of tricks were possible by exploiting some ways the prefetch queue for the CPU was mishandled by allowing the overwriting of the next instruction to execute after an exception occurred. Ferrie (2008, p. 10) says that the REP MOVS and REP STOS instructions can still be used to exploit this mishandling. These two instructions are cached by the CPU and will execute them even if the same instructions in memory have been overwritten.

2.8.2.2. Hardware Breakpoints

There are 8 debug registers (DR0 – DR7) that are used to set hardware breakpoints. Malware can detect that it is being debugged by setting them to particular values and checking them later, or by simply resetting them. Ferrie (2008, p. 11) says that the packer called *Telock* (TGM, 2004) employs this technique to detect the use of a debugger as does *ASProtect*. Debug registers cannot be set directly in user mode, but other ways Falliere (2007, p. 11) lists include:

- Throwing an exception and then modifying the thread context because it contains the contents of the CPU registers, and then resuming normal execution with the modified context.
- Using the NtGetContextThread and NtSetContextThread system calls through the kernel32 DLL functions GetThreadContext and SetThreadContext.

2.8.2.3. Instruction Counting

This technique registers an exception handler and then sets some hardware breakpoints. When the addresses of the breakpoints are hit, an EXCEPTION_SINGLE_STEP exception is raised and passed to the exception handler which is then able to adjust the instruction pointer to point to a new instruction from which execution can resume. The kernel32 GetThreadContext() function can be used to access the context structure of the thread. Some debuggers do not correctly handle hardware breakpoints not set by the debugger itself and this may result to instructions not being counted properly (Ferrie, 2008, p. 11).

2.8.2.4. Execution Timing

Packers and debug detection routines take advantage of the fact that code running in a debugger is going to take longer to execute than when not running in a debugger. The routines measure the time elapsed and compares the time differential with a normal run time value. If it took longer to run than expected, then it is probably running in a debugger. The RDTSC (Read Time Stamp Counter) instruction can be used before and after a routine to determine how much time elapsed.

The kernel32 DLL has a function called GetTickCount that returns with the number of milliseconds elapsed since the system was started. A

SharedUserData data structure is always located at address *0x7FFE0000* and contains the fields TickCountLow and TickCountMultiplier.

A simple solution would be to identify where the timing checks are being performed in the code, and then set a breakpoint before the first time delta measurement and then perform a run instead of a step until the breakpoint is hit (Yason, 2007, p. 8). Alternatively the return result from a call to GetTickCount and modify the return value. Yason says that *Olly Advanced* (MaRKuS, 2006) installs a kernel mode driver that performs the following:
Sets the Time Stamp Disable bit (TSD) in the CR4 control register which will trigger a General Protection (GP) exception if the RDTSC instruction is executed in a privilege level other than 0.

The Interrupt Descriptor Table (IDT) is setup so that the GP exception is hooked and the execution of the RDTSC is filtered.

2.8.2.5. EIP via Exceptions

Ferrie (2008, p. 12) says that it is a very common trick of unpackers such as *PECompact* (Bitsum Technologies, 2008) to use exceptions to alter the EIP and also to gain a measure of obfuscation if the trigger of the exception is not obvious.

2.8.3. Process Tricks

A number of process related techniques are available to determine if the process is being debugged and to hinder the analysis process. These techniques are discussed in the following subsections.

2.8.3.1. Header Entry Point

Since the PE header is read only by default, some unpackers, including *MEW* (Northfox, 2004), set the entry point of the program in it. This effectively blocks the debugger from setting a break point at the entry point, unless the kernel32 VIrtualProtectEx() function is called first (Ferrie, 2008, p. 13).

2.8.3.2. *Parent Process*

A process often has explorer.exe as its parent process, and a parent other than explorer.exe may have been spawned by a debugger. Yason (2007, p. 10) says that this can be determined by the following process.

1. Get the current process PID via the `TEB` (`TEB.ClientID`) or by calling `GetCurrentProcessId()`.

2. Use `Process32First/Next()` and get `explorer.exe's` PID from `PROCESSENTRY32.szExeFile` and get the PID of the parent process of the current process from `PROCESSENTRY32.th32ParentProcessID`.

3. The target may be being debugged if the PID of the parent process is not the same as the PID of explorer.exe.

A false positive may result if the process was launched using the command prompt or if the default shell is different. Yason says that this can be mitigated by setting `Process32Next()` to always fail when using *Olly Advanced*. Ferrie (2008, p. 13) reports that *Yoda's Protector* is among the packers that use this technique.

2.8.3.3. *Self Execution*

A process can escape the control of a debugger by executing a copy of itself by utilizing a `mutex`. The initiating process creates the `mutex` and then executes a copy of the process which will not be debugged, even if the first process was being debugged and will know that it is a copy since the `mutex` will be found to already exist (Ferrie, 2008, p. 15).

2.8.3.4. *Process Name*

Some packers look for process names that match the names of debugging or malware analysis tools using the `kernel32` `CreateToolhelp32Snapshot()` function (Ferrie, 2008, p. 16).

2.8.3.5. *Threads*

Some packers such as *PE-Crypt32* (random, killa, & acpizer, 1999) use threads to check for the presence of a debugger, or to check the integrity of the main code (Ferrie, 2008, p. 16).

2.8.3.6. *Self Debugging*

Ferrie (2008, pp. 16-17) says that this technique used by *`Armadillo`* and other packers, runs a copy of a process and attaches to the copy as a debugger. This makes the process un-debuggable because only one debugger can attach to a process at any one point in time. This technique can be defeated by using kernel mode code to zero the `EPROCESS->DebugPort` field to allow another debugger to attach to the process. A DLL can also be injected into the process space by using the `kernel32 OpenProcess()` function. On Windows XP and later, the `kernel32 DebugActiveProcessStop()` function can be utilized to detach the debugger.

2.8.3.7. *Disassembly*

Breakpoints set within the first few instructions of an API can be bypassed if the packer uses API interception and copies the first few instructions of the function into a private buffer, and executes the instructions from there. The packer places a jump at the end of the last copied instruction so that execution of the original code resumes just after the point the last copied instruction was made. This also gives the packer the opportunity to search for breakpoints that have been set in the code which is an indication that the program is being debugged (Ferrie, 2008, p. 17).

2.8.3.8. *TLS Callback*

This technique is used to change the original entry point of a program to a different entry point so that an initial check can made to see if a debugger or other analysis tools are being run. It changes the PE loader so that the entry point of the program is referenced in Thread Local Storage (TLS), which is the 10th directory entry in the optional PE header (Falliere, 2007). TLS callbacks can be identified by examining the Data Directory of the PE header using a tool such as *pedump* (Pietrek, n.d.) because it will show if a TLS directory is in the executable (Yason, 2007, p. 28).

2.8.3.9. *Device Names*

Packers can use a device driver technique to detect debuggers such as *`OllyDbg`* and *`IDA Pro`* as well as monitors running at the system level such as the SysInternals tools *`Regmon`* and *`Filemon`*. This technique uses `kernel32 CreateFile` against well known names. Yason (2007, p. 13) says that some versions of *`SoftICE`* append numbers to the device name which will cause this check to fail. However, a brute force approach can be used to find the name by appending numbers to the search routine in a loop.

Ferrie (2008) provides examples of some device names used by popular analysis tools that are reproduced in Figure 2-12.

```
SoftIce
     \\.\SICE
     \\.\SIWVID
     \\.\NTICE
Regmon
     \\.\REGVXG
     \\.\REGSYS

FileMon
     \\.\FILEVXG
     \\.\FILEM
```

Figure 2-12 Device names used by popular debugging tools that can be used by malware to detect their presence.

2.8.3.10. SoftIce Specific

`SoftIce` was a popular ring 3 and ring 0 debugger for the Windows platform.

2.8.3.10.1. Driver Information

`SoftIce` device drivers can be enumerated using the `ntdll NtQuerySystemInformation()` function. The version information of each file can then be determined using the `VerQueryValue()` function as well as strings that can be matched including *`SoftIce`* (Ferrie, 2008, p. 18).

2.8.3.10.2. Interrupt 1

Ferrie (2008, p. 18) explains that the `int 1` instruction cannot be set from ring 3 and will raise an `EXCEPTION_ACCESS_VIOLATION` exception (General Protection Fault) if this interrupt is called directly. However, *`SoftIce`* hooks this interrupt and adjusts the Descriptor Privilege Level (DPL) to 3 from its normal DPL of 0 to enable *`SoftIce`* to single step user mode code. When the `int 1` occurs, *`SoftIce`* does not check the cause was a software interrupt or the trap flag and it always calls the handler for `interrupt 1` and an EXCEPTION_SINGLE_STEP exception is raised when an EXCEPTION_ACCESS_VIOLATION exception should have been raised resulting in the detection of the presence of *SoftIce*.

2.8.3.11. *OllyDbg Specific*

OllyDbg is a very popular ring 3 debugger. The techniques in the following subsections examine ways of detecting its presence.

2.8.3.11.1. *Malformed Files*

Ferrie (2008, p. 19) says that *OllyDbg* “will refuse to open a file whose data directories do not end exactly at the end of the Optional Header”. *OllyDbg* tries to allocate the amount of memory that the Export Directory Size, Base Relocation Directory Size, Export Address Table Entries and PE->SizeOfCode fields say, no matter how large the values are which can cause the system swap file to grow so large that it affects the performance of the system.

2.8.3.11.2. *Initial ESI Value*

Some packers try to detect *OllyDbg* by examining the initial value of the ESI register. Ferrie (2008, p. 19) reports that this value is *0xFFFFFFFF* on Windows XP, but 0 in Windows 2000, and is just a coincidence.

2.8.3.11.3. *Output Debug String*

Falliere (2007, p. 7) reports if OutputDebugStringA is called with a valid ASCII string under the control of a debugger, the return value will the address of the string passed as a parameter. When not run in a debugger, the return value should be 1. Yason (2007, p. 26) says that this technique is specific to *OllyDbg* because it is vulnerable to a format string bug.

2.8.3.11.4. *Find Window*

The user32 function FindWindow() and FindWindowEx() can be used to find out if known applications are being run including *OllyDbg* (Yason, 2007, p. 22).

2.8.3.11.5. *Guard Pages*

An attempt to execute instructions in a guarded page should result in an exception, but *OllyDbg* executes them (Ferrie, 2008, p. 19).

2.8.3.12. *Hide Debugger Specific*

OllyDbg has an enormous variety of plugins including ones to counter detection techniques. One of these is *HideDebugger* (Shub-Nigurrath, 2006) which hooked the kernel32 OpenProcess() function by setting a far jump to a new handler. The detection of this jump provides a good indication of the presence of the plugin (Ferrie, 2008, p. 19).

2.8.3.13. *Immunity Debugger Specific*

Ferrie (2008, p. 20) points out that the *Immunity Debugger* (Immunity, 2008) is a customization of *OllyDbg* with a *Python* command-line interface and is vulnerable to all the same detection and vulnerabilities as *OllyDbg*.

2.8.3.14. *WinDbg Specific*

WinDbg is a Microsoft distributed, ring 3 and ring 0 debugger. The following techniques are *WinDbg* specific.

2.8.3.14.1. *Find Window*

Ferrie (2008, p. 20) says that the user32 FindWindow() function can be used to detect *WinDbg* by using the class name WinDbgFrameClass.

2.8.3.15. *Miscellaneous Tools*

The following sub section discusses various miscellaneous tools.

2.8.3.15.1. *Find Window*

Less common tools that malware searches for includes the window name string of Import REConstructor v1.6 FINAL © 2001-2003 MackT/uCF or class name of TESTDBG, kk1, Eew57 or Shadow (Ferrie, 2008, p. 20).

2.8.4. *Anti Unpacking by Anti Emulating*

This section discusses some of the techniques used to detect emulators and virtual machines.

2.8.4.1. *Software Interrupts*

2.8.4.1.1. *Interrupt 3*

An emulator can be detected if it does not behave the same way as Windows. The EIP has already been advanced to the next instruction when an `EXCEPTION_BREAKPOINT` occurs and Windows tries to set the EIP back to where it should be, but Windows assumes that the exception is caused by the short form of `int 3 (CC)`. However, the EIP will point to the wrong place if the long form of `int 3 (CD 03)` caused the exception (Ferrie, 2008, p. 20).

2.8.4.2. *Time Locks*

Anti-emulation code can exploit the characteristic of emulators to limit the amount of time and/or the number of instructions that will be executed before exiting with no detection. The anti-emulation code can use a dummy loop to force the emulator to give up (Ferrie, 2008, p. 20).

2.8.4.3. *Invalid API Parameters*

For the purpose of simplicity, some emulators do not provide error checking for the return results of API calls. Some anti-emulator code can exploit this vulnerability to detect the presence of an emulator including that of the *`Tibs`* packer (Ferrie, 2008, p. 20). The *`Tibs`* (Anonymous, n.d.-m) packer is often used to pack the *`storm`* worm and has anti-emulation capability (Websense, 2008).

2.8.4.4. *Get Proc Address*

The address of a function exported by a DLL is obtained by using the `kernel32` function `GetProcAddress()`, however, not all functions are provided by the virtual environment such as the `kernel32` function `GetTapeParameters()`. Because some packers try to exploit this, some anti-malware emulators return a value for `GetProcAddress()` without due consideration to the parameters that were passed to it. The anti-emulator code can call a function with invalid parameters fully expecting not to receive a return value, and an emulator can be detected if a valid result is returned (Ferrie, 2008, p. 21).

2.8.4.5. Get Proc Address (Internal)

Ferrie (2008, p. 21) says that “some anti-malware emulators export special APIs, which can be used to communicate with the host environment”.

2.8.4.6. “Modern” CPU Instructions

Ferrie (2008, p. 21) advises that for the purposes of simplicity, some anti-malware emulators do not implement the entire CPU instruction set and leave out less common instructions such as `CMPXCH8B` and entire instruction classes such as Floating Point Unit (FPU), Multimedia Extensions (MMX) and Streaming Single Instruction Multiple Data Extensions (SSE). This can be used by the packer to detect the presence of the emulator, or the emulator may fail to determine what the malware is doing.

2.8.4.7. Undocumented Instructions

Am emulator may fail to detect the intention of the malware if a packer can use undocumented instructions that are not supported by the emulator (Ferrie, 2008, p. 22).

2.8.4.8. Selector Verification

Ferrie (2008, p. 22) says that packers such as *MSLRH* (Anonymous, n.d.-i) can use the `kernel32 GetVersion()` function to get the operating system version which can then be compared with the descriptor table layout. On a Windows NT-based system the CS selector should be *0x1B* for ring 3 code, whilst on Windows 9x-based platforms the CS selector can exceed *0xFF* (Ferrie, 2008, p. 22).

2.8.4.9. Memory Layout

Anti-malware emulators may not include the in-memory structures that a real system will have such as the RTL_USER_PROCESS_PARAMETERS which should appear at memory location *0x20000* (Ferrie, 2008, p. 22).

2.8.4.10. File Format Tricks

This section discusses a number of PE Header file format tricks used by malware that do not conform to the way the emulator expects to file to appear.

2.8.4.10.1. Non Aligned Size of Image

The `PE->SizeOfImage` field is stated in the file format documentation to be a multiple of the value in the `PE->SectionAlignment` field but is not a requirement and Windows can round up the value if required. Malware can take advantage of this to ensure that it will not run within a VM and hence hinder analysis (Ferrie, 2008, p. 22)

.

2.8.4.10.2. Overlapping Instructions

Structures in the PE Header file can be made to overlap such as the `MZ->lfanew` field so that the PE header appears inside the `MZ` header. The `PE->SizeOfOptionalHeader` field can be set so that it appears as if a section table is in the `DataDirectory` array. The Import Address Table and the Import Lookup Table virtual address values can "produce an import table which has fields inside the PE header" (Ferrie, 2008, p. 22).

2.8.4.10.3. Non Standard Number of RVA and Sizes

The location of the section table should be determined by using the `PE->SizeOfOptionalHeader` field. Ferrie (2008, p. 22) says a common mistake made by both *`SoftIce`* and *`OllyDbg`* is to "assume that the value in the `PE->NumberOfRvaAndSizes` field is set to the value that exactly fills the Optional Header, and that the section table follows immediately".

2.8.4.10.4. Non Aligned SizeOfRawData

By recognizing that Windows automatically rounds up the `SizeOfRawData` field in the section table, a section table can be created whose entry point appears in pure virtual memory but there will not have physical data to execute because of the rounding (Ferrie, 2008, p. 23).

2.8.4.10.5. Non Aligned PointerToRawData

A section can be created where the entry point appears to point to data anywhere other than what should be executed because the

PointerToRawData field in the section table is subject to rounding down by Windows (Ferrie, 2008, p. 23).

2.8.4.10.6. No Section Table

If the value of the PE->SectionAlignment field is reduced to less than 4kb, the PE header is marked as both executable and writeable and the contents of the section table become optional. This means the entire section table can be zeroed out. The file is then mapped as if it were only one section where the size is that of the value set in the PE->SizeOfImage field (Ferrie, 2008, p. 23).

2.8.5. Anti Unpacking by Anti Intercepting

2.8.5.1. Write->Exec

Some unpacking tools try to determine when the unpacker has completed the unpacking process and transferred control to the host. It can do this by intercepting the execution of newly written pages by first writing and then executing a dummy instruction. This can cause the interceptor to exit early (Ferrie, 2008, p. 23).

2.8.5.2. Write^Exec

Ferrie (2008, p. 23) says that some unpacking tools can change the page attributes of memory from writeable-executable to writeable or executable but not both.

2.8.6. Miscellaneous

2.8.6.1. Fake Signatures

Packers such as *RLPack Professional* (Reversing Labs, 2008) provide a false signature so that packer signature matching tools such as *PEiD* (Jibz, Qwerton, Snaker, & XineohP, 2006) incorrectly identify the packer (Ferrie, 2008, p. 24).

2.9. PROCESS INJECTION TECHNIQUES

Harbour (2007, p. 21) explains that process injection is used to inject code into another running process. The target process executes the malicious code. In so doing, it acts to conceal the source of the malicious behaviour. It can be used to bypass process specific security mechanisms and host base firewalls. The Windows Hooks mechanism can be used to achieve this by letting the process run specific code when a particular message is received. The Win32 API call `SetWindowsHookEx()` allows the target process to load a specified DLL into the memory space of the executable and select a function as a hook to handle a particular event. When the event is received, the target process executes the malicious code. An example provided in the paper by Harbour (2007, p. 25) is reproduced as follows in Figure 2-13

```
HANDLE hLib, hProc, hHook;
hLib = LoadLibrary(“evil.dll”);
hProc = GetProcAddress(hLib, “EvilFunction”);
hHook = SetWindowsHookEx(WH_CALLWNDPROC, hProc, hLib, 0);
```

Figure 2-13 Code snippet showing SetWindowsHook function to load a malicious DLL.

Another method is to use library injection. A new thread is created in the process which is used to load the malicious library. “When the library is loaded by the new thread, the `DllMain()` function is called, executing your malicious code in the target process” (Harbour, 2007, p. 29). An example provided by Harbour (2007, p. 30) is reproduced as follows in Figure 2-14.

```
char libPath[] = “evil.dll”;
char * remoteLib;
HMODULE hKern32 = GetModuleHandle(“Kernel32”);
void *loadLib = GetProcAddress(hKern32, “LoadLibraryA”);
remoteLib = VirtualAllocEx(hProc, NULL, sizeof (liPath),
     MEM_COMMIT, PAGE_READWRITE);
CreateRemoteThread(hProc, NULL, 0, loadLib, remoteLib, 0,
     NULL));
```

Figure 2-14 Code snippet showing library injection to load a malicious DLL.

Yet another method pointed out by Harbour (2007) is to use Direct Injection. This is where the memory space of the process is populated with the malicious code, which could be a function or an entire DLL, which he says is much harder to do. API's required include `VirtualAllocEx()`, `WriteProcessMemory()` and `CreateRemoteThread()` which is used to create a new thread in the process.

2.10. CODE EXECUTION FROM MEMORY

If the code is executed directly from memory and never resides on the hard drive, it may not be detected during a forensic acquisition. The "memory buffer to be executed will most likely be populated directly by a network transfer" (Harbour, 2007, p. 35). Source code contained in a variable can be executed by something similar to `exec()` or `eval()`.

Harbour (2007, p. 42) discusses a technique known as the Nebbett Shuttle to launch Win32 executables from a memory buffer and provides an example that is reproduced of what an implementation could look like. Essentially the technique launches a process in a suspended state and then overwrites the allocated memory space with a new executable.

```
CreateProcess(..., “cmd”, ..., CREATE_SUSPEND, ...);
ZwUnmapViewOfSection(...);
VirtualAllocEx(..., ImageBase, SizeOfImage, ...)
WriteProcessMemory(..., headers, ...);
for (i=0; i< NumberOfSections; i++) {
     WriteProcessMemory(..., section, ...);
}
Resumethread();
```

Figure 2-15 Code snippet using Nebbet shuttle to launch Win32 executable code.

A specified process `cmd` is loaded into memory, but is suspended at the entry point. All memory that is allocated to the process is released. Area is allocated to put the new executable image in the memory space of the original process. The PE headers are written to the start of the memory region. Each section of the new executable is written to its new virtual address. The new, malicious process is still named as `cmd` in the task list, and since the process inherits privileges from the original code, if the original code was allowed to communicate through a host based firewall, the replacement code will be allowed to as well.

2.11. CHECKSUM CHECKS

Malware can use checksums to try to determine if the code has been changed. This could have been done by the malware analyst to change the flow of the malware, or to have patched out anti forensic implementations in the code (W. Yan, Zhang, & Ansari, 2008).

2.12. PROCESS CAMOUFLAGE

“A cleverly named process is often enough to fly beneath the radar and avoid immediate detection” (Harbour, 2007, p. 32). There can often be several copies of `svchost.exe` and `spoolsv.exe` running in memory, and additional processes with the same name may go unnoticed. Other name variations could include `svcshost.exe`, `spoolsvc.exe`, `spoolsvr.exe`, `scardsv.exe` and `lsasss.exe`.

2.13. STRUCTURED EXCEPTION HANDLING

Structured Exception Handlers (SEH) can be used to detect the presence of a debugger. All Win32 applications have an Operating System (OS) supplied SEH, and the exception handling mechanism is thread based. The exception handling mechanism in Linux is process based, and the exception handler is set up with a `signal()` system call. The global handler in `ntdll.dll` catches the exception and determines where control is given to. The SEH is a function pointer, and it is possible to overwrite the pointer to a SEH chain (exception-handler list), where if one handler chooses not to handle the exception, then the next handler can do it. The final handler is a default handler for the process which must handle it (Koziol et al., 2004, p. 116).

The exception handler list is stored in the Thread Information Block (TIB) data structure, which can be found at `FS:[0]`. A single process can have multiple threads, and each thread has a TIB, but all threads see the same memory, and all share the same address space (Eilam, 2005, p. 106).

Packers such as *AsProtect* use this mechanism to gain control, and to see if it is running inside a debugger. *AsProtect* creates multiple exceptions and a trick to unpacking *AsProtect* is to count the number of exceptions. *OllyDbg* can be configured to either pass exceptions to the process to handle, or to handle within the debugger. If the debugger is set to handle the exceptions, it will give the user the choice to handle the exception, or to pass back to the process. Using this iterative process, the number of exceptions can be counted until the process freely runs. This gives the analyst the opportunity to break on the OEP. If the count of exceptions is n, then the next time it is run, only pass n-1 exceptions to the process. At this point, the memory map can be viewed, and the code section can be seen where the OEP is in. A break point can be set on the code section (set memory break point on access). Then, when the jump to the OEP is conducted, the breakpoint on the entire section will be triggered on the OEP and the process can dumped (Anthracene, 2006).

2.14. IMPORT ADDRESS TABLE

Much of the functionality in a program is derived from calls to functions arranged in libraries called Dynamic Link Libraries (DLL), and the information necessary to call DLL functions is stored in the Import Address Table (IAT) of a binary. Programs typically use the DLL's supplied by Microsoft to interact with the OS to perform common tasks. Because these tasks are so common, multiple programs can share the same DLL's that are loaded, and reduce unnecessary duplication. The PE header of a program is read when it is loaded by the dynamic linker, and the addresses of the DLL functions (function pointers) the program requires are filled in, in the IAT (Eilam, 2005, p. 487).

Typically however, the Import Address Table (IAT) will be obfuscated by the packer or protector. Craig (2006) explains that at compile time, the IAT contains `NULL` memory pointers for each function, but when the executable is loaded at run time, Windows overwrites the `NULL`s with the correct memory location for each function. This is because the address of the DLL in memory will be different on any particular machine.

"The IAT is resolved with a `LoadLibrary` loop, just before a jump to the original entry point" (Falliere, 2006, p. 1). The import name table is typically messed up, but can be rebuilt using tools such as *ImpRec* (MackT, 2008). Most packers used by malware do their best to mess up the IAT so that the analyst cannot easily determine the DLL functions called. Typically, only three DLL functions will be visible for programs that have been packed at load time. The malware packer may have generally messed up the IAT by encrypting it, altered its size, or mangled it some other way. It is important to understand how the IAT should look, because the analyst may have to repair it.

2.15. ROOTKITS

Windows uses four privilege levels, known as rings, to determine the access level for access control. Access control determines how hardware can be accessed, what instructions a process may use, what files may be modified and which areas of memory can be accessed or changed. Ring 0 is the most privileged level and Ring 3 has the least amount of privilege. Most applications users run, are run in Ring 3 and these applications cannot access hardware directly and have limited access to memory. Ring 3 is often referred to as "user land". Ring 0 applications run with full system privileges and can perform IO and memory management, run device drivers, execute privileged instructions, access all memory space, access all hardware and access all components of the kernel. This is often referred to as "kernel land".

A special mechanism exists so that a user land program can access kernel land in a controlled fashion so that device drivers (`*.sys` file) can be installed. Root kits exploit this mechanism so that they can install their own device driver into kernel land, giving their program full privileges at Ring 0 and hence control the environment in which other software runs. In this way, it can avoid detection (Hoglund & Butler, 2005).

2.15.1. System Service Dispatch Table

System calls are used by user land programs to initiate a function in kernel land which works by interrupting the execution of the user land program and transfers control of execution to the kernel which is then responsible for processed the request. System calls are identified by a system call number, which is placed into the EAX register and are processed by a kernel routine called KiSystemService. After processing the request, the user land program resumes execution. KiSystemService looks up the system call number that is in EAX in a table known as the System Service Dispatch Table (SSDT). The SSDT contains the memory addresses of all of the system calls and is an ideal target for malicious code to get control of to control the execution of the kernel by rerouting calls to legitimate functions to functions the rootkit wants to call instead. This technique is referred to as Hooking and is used by legitimate software as well.

2.15.2. IAT Hooking

The Import Address Table (IAT) is a structure that contains library function (DLL) names and addresses in memory that a loaded program requires to execute. Rootkits can alter the IAT of a program so that its own function will be called instead of the legitimate function by overwriting the address of the IAT function with the address of its own function loaded into memory space, as illustrated in Figure 2-16. The sold line shows the normal sequence of calls. The dashed line from the IAT to the Rootkit code shows the hooking from the IAT to the Rootkit code.

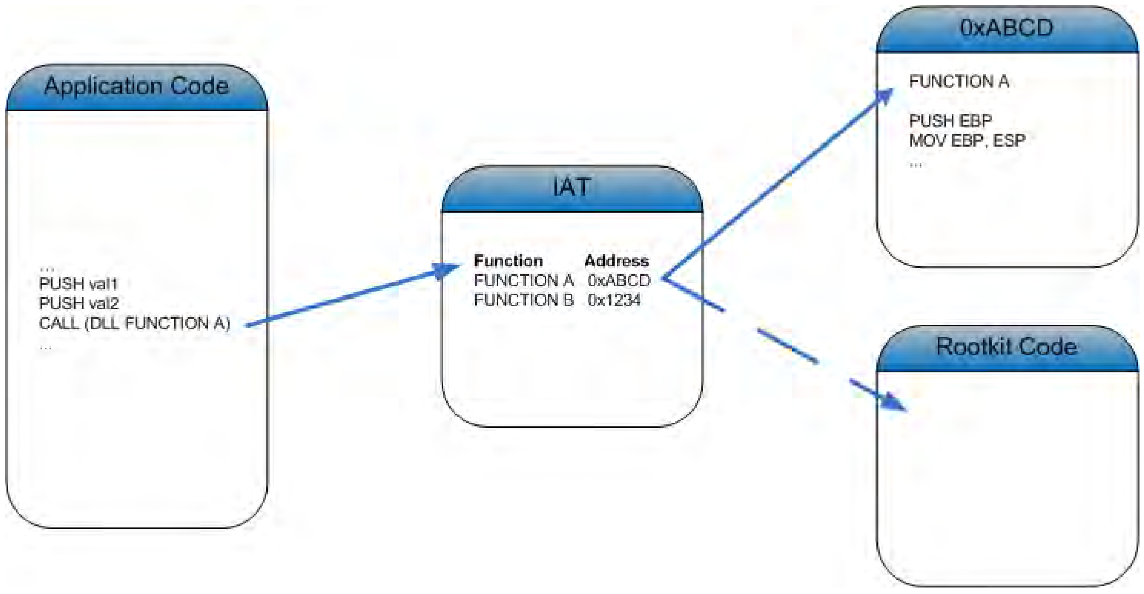


Figure 2-16 Altering the IAT of a program so that rootkit code is called instead (hooking).

2.15.3. Inline Function Hooking

Instead of over writing the address of the DLL in the IAT, the function code can be directly modified in memory and this is known as an inline function hook. This is achieved by over writing the first few instructions of the hooked function with instructions that will jump to the rootkit code. After the rootkit code has completed, it may return the flow of execution to the code that was originally intended to be called.

2.15.4. SSDT Hooking

Hooking the flow of execution can also occur in the kernel by using the SSDT in a fashion similar to IAT hooking. The original functions address can be replaced by the rootkit function. Functions that return results of open ports or list running processes, can be subverted and allow the presence of the rootkit to remain undetected.

2.15.5. Direct Kernel Object Manipulation

Tools exist for detecting the hooks installed by rootkits, such as *Root Kit Revealer* (Microsoft, 2008c) and a more advanced method to hide processes is to alter the kernel memory data structures that are used for keeping track of the state of the operating system itself. This is known as Direct Kernel Object Manipulation (DKOM) and is hard to detect because “directly modifying the raw main memory contents with a Ring 0 rootkit cannot be controlled by any built-in security mechanism in Windows” (Schwittay, 2006, p. 80). These undocumented data structures contain lists of running processes, threads scheduled for execution and other data. A disadvantage of using DKOM is that it may make the system unstable or even crash. It is especially easy to crash because the actual structure is undocumented and minor operating system changes could change the way the operating system defines and uses the structures. Processes can be hidden by manipulating the in memory data structures that use forward and backward pointers to keep track of processes by reorganizing the pointers of these doubly linked lists. “Because the scheduling of processes does not depend on a process being present in that list, this technique hides the process successfully (e.g. From the Task Manager), but the process is still executed unnoticed” (Schwittay, 2006, p. 80). Figure 2-17, adapted from Schwittay, shows the normal linking between data structures in the top half of the diagram. The lower half of the diagram shows how the middle process is hidden by manipulating the pointers.

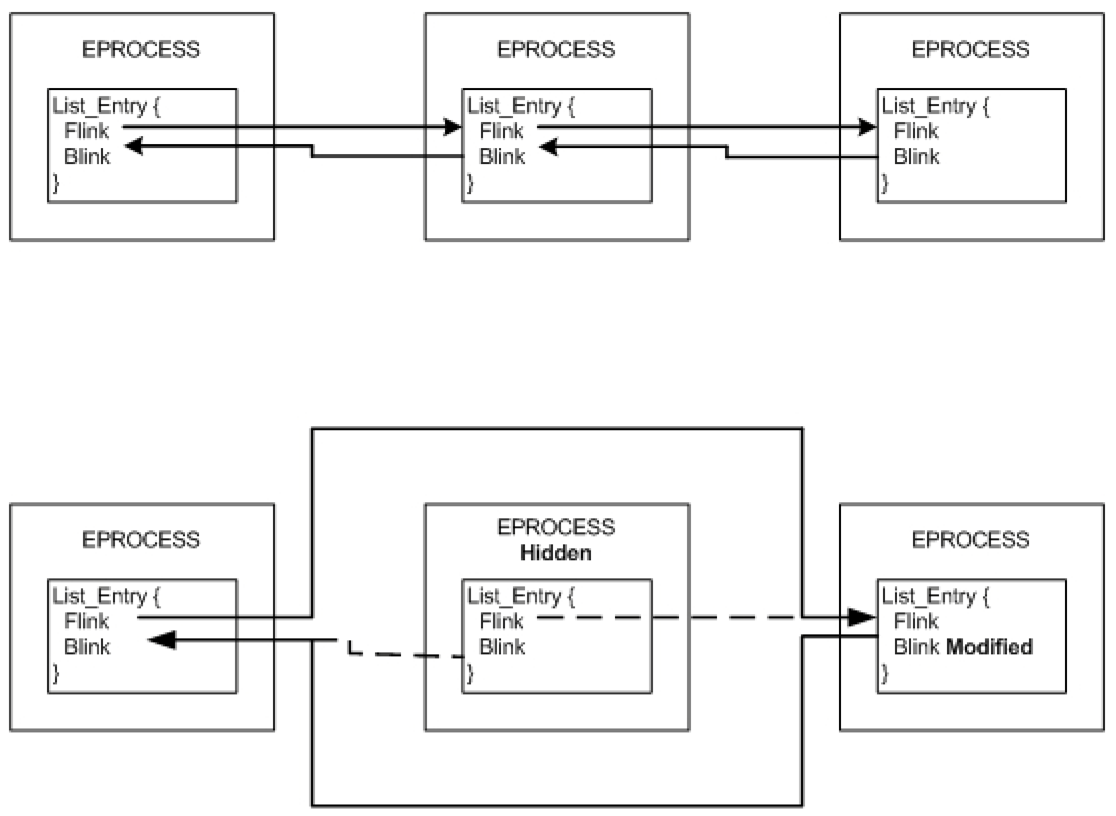


Figure 2-17 Using DKOM pointer manipulation to hide a process (Schwittay, 2006, p. 80)

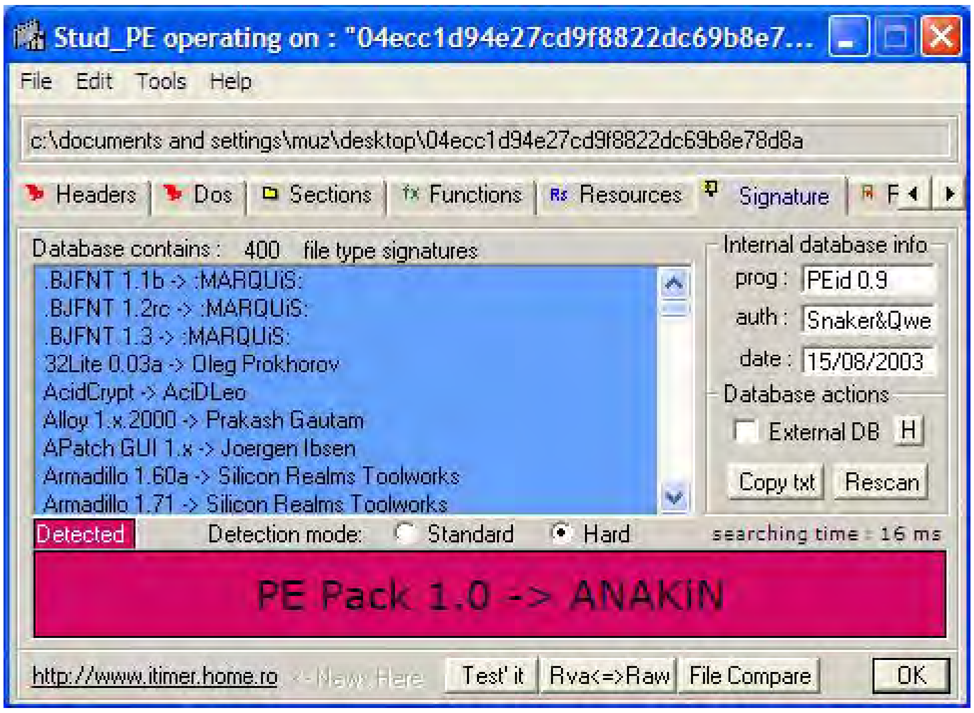
2.16. PACKERS AND PROTECTORS

Packers make static analysis of the binary difficult because the actual code instructions and data is not able to be read until the code has been unpacked. It is very similar to compression. Unpackers exist for many packers in the form of scripts, plugins, programs and in the form of advice on how to unpack manually with the use of a debugger. The unpacked code can then be analyzed with a debugger such as *IDA Pro*, or *Ollydbg*. If malware to be analyzed has been packed by an unknown packer, it can often be loaded into memory, and then process dumped and examined using tools such as the *Ollydbg* plugin, *LordPE (yoda, 2005a)*, or any other memory dumping tool. It should be noted that the code may use techniques to determine if a debugger is being used and respond by protecting itself using some combination of the anti forensic techniques that have been discussed earlier in this literature review. The analyst needs to be in a position to statically analyse the executable as soon as it has unpacked itself, by starting analysis at the Original Entry Point (OEP), otherwise code can be written over and evidence overlooked. A multitude of packers are available and there are methods for unpacking them. The general steps outlined by Craig (2006) for unpacking are:

1. Locate the OEP.
2. Dump the executable image.
3. Change the Entry Point of the dumped image.
4. Calculate the Entry Point Relative Virtual Address (RVA).

   Where $\boxed{\text{RVA EP = OEP – Base Image}}$
5. Fix the Import Address Table (IAT).
6. Reinsert the fixed IAT into the dumped executable.
7. Execute the binary (break at EP), and the binary will populate the IAT with the correct values.

Packer signatures can be detected by tools such as *Stud_PE* (CG SoftLabs, 2008). Figure 2-18 displays the signature view of a malware specimen, and shows that the packer used is *PE Pack 1.0* (ANAKiN, 2005). Note that *Stud_PE* in this case is using the *PEiD* packer signature database. The *PEiD* database contains over four hundred signatures, but is starting to become dated.



c:\documents and settings\muz\desktop\04ecc1d94e27cd9f8822dc69b8e78d8a

Database contains : 400 file type signatures

```
.BJFNT 1.1b -> :MARQUiS:
.BJFNT 1.2rc -> :MARQUiS:
.BJFNT 1.3 -> :MARQUiS:
32Lite 0.03a -> Oleg Prokhorov
AcidCrypt -> AciDLeo
Alloy 1.x.2000 -> Prakash Gautam
APatch GUI 1.x -> Joergen Ibsen
Armadillo 1.60a -> Silicon Realms Toolworks
Armadillo 1.71 -> Silicon Realms Toolworks
```

Internal database info
prog : PEid 0.9
auth : Snaker&Qwe
date : 15/08/2003

Database actions
External DB H

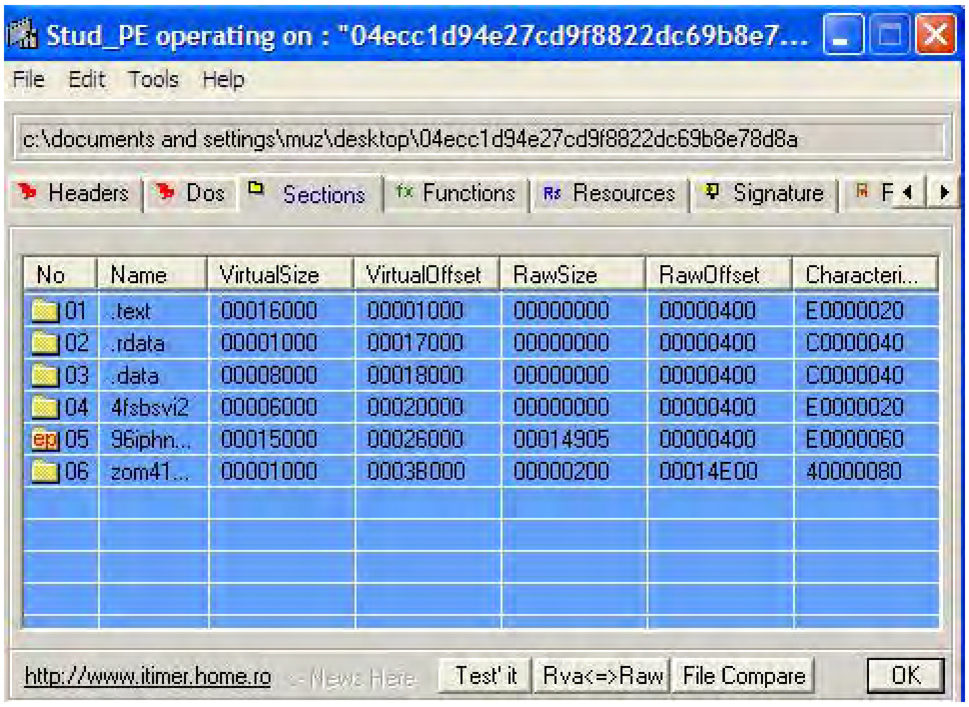
Detected Detection mode: Standard [✓ Hard] searching time : 16 ms

PE Pack 1.0 -> ANAKiN

http://www.itimer.home.ro <- New: Here

Figure 2-18 Screen shot of Stud_PE showing detection of PE Pack signature

Figure 2-19 displays a view of the sections contained in a malware sample using *Stud_PE.* "Sections contain executable code, data, debugging information, resources and additional metadata used by the program" (Harbour, 2007, p. 13).



c:\documents and settings\muz\desktop\04ecc1d94e27cd9f8822dc69b8e78d8a

No	Name	VirtualSize	VirtualOffset	RawSize	RawOffset	Characteri...
01	.text	00016000	00001000	00000000	00000400	E0000020
02	.rdata	00001000	00017000	00000000	00000400	C0000040
03	.data	00008000	00018000	00000000	00000400	C0000040
04	4fsbsvi2	00006000	00020000	00000000	00000400	E0000020
05	96iphn...	00015000	00026000	00014905	00000400	E0000060
06	zom41...	00001000	0003B000	00000200	00014E00	40000080

http://www.itimer.home.ro

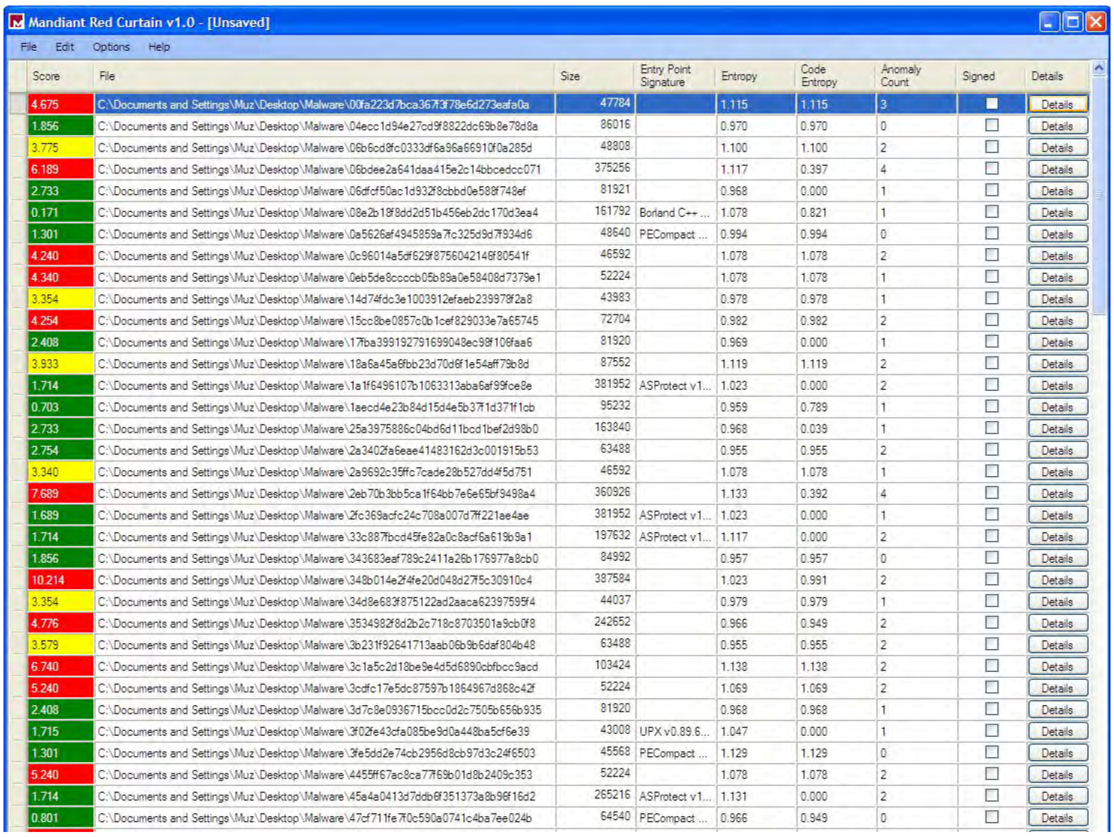
Figure 2-19 Screen shot of Stud_PE showing useful information on sections

Another way of recognizing a packed file is that the first section could have a physical size of 0 bytes. This section will be filled with data that will be unpacked from another section (Falliere, 2006, p. 1). Once unpacked, the classic entry point can be recognized as follows in Figure 2-20.

```
PUSH EBP
MOV EBP, ESP
```

Figure 2-20 Classic entry point signature for recognition purposes.

Harbour (2007, p. 72) points out that a custom packer will likely defeat low level reversers, and that a binary packed by a custom packer is unlikely to be identified at all. The Executable Toolkit, *exetk* (Anonymous, n.d.-c) is a custom packer that is available with source code. Harbour (2007, p. 72) says that tools such as *PeiD* are easily fooled and recommends using *Mandiant Red Curtain (MRC)* (Mandiant, 2007) for detecting packed binaries. MRC examines and scores executable files based on a set of criteria including entropy (randomness), detection of packers, compiler and packer signatures to develop a threat score on how suspicious the file is. This score can then be used to determine if a file should be further examined. A screen shot of *MRC* is shown in Figure 2-21. Useful columns include the threat score, the Entry Point Signature (Packer Signature), the entropy of the entire program, the entropy of the code and a count of the anomalies found.



Score	File	Size	Entry Point Signature	Entropy	Code Entropy	Anomaly Count	Signed	Details
4.675	C:\Documents and Settings\Muz\Desktop\Malware\00fa223d7bca367f3f78e6d273eafa0a	47784		1.115	1.115	3	☐	Details
1.856	C:\Documents and Settings\Muz\Desktop\Malware\04ecc1d94e27cd9f8822dc69b8e78d8a	86016		0.970	0.970	0	☐	Details
3.775	C:\Documents and Settings\Muz\Desktop\Malware\06b6cd8fc0333df6a96a66910f0a285d	48808		1.100	1.100	2	☐	Details
6.189	C:\Documents and Settings\Muz\Desktop\Malware\06bdee2a641daa415e2c14bbcedcc071	375256		1.117	0.397	4	☐	Details
2.733	C:\Documents and Settings\Muz\Desktop\Malware\06dfcf50ac1d932f8cbbd0e588f748ef	81921		0.968	0.000	1	☐	Details
0.171	C:\Documents and Settings\Muz\Desktop\Malware\08e2b18f8dd2d51b456eb2dc170d3ea4	161792	Borland C++ ...	1.078	0.821	1	☐	Details
1.301	C:\Documents and Settings\Muz\Desktop\Malware\0a5626af4945859a7fc325d9d7f934d6	48640	PECompact ...	0.994	0.994	0	☐	Details
4.240	C:\Documents and Settings\Muz\Desktop\Malware\0c96014a5df629f8756042146f80541f	46592		1.078	1.078	2	☐	Details
4.340	C:\Documents and Settings\Muz\Desktop\Malware\0eb5de8ccccb05b89a0e58408d7379e1	52224		1.078	1.078	1	☐	Details
3.354	C:\Documents and Settings\Muz\Desktop\Malware\14d74fdc3e1003912efaeb239978f2a8	43983		0.978	0.978	1	☐	Details
4.254	C:\Documents and Settings\Muz\Desktop\Malware\15cc8be0857c0b1cef829033e7a65745	72704		0.982	0.982	2	☐	Details
2.408	C:\Documents and Settings\Muz\Desktop\Malware\17fba399192791699048ec98f106faa6	81920		0.969	0.000	1	☐	Details
3.933	C:\Documents and Settings\Muz\Desktop\Malware\18a6a45a6fbb23d70d6f1e54aff79b8d	87552		1.119	1.119	2	☐	Details
1.714	C:\Documents and Settings\Muz\Desktop\Malware\1a1f6496107b1063313aba6af99fce8e	381952	ASProtect v1...	1.023	0.000	2	☐	Details
0.703	C:\Documents and Settings\Muz\Desktop\Malware\1aecd4e23b84d15d4e5b37f1d371f1cb	95232		0.959	0.789	1	☐	Details
2.733	C:\Documents and Settings\Muz\Desktop\Malware\25a3975886c04bd6d11bcd1bef2d98b0	163840		0.968	0.039	1	☐	Details
2.754	C:\Documents and Settings\Muz\Desktop\Malware\2a3402fa6eae41483162d3c001915b53	63488		0.955	0.955	2	☐	Details
3.340	C:\Documents and Settings\Muz\Desktop\Malware\2a9692c35ffc7cade28b527dd4f5d751	46592		1.078	1.078	1	☐	Details
7.689	C:\Documents and Settings\Muz\Desktop\Malware\2eb70b3bb5ca1f64bb7e6e65bf9498a4	360926		1.133	0.392	4	☐	Details
1.689	C:\Documents and Settings\Muz\Desktop\Malware\2fc369acfc24c708a007d7ff221ae4ae	381952	ASProtect v1...	1.023	0.000	1	☐	Details
1.714	C:\Documents and Settings\Muz\Desktop\Malware\33c887fbcd45fe82a0c8acf6a619b9a1	197632	ASProtect v1...	1.117	0.000	2	☐	Details
1.856	C:\Documents and Settings\Muz\Desktop\Malware\343683eaf789c2411a26b176977a8cb0	84992		0.957	0.957	0	☐	Details
10.214	C:\Documents and Settings\Muz\Desktop\Malware\348b014e2f4fe20d048d27f5c30910c4	387584		1.023	0.991	2	☐	Details
3.354	C:\Documents and Settings\Muz\Desktop\Malware\34d8e683f875122ad2aaca62397595f4	44037		0.979	0.979	1	☐	Details
4.776	C:\Documents and Settings\Muz\Desktop\Malware\3534982f8d2b2c718c8703501a9cb0f8	242652		0.966	0.949	2	☐	Details
3.579	C:\Documents and Settings\Muz\Desktop\Malware\3b231f92641713aab06b9b6daf804b48	63488		0.955	0.955	2	☐	Details
6.740	C:\Documents and Settings\Muz\Desktop\Malware\3c1a5c2d18be9e4d5d6890cbfbcc9acd	103424		1.138	1.138	2	☐	Details
5.240	C:\Documents and Settings\Muz\Desktop\Malware\3cdfc17e5dc87597b1864967d868c42f	52224		1.069	1.069	2	☐	Details
2.408	C:\Documents and Settings\Muz\Desktop\Malware\3d7c8e0936715bcc0d2c7505b656b935	81920		0.968	0.968	1	☐	Details
1.715	C:\Documents and Settings\Muz\Desktop\Malware\3f02fe43cfa085be9d0a448ba5cf6e39	43008	UPX v0.89.6...	1.047	0.000	1	☐	Details
1.301	C:\Documents and Settings\Muz\Desktop\Malware\3fe5dd2e74cb2956d8cb97d3c24f6503	45568	PECompact ...	1.129	1.129	0	☐	Details
5.240	C:\Documents and Settings\Muz\Desktop\Malware\4455ff67ac8ca77f69b01d8b2409c353	52224		1.078	1.078	2	☐	Details
1.714	C:\Documents and Settings\Muz\Desktop\Malware\45a4a0413d7ddb6f351373a8b96f16d2	265216	ASProtect v1...	1.131	0.000	2	☐	Details
0.801	C:\Documents and Settings\Muz\Desktop\Malware\47cf711fe7f0c590a0741c4ba7ee024b	64540	PECompact ...	0.966	0.949	0	☐	Details

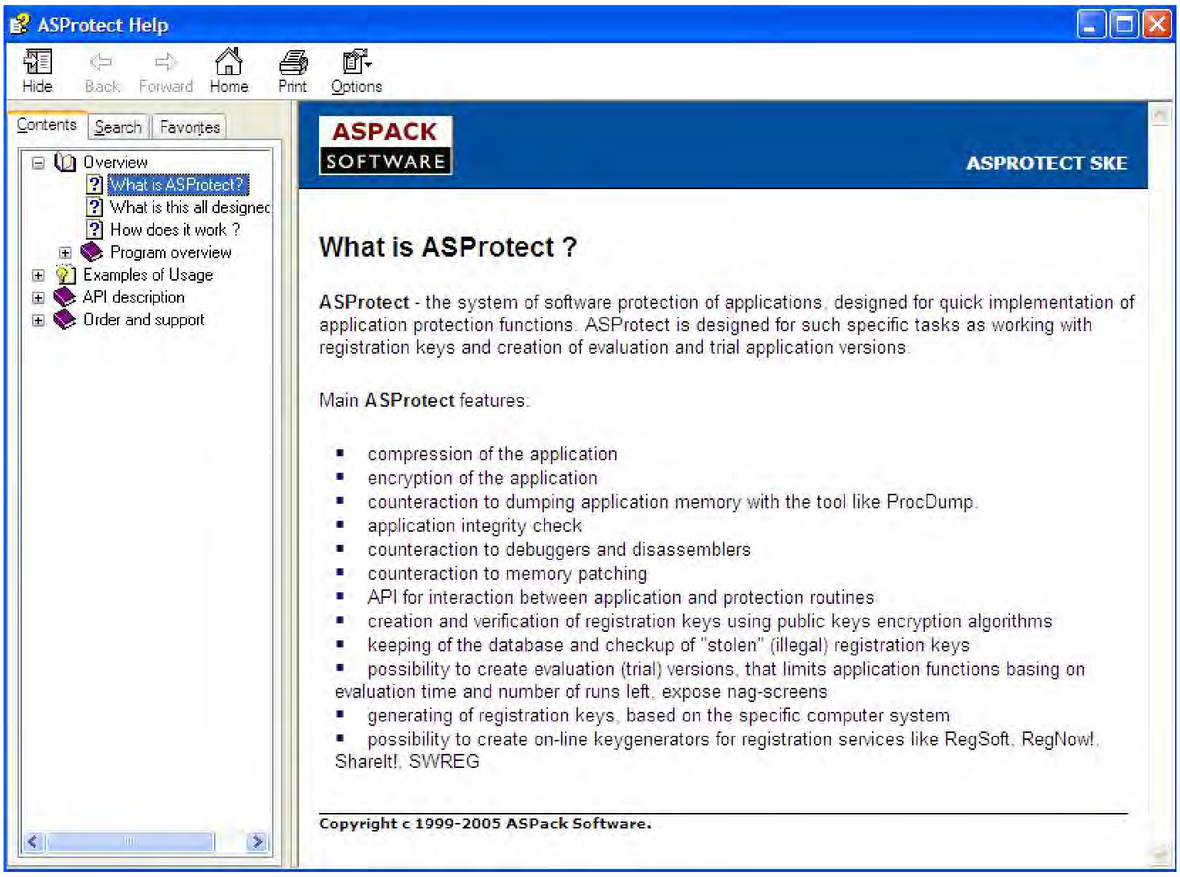
Figure 2-21 Mandiant Red Curtain screen shot showing useful information including entropy and anomaly count

Lyda and Hamrock (2007) explain that entropy is a method for measuring uncertainty in a series of bytes, and although a file compressed with a software compressor may have a high entropy level, the data is structured and is not random. In contrast, measuring the entropy of packed malware measures the lack of structure in the packed malware. The packer typically modifies the original programs standard sections (.text, .data, .rsrc) and compresses these sections into one or two new sections. Lyda *et al*. performed a series of controlled experiments to compute the entropy of 21,567 Windows based malware samples collected between January 2000 and December 2005 and found that entropy measurement was very effective at identifying packed malware. This approach is supported as effective at detecting packed malware by other researchers such as Ebringer and Sun (2008).

2.16.1. ASProtect

ASProtect is a popular, commercial packer and protector that is used to obfuscate demo programs and shareware. Protectors differ from packers by incorporating encryption features. It is also used by malware authors to deter and hinder AV software and malware analysts from analyzing their code. It inserts anti debugging code into the binaries it is packing/protecting and can insert registration schemes and time limits. Run time tracing can be made complicated by exploiting Microsoft Windows Structured Exception Handling (SEH) scheme. It can also use techniques to hinder the dumping of memory. Dumping of memory can be useful for the malware analyst by letting obfuscated programs unpack themselves as they run, catching and halting the program at the moment the unpacking stops, and then dumping the unpacked program in memory. This allows the code to then be analyzed. *ASProtect* can make this dumping process less useful by deleting a section of code as soon as it has finished executing. This technique is known as "stolen bytes". These bytes must be restored if the dumped program is to be run again. The extensive range of features that *ASProtect* can incorporate is listed in the screen shot of Figure 2-22. Figure 2-23 displays the dialog that allows the selection of features that can be incorporated into the code and shows this this is as simple as selecting check boxes. Figure 2-24 shows a screen shot of the dialog box displayed at the end of the packing and protection implementation routine. It shows that the original, 6k byte file has grown to 305k bytes with the addition of CRC check protection, anti debugging and IAT protection.

The view of the OEP in OllyDbg is shown in the screen shot of Figure 2-25 before ASProtect is applied to the program. The code and function calls can be easily read and followed. The original IAT is shown in the screen shot of Figure 2-26 and the imports can be easily read as well, before the application of *ASProtect*. In contrast, Figure 2-27 shows the screen shot of *OllyDbg* after the application of *ASProtect* and that the file has grown from 6 KB to 305 KB with the addition of protection such as CRC code checking and anti debugging. The obfuscation introduced by the protector is clearly evident and demonstrates that the code has to be unpacked and unprotected before analysis can begin.



What is ASProtect ?

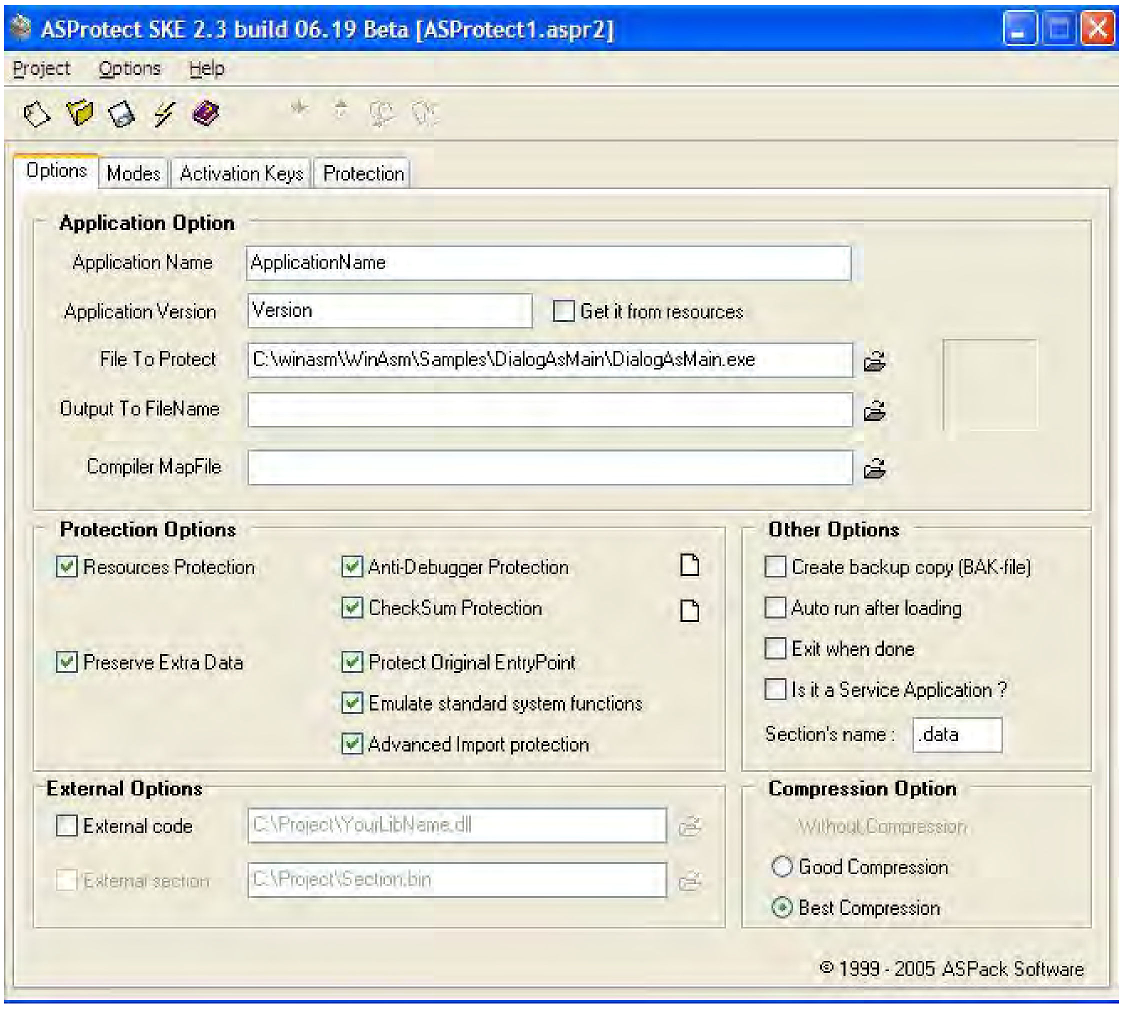
ASProtect - the system of software protection of applications, designed for quick implementation of application protection functions. ASProtect is designed for such specific tasks as working with registration keys and creation of evaluation and trial application versions.

Main **ASProtect** features:

- compression of the application
- encryption of the application
- counteraction to dumping application memory with the tool like ProcDump.
- application integrity check
- counteraction to debuggers and disassemblers
- counteraction to memory patching
- API for interaction between application and protection routines
- creation and verification of registration keys using public keys encryption algorithms
- keeping of the database and checkup of "stolen" (illegal) registration keys
- possibility to create evaluation (trial) versions, that limits application functions basing on evaluation time and number of runs left, expose nag-screens
- generating of registration keys, based on the specific computer system
- possibility to create on-line keygenerators for registration services like RegSoft, RegNow!, ShareIt!, SWREG

Copyright c 1999-2005 ASPack Software.

Figure 2-22 List of ASProtect Features



Application Option

Application Name: ApplicationName

Application Version: Version

[ ] Get it from resources

File To Protect: C:\winasm\WinAsm\Samples\DialogAsMain\DialogAsMain.exe

Output To FileName:

Compiler MapFile:

Protection Options

[✓] Resources Protection

[✓] Anti-Debugger Protection

[✓] CheckSum Protection

[✓] Preserve Extra Data

[✓] Protect Original EntryPoint

[✓] Emulate standard system functions

[✓] Advanced Import protection

Other Options

[ ] Create backup copy (BAK-file)

[ ] Auto run after loading

[ ] Exit when done

[ ] Is it a Service Application ?

Section's name : .data

External Options

[ ] External code: C:\Project\YourLibName.dll

[ ] External section: C:\Project\Section.bin

Compression Option

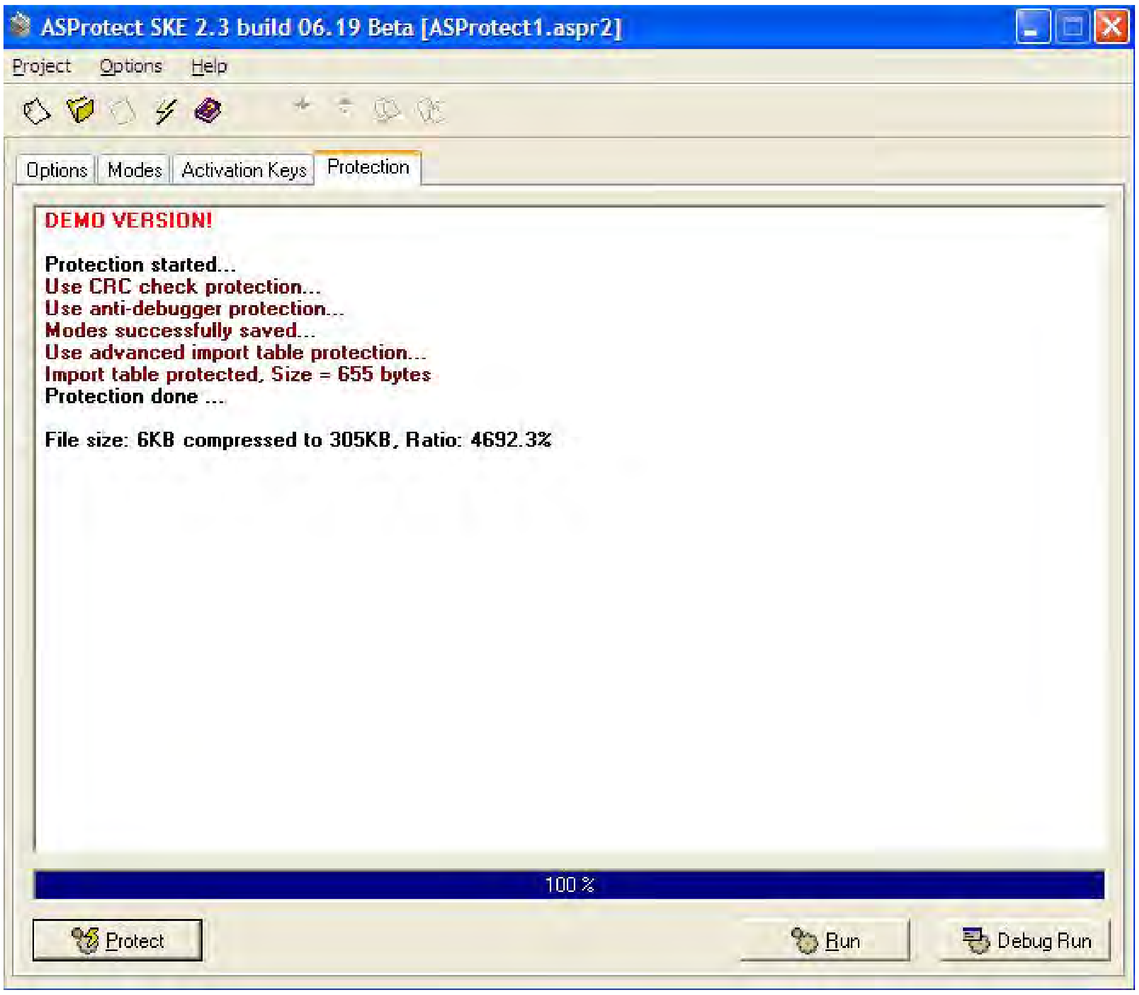
Without Compression

( ) Good Compression

(●) Best Compression

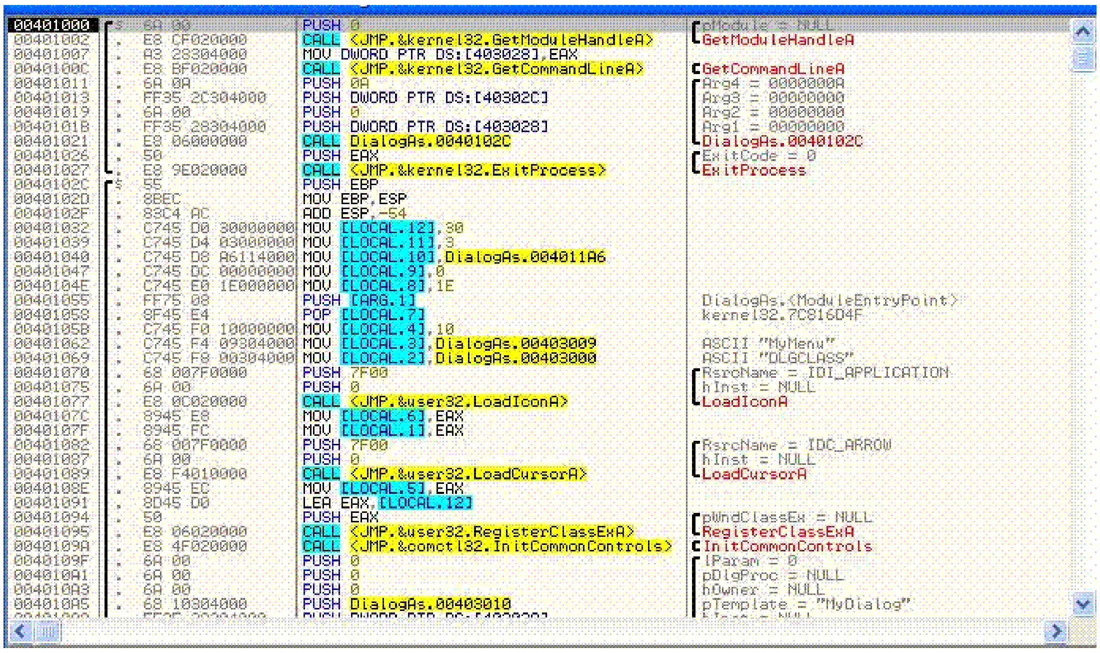
© 1999 - 2005 ASPack Software

Figure 2-23 Dialog showing range of available options in ASProtect to protect code and hinder analysis.



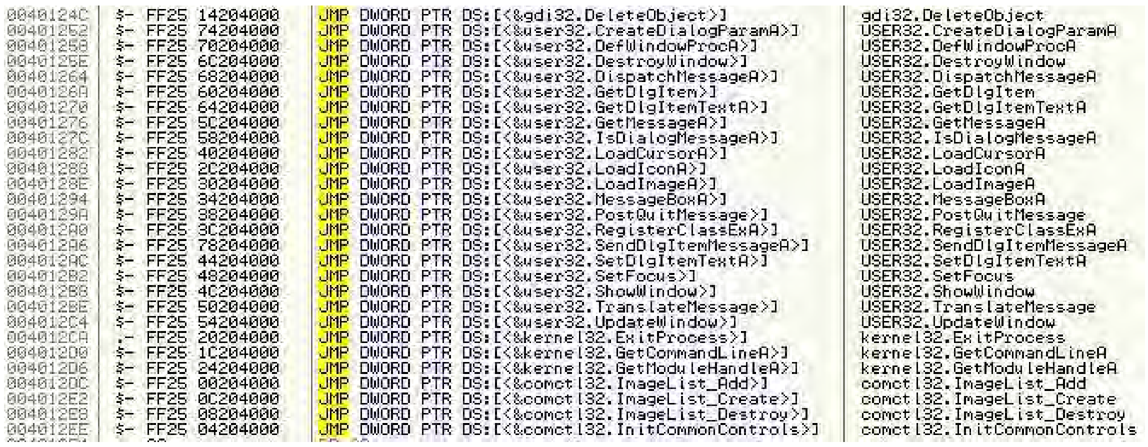
```
DEMO VERSION!

Protection started...
Use CRC check protection...
Use anti-debugger protection...
Modes successfully saved...
Use advanced import table protection...
Import table protected, Size = 655 bytes
Protection done ...

File size: 6KB compressed to 305KB, Ratio: 4692.3%
```

Figure 2-24 ASProtect completion showing the file size has grown markedly with added protection.



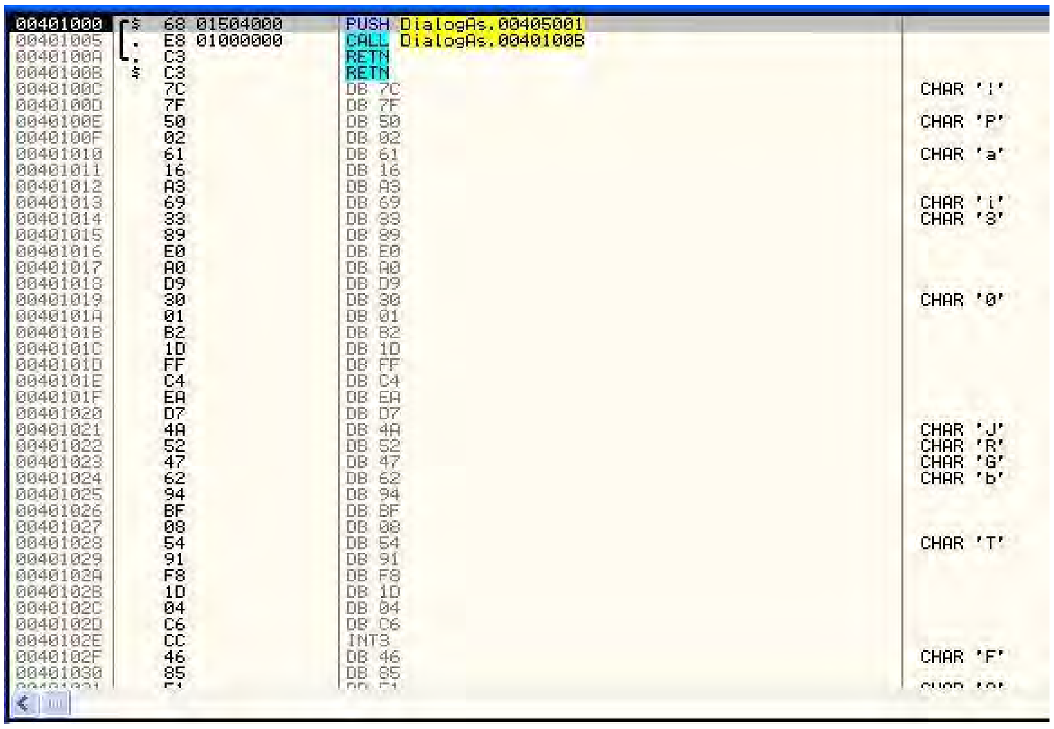
```
00401000 $  6A 00           PUSH 0                                      pModule = NULL
00401002 .  E8 CF020000     CALL <JMP.&kernel32.GetModuleHandleA>       GetModuleHandleA
00401007 .  A3 28304000     MOV DWORD PTR DS:[403028],EAX
0040100C .  E8 BF020000     CALL <JMP.&kernel32.GetCommandLineA>        GetCommandLineA
00401011 .  6A 0A           PUSH 0A                                     Arg4 = 0000000A
00401013 .  FF35 2C304000   PUSH DWORD PTR DS:[40302C]                  Arg3 = 00000000
00401019 .  6A 00           PUSH 0                                      Arg2 = 00000000
0040101B .  FF35 28304000   PUSH DWORD PTR DS:[403028]                  Arg1 = 00000000
00401021 .  E8 06000000     CALL DialogAs.0040102C                      DialogAs.0040102C
00401026 .  50              PUSH EAX                                    ExitCode = 0
00401027 .  E8 9E020000     CALL <JMP.&kernel32.ExitProcess>            ExitProcess
0040102C $  55              PUSH EBP
0040102D .  8BEC            MOV EBP,ESP
0040102F .  83C4 AC         ADD ESP,-54
00401032 .  C745 D0 30000000 MOV [LOCAL.12],30
00401039 .  C745 D4 03000000 MOV [LOCAL.11],3
00401040 .  C745 D8 A6114000 MOV [LOCAL.10],DialogAs.004011A6
00401047 .  C745 DC 00000000 MOV [LOCAL.9],0
0040104E .  C745 E0 1E000000 MOV [LOCAL.8],1E
00401055 .  FF75 08         PUSH [ARG.1]                                DialogAs.<ModuleEntryPoint>
00401058 .  8F45 E4         POP [LOCAL.7]                               kernel32.7C816D4F
0040105B .  C745 F0 10000000 MOV [LOCAL.4],10
00401062 .  C745 F4 09304000 MOV [LOCAL.3],DialogAs.00403009            ASCII "MyMenu"
00401069 .  C745 F8 00304000 MOV [LOCAL.2],DialogAs.00403000            ASCII "DLGCLASS"
00401070 .  68 007F0000     PUSH 7F00                                   RsrcName = IDI_APPLICATION
00401075 .  6A 00           PUSH 0                                      hInst = NULL
00401077 .  E8 0C020000     CALL <JMP.&user32.LoadIconA>                LoadIconA
0040107C .  8945 E8         MOV [LOCAL.6],EAX
0040107F .  8945 FC         MOV [LOCAL.1],EAX
00401082 .  68 007F0000     PUSH 7F00                                   RsrcName = IDC_ARROW
00401087 .  6A 00           PUSH 0                                      hInst = NULL
00401089 .  E8 F4010000     CALL <JMP.&user32.LoadCursorA>              LoadCursorA
0040108E .  8945 EC         MOV [LOCAL.5],EAX
00401091 .  8D45 D0         LEA EAX,[LOCAL.12]
00401094 .  50              PUSH EAX                                    pWndClassEx = NULL
00401095 .  E8 06020000     CALL <JMP.&user32.RegisterClassExA>         RegisterClassExA
0040109A .  E8 4F020000     CALL <JMP.&comctl32.InitCommonControls>     InitCommonControls
0040109F .  6A 00           PUSH 0                                      lParam = 0
004010A1 .  6A 00           PUSH 0                                      pDlgProc = NULL
004010A3 .  6A 00           PUSH 0                                      hOwner = NULL
004010A5 .  68 10304000     PUSH DialogAs.00403010                      pTemplate = "MyDialog"
```

Figure 2-25 Original Entry Point clearly evident in OllyDbg before protection.



```
0040124C  $-  FF25 14204000   JMP DWORD PTR DS:[<&gdi32.DeleteObject>]              gdi32.DeleteObject
00401252  $-  FF25 74204000   JMP DWORD PTR DS:[<&user32.CreateDialogParamA>]       USER32.CreateDialogParamA
00401258  $-  FF25 70204000   JMP DWORD PTR DS:[<&user32.DefWindowProcA>]           USER32.DefWindowProcA
0040125E  $-  FF25 6C204000   JMP DWORD PTR DS:[<&user32.DestroyWindow>]            USER32.DestroyWindow
00401264  $-  FF25 68204000   JMP DWORD PTR DS:[<&user32.DispatchMessageA>]         USER32.DispatchMessageA
0040126A  $-  FF25 60204000   JMP DWORD PTR DS:[<&user32.GetDlgItem>]               USER32.GetDlgItem
00401270  $-  FF25 64204000   JMP DWORD PTR DS:[<&user32.GetDlgItemTextA>]          USER32.GetDlgItemTextA
00401276  $-  FF25 5C204000   JMP DWORD PTR DS:[<&user32.GetMessageA>]              USER32.GetMessageA
0040127C  $-  FF25 58204000   JMP DWORD PTR DS:[<&user32.IsDialogMessageA>]         USER32.IsDialogMessageA
00401282  $-  FF25 40204000   JMP DWORD PTR DS:[<&user32.LoadCursorA>]              USER32.LoadCursorA
00401288  $-  FF25 2C204000   JMP DWORD PTR DS:[<&user32.LoadIconA>]                USER32.LoadIconA
0040128E  $-  FF25 30204000   JMP DWORD PTR DS:[<&user32.LoadImageA>]               USER32.LoadImageA
00401294  $-  FF25 34204000   JMP DWORD PTR DS:[<&user32.MessageBoxA>]              USER32.MessageBoxA
0040129A  $-  FF25 38204000   JMP DWORD PTR DS:[<&user32.PostQuitMessage>]          USER32.PostQuitMessage
004012A0  $-  FF25 3C204000   JMP DWORD PTR DS:[<&user32.RegisterClassExA>]         USER32.RegisterClassExA
004012A6  $-  FF25 78204000   JMP DWORD PTR DS:[<&user32.SendDlgItemMessageA>]      USER32.SendDlgItemMessageA
004012AC  $-  FF25 44204000   JMP DWORD PTR DS:[<&user32.SetDlgItemTextA>]          USER32.SetDlgItemTextA
004012B2  $-  FF25 48204000   JMP DWORD PTR DS:[<&user32.SetFocus>]                 USER32.SetFocus
004012B8  $-  FF25 4C204000   JMP DWORD PTR DS:[<&user32.ShowWindow>]               USER32.ShowWindow
004012BE  $-  FF25 50204000   JMP DWORD PTR DS:[<&user32.TranslateMessage>]         USER32.TranslateMessage
004012C4  $-  FF25 54204000   JMP DWORD PTR DS:[<&user32.UpdateWindow>]             USER32.UpdateWindow
004012CA  .-  FF25 20204000   JMP DWORD PTR DS:[<&kernel32.ExitProcess>]            kernel32.ExitProcess
004012D0  $-  FF25 1C204000   JMP DWORD PTR DS:[<&kernel32.GetCommandLineA>]        kernel32.GetCommandLineA
004012D6  $-  FF25 24204000   JMP DWORD PTR DS:[<&kernel32.GetModuleHandleA>]       kernel32.GetModuleHandleA
004012DC  $-  FF25 00204000   JMP DWORD PTR DS:[<&comctl32.ImageList_Add>]          comctl32.ImageList_Add
004012E2  $-  FF25 0C204000   JMP DWORD PTR DS:[<&comctl32.ImageList_Create>]       comctl32.ImageList_Create
004012E8  $-  FF25 08204000   JMP DWORD PTR DS:[<&comctl32.ImageList_Destroy>]      comctl32.ImageList_Destroy
004012EE  $-  FF25 04204000   JMP DWORD PTR DS:[<&comctl32.InitCommonControls>]     comctl32.InitCommonControls
```

Figure 2-26 Imports before protection clearly showing imported functions.



```
00401000  $  68 01504000   PUSH DialogAs.00405001
00401005  .  E8 01000000   CALL DialogAs.0040100B
0040100A  .  C3            RETN
0040100B  $  C3            RETN
0040100C     7C            DB 7C                       CHAR '|'
0040100D     7F            DB 7F
0040100E     50            DB 50                       CHAR 'P'
0040100F     02            DB 02
00401010     61            DB 61                       CHAR 'a'
00401011     16            DB 16
00401012     A3            DB A3
00401013     69            DB 69                       CHAR 'i'
00401014     33            DB 33                       CHAR '3'
00401015     89            DB 89
00401016     E0            DB E0
00401017     A0            DB A0
00401018     D9            DB D9
00401019     30            DB 30                       CHAR '0'
0040101A     01            DB 01
0040101B     B2            DB B2
0040101C     1D            DB 1D
0040101D     FF            DB FF
0040101E     C4            DB C4
0040101F     EA            DB EA
00401020     D7            DB D7
00401021     4A            DB 4A                       CHAR 'J'
00401022     52            DB 52                       CHAR 'R'
00401023     47            DB 47                       CHAR 'G'
00401024     62            DB 62                       CHAR 'b'
00401025     94            DB 94
00401026     BF            DB BF
00401027     08            DB 08
00401028     54            DB 54                       CHAR 'T'
00401029     91            DB 91
0040102A     F8            DB F8
0040102B     1D            DB 1D
0040102C     04            DB 04
0040102D     C6            DB C6
0040102E     CC            INT3
0040102F     46            DB 46                       CHAR 'F'
00401030     85            DB 85
```

Figure 2-27 Packed View of Entry Point in OllyDbg showing obfuscation.

2.16.1.1. Unpacking ASProtect

Anthracene (2006) provides an overview on how to deal with some of the features of *ASProtect* and is only a single demonstration of a plethora of informal papers and demonstrations that are available on reverse engineering sites that cater mostly for software crackers. Software crackers use reverse engineering techniques to defeat protection mechanisms of legitimate software to avoid licensing, or to extract information on how software works beneath the hood. Anthracene's treatise is quite extensive and very typical of the step by step advice that is often required to unpack packed software to arrive at the OEP and to repair the IAT so that detailed analysis can be conducted. Essentially, the technique discussed by Anthracene is summarized in the following sequence:

1. Confirm the signature of the packer used, using *PEiD*.
2. Open the file in *OllyDbg*.
3. Set the options in *OllyDbg* to pass all exceptions to the program being debugged. This is because it uses exception handling tricks to try to determine if it is being debugged.
4. Set *OllyDbg* to remove analysis from module. This is because code and data are intertwined.
5. The entry point is characterized by a PUSH and a RETN. This is equivalent to a JMP. Jump to the address.
6. Set a hardware breakpoint on access to the DWORD pointed to by the ESP register. Then hit run.
7. The break could be on a JMP EAX instruction. This could be the jump to the OEP. Step over this instruction, and this could be the OEP. This will be characterized by a typical stack frame setup.
8. Dump the file using the *OllyDump* plugin.
9. Start *ImpRec*, attach to the process being debugged and fill in the OEP.
10. Click on IAT autosearch, click Ok and then click on Get Imports.
11. Repair the Imports (which is a detailed activity in itself).

Although presented above as a simple list of summarized instructions, the details in Anthracene's discussion covers more than 23 pages. This exemplifies the work required to manually unpack, but only hints at what could be considered a much more difficult exercise if more anti-analysis features are added to the protector.

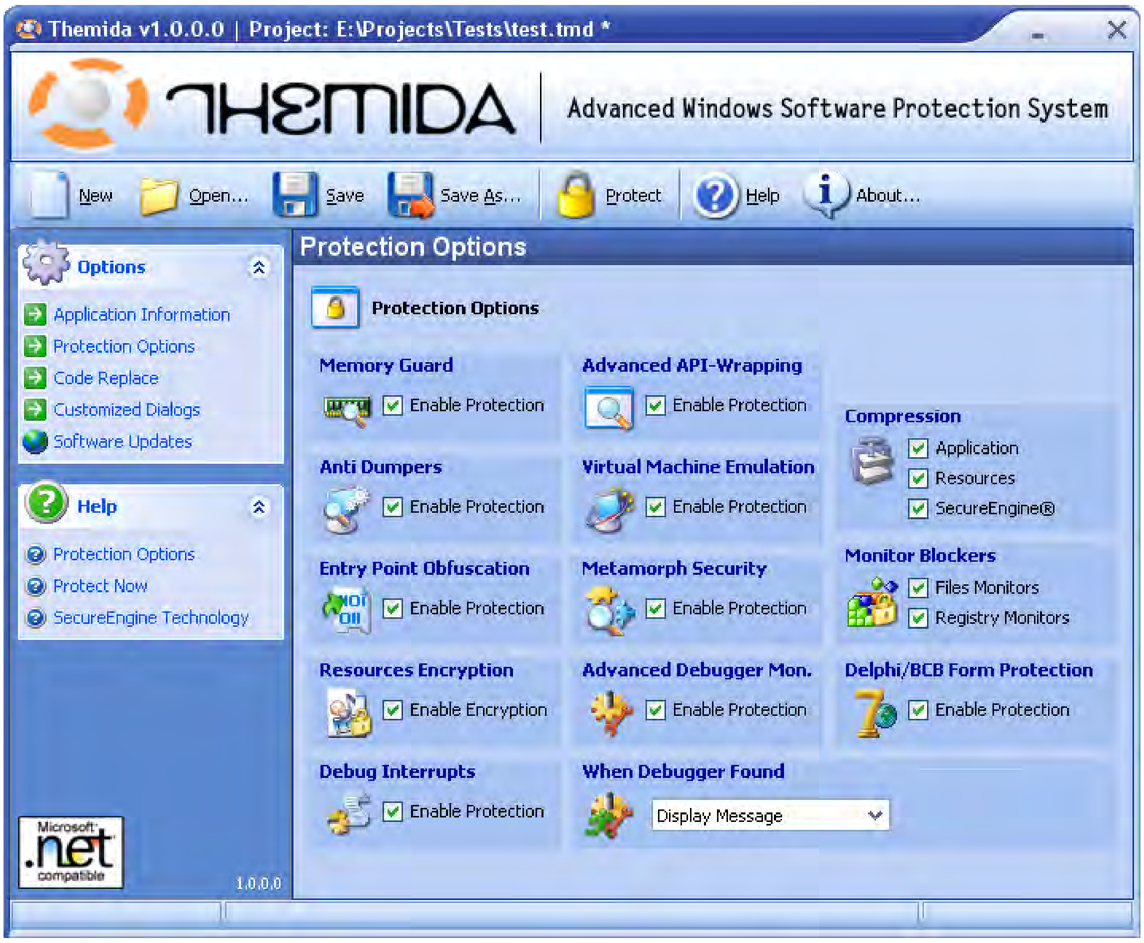
2.16.2. The Problem with Packers

Packing is becoming a dominant problem for AV software because of the number and sophistication of the packers that are now available (Sun et al., 2008, p. 2). Even though scripts and plugins are available for unpacking, they only work when simple packers are used, and fail when sophisticated packers have been used. Such tools often use heuristics to search for the OEP whilst the unpacker is allowed to run. Sun *et al*. propose a method of unpacking by creating an execution trace of the instruction pointer `EIP`, and creating a histogram of the addresses of the executed instructions and ordering them by the last time an address is executed. This is based on their observation that:

a. OEP bytes are invariably only executed once, even in a packed program.
b. Generally, the packer will unpack the original program to a region of memory which has not been executed previously.

The results documented in the paper by Sun *et al*. appear to be very good but only fairly simple packers were examined, including *UPX* (Oberhumer, Molnár, & Reiser, 2008), *Morphine* (Anonymous, n.d.-j)*, MEW* and *FSG (Bart & Xtreeme, 2005)* as well as a multi packer example which packed the file with *UPX 2.03* and then *Morphine 2.7.* Future work identified in the paper includes optimizing the tracer to resist anti-analysis techniques.

Figure 2-28 is a screenshot of the protection options dialog that users of *Themida®* can use to protect their code. An extensive list of options are available that provide coverage of some of the most significant anti-analysis techniques discussed in this literature review.



Protection Options

Protection Options

Memory Guard
☑ Enable Protection

Advanced API-Wrapping
☑ Enable Protection

Compression
☑ Application
☑ Resources
☑ SecureEngine®

Anti Dumpers
☑ Enable Protection

Virtual Machine Emulation
☑ Enable Protection

Entry Point Obfuscation
☑ Enable Protection

Metamorph Security
☑ Enable Protection

Monitor Blockers
☑ Files Monitors
☑ Registry Monitors

Resources Encryption
☑ Enable Encryption

Advanced Debugger Mon.
☑ Enable Protection

Delphi/BCB Form Protection
☑ Enable Protection

Debug Interrupts
☑ Enable Protection

When Debugger Found
Display Message

Figure 2-28 Themida® dialog showing extensive range of protection options.

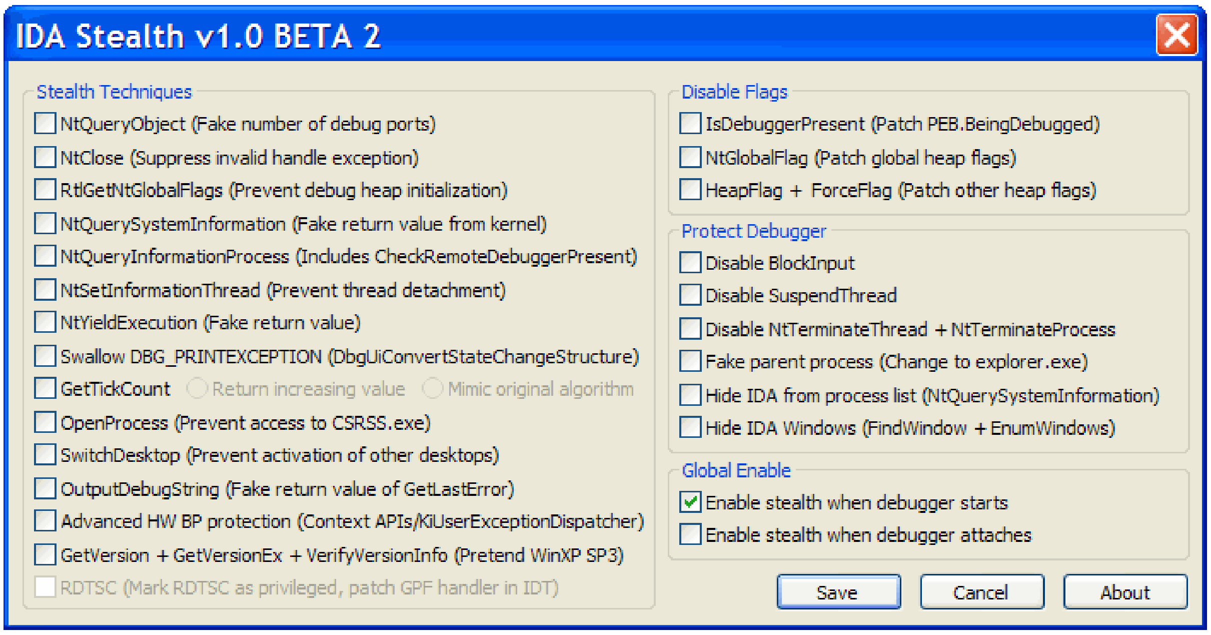
2.17. PLUGINS

Plugins exist for most of the popular tools used for reverse engineering and are typically DLL's that are simply installed to a known directory pre determined by the debugger, which then makes the plugin available via a menu. A variety of plugins are available from the internet, particularly reverse engineering and cracking sites. It is highly conceivable that these plugins contain malicious software themselves and it is advisable to treat them with caution and to analyse the source code for the plugin if it is possible, especially if forensic evidence has been collected using the plugin. The functionality of plugins can be replicated using scripting languages that accompany the most popular disassemblers and debuggers such as *IDA Pro*.

IDA Stealth (Newger, 2008) is a free plugin for *IDA Pro* (Hex-Rays, 2008), a commercial disassembler and debugger. The dialog box for *IDA Stealth* is displayed in Figure 2-29. It lists a limited subset of the techniques discussed by (Ferrie, 2008), who in turn says the 52 techniques discussed in his paper are only the most widely used techniques. The plugin functions are divided into the following sections:

- Stealth Techniques
- Disable Flags
- Protect Debugger
- Global Enable

The particular technique to be used is simply enabled by selecting the appropriate checkbox.


IDA Stealth v1.0 BETA 2

Stealth Techniques
- [ ] NtQueryObject (Fake number of debug ports)
- [ ] NtClose (Suppress invalid handle exception)
- [ ] RtlGetNtGlobalFlags (Prevent debug heap initialization)
- [ ] NtQuerySystemInformation (Fake return value from kernel)
- [ ] NtQueryInformationProcess (Includes CheckRemoteDebuggerPresent)
- [ ] NtSetInformationThread (Prevent thread detachment)
- [ ] NtYieldExecution (Fake return value)
- [ ] Swallow DBG_PRINTEXCEPTION (DbgUiConvertStateChangeStructure)
- [ ] GetTickCount ○ Return increasing value ○ Mimic original algorithm
- [ ] OpenProcess (Prevent access to CSRSS.exe)
- [ ] SwitchDesktop (Prevent activation of other desktops)
- [ ] OutputDebugString (Fake return value of GetLastError)
- [ ] Advanced HW BP protection (Context APIs/KiUserExceptionDispatcher)
- [ ] GetVersion + GetVersionEx + VerifyVersionInfo (Pretend WinXP SP3)
- [ ] RDTSC (Mark RDTSC as privileged, patch GPF handler in IDT)

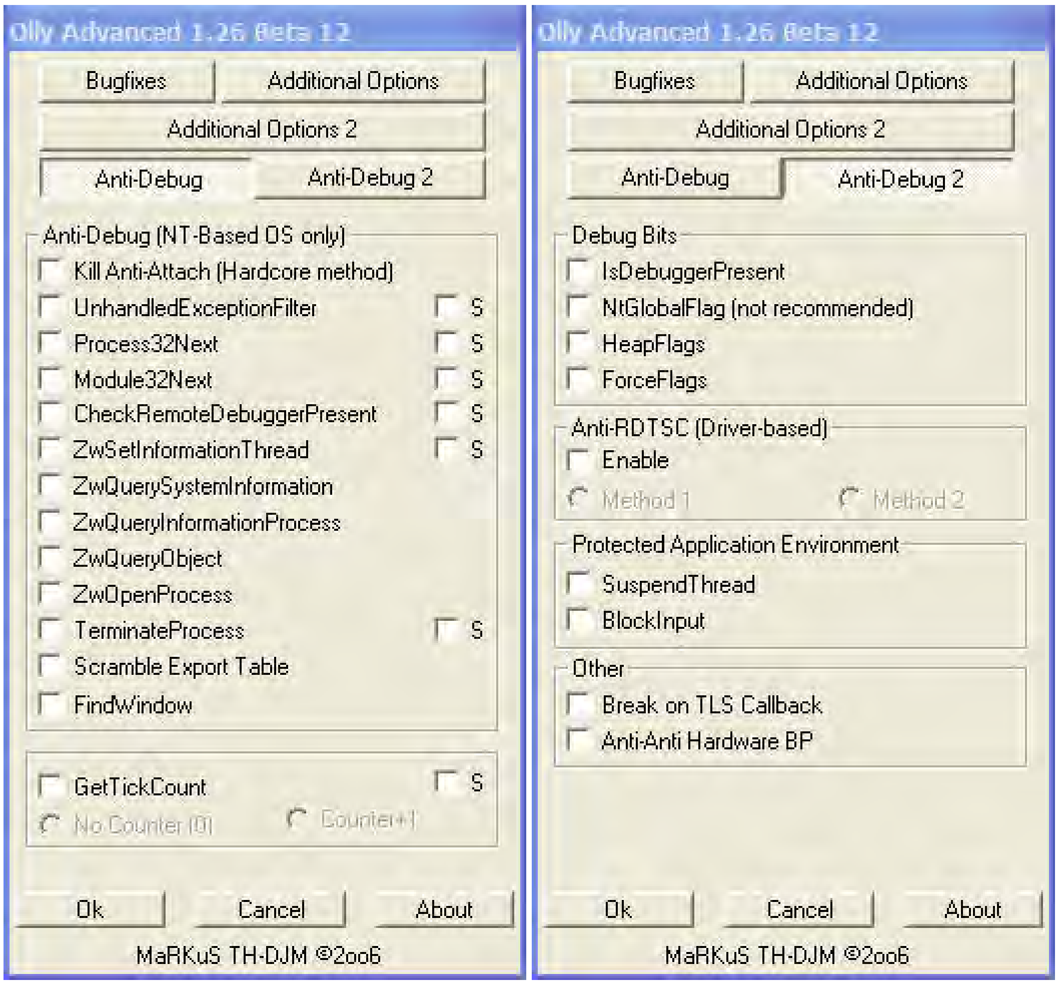
Disable Flags
- [ ] IsDebuggerPresent (Patch PEB.BeingDebugged)
- [ ] NtGlobalFlag (Patch global heap flags)
- [ ] HeapFlag + ForceFlag (Patch other heap flags)

Protect Debugger
- [ ] Disable BlockInput
- [ ] Disable SuspendThread
- [ ] Disable NtTerminateThread + NtTerminateProcess
- [ ] Fake parent process (Change to explorer.exe)
- [ ] Hide IDA from process list (NtQuerySystemInformation)
- [ ] Hide IDA Windows (FindWindow + EnumWindows)

Global Enable
- [✓] Enable stealth when debugger starts
- [ ] Enable stealth when debugger attaches

Figure 2-29 IDA Stealth Plugin showing available options to hide the debugger from only a selection of techniques discussed in the literature review.

OllyAdvanced (TH-DJM, 2006) is a plugin for *OllyDbg* (Yuschuk, 2008) a free disassembler and debugger. *Olly Advanced* is similar to the *IDA Stealth* plugin as depicted in Figure 2-30.


Olly Advanced 1.26 Beta 12

Bugfixes | Additional Options | Additional Options 2 | Anti-Debug | Anti-Debug 2

Anti-Debug (NT-Based OS only)
- [ ] Kill Anti-Attach (Hardcore method)
- [ ] UnhandledExceptionFilter [ ] S
- [ ] Process32Next [ ] S
- [ ] Module32Next [ ] S
- [ ] CheckRemoteDebuggerPresent [ ] S
- [ ] ZwSetInformationThread [ ] S
- [ ] ZwQuerySystemInformation
- [ ] ZwQueryInformationProcess
- [ ] ZwQueryObject
- [ ] ZwOpenProcess
- [ ] TerminateProcess [ ] S
- [ ] Scramble Export Table
- [ ] FindWindow

- [ ] GetTickCount [ ] S
- ( ) No Counter (0) ( ) Counter+1

Ok | Cancel | About

MaRKuS TH-DJM ©2oo6

Olly Advanced 1.26 Beta 12

Bugfixes | Additional Options | Additional Options 2 | Anti-Debug | Anti-Debug 2

Debug Bits
- [ ] IsDebuggerPresent
- [ ] NtGlobalFlag (not recommended)
- [ ] HeapFlags
- [ ] ForceFlags

Anti-RDTSC (Driver-based)
- [ ] Enable
- ( ) Method 1 ( ) Method 2

Protected Application Environment
- [ ] SuspendThread
- [ ] BlockInput

Other
- [ ] Break on TLS Callback
- [ ] Anti-Anti Hardware BP

Ok | Cancel | About

MaRKuS TH-DJM ©2oo6

Figure 2-30 Olly Advanced Plugin showing available options to hide the debugger from only a selection of techniques discussed in the literature review.

Plugins are useful for manual analysis but typically do not tell the operator that the technique that has been selected has been located or mitigated, their main function is to hide the debugger. It is also evident too, that the extensive list of anti-forensic techniques discussed in the sections above, are not fully reflected in the number of options in the plugins. This leaves a gap between what is available and what could be required by the analyst. This gap can be addressed by the use of scripting languages.

2.18. SCRIPTING LANGUAGES

Disassemblers and Debuggers such as *IDA Pro* and *OllyDbg* are supported by scripting languages as well as Application Programming Interfaces (API) for the development of plugins. “Potential uses for scripts are infinite and can range from simple one-liners to full-blown programs that automate common tasks or perform complex analysis functions" (Eagle, 2008a, p. 249). *IDA Pro's* native scripting language is called *IDC* and appears very C like in appearance and is used to query the database that *IDA Pro* stores the file being analyzed in. *IDA Python* (Erdélyi, 2008) is a *Python* plugin for *IDA Pro* that allows the analyst to access the functions of IDC and the full power of *Python*. Similar plugins for other popular scripting languages such as *Perl* and *Ruby* are also available for *IDA Pro*.

Scripting languages for *OllyDbg* (also in the form of plugins) include *OllyScript* (SHaG, 2006) which is very similar in appearance to assembly language. Other plugins include *OllyPerl* (Stewart, 2006) and *OllyPython* (Vilhonen, 2007) that leverage from *Perl* and *Python* respectively. The *Immunity Debugger* (Immunity, 2008) is an extension of *OllyDbg* that is integrated with *Python*.

Existing scripts for *OllyDbg* are plentiful on the web for performing a myriad of analysis and reverse engineering tasks and far exceed those available freely for *IDA Pro*. It is this researcher's conjecture that this is because *OllyDbg* and more recently, the *Immunity Debugger,* have been the favorite tool of software crackers who have a spirit of sharing more prevalent than the commercial users of *IDA Pro*. *IDA Pro* was initially only a disassembler used for performing static analysis and a debugging capability was added in the past few years. The existing scripts for *OllyDbg* include a very wide variety of unpackers which are not only useful in their own right, but also serve as a source of information on how to unpack particular packers. These can also be used as an algorithmic template to implement the routine in other scripting languages such as *IDA Python*.

Scripts written in *IDC* or *IDAPython* can then be run against the *IDA Pro* database, which is stored in an *IDB* (IDA Pro Database) file, or against the original executable itself on the command line, or through the Graphical User Interface (GUI). The *IDB* file saves previous analysis work that has been conducted on the file which can include identification of functions, structures, enumerations and unions as well as any mitigation work against anti forensic techniques and obfuscation. This assists in automating analysis on malicious files. For example, to run an *IDAPython* script with *IDA Pro* on the command line named *walkTheSegments.py* against an *IDB* file named *CheckRemoteDebuggerPresent.idb*, the following would be entered on the command line or in a script as shown in Figure 2-31. This feature greatly assists automation.

```
idag -A -OIDAPython:walkTheSegments.py
CheckRemoteDebuggerPresent.idb
```

Figure 2-31 Calling IDA Pro on the command line to run a IDAPython script assists automation of code analysis.

2.19. TRACING

Scripts and plugins that are used to unpack malware typically allow the malware to unpack itself at run time, and halt execution when the OEP is recognized. Ideally, the analyst can then use a memory dumping tool to capture the unpacked malware in memory and analyse it (Aquilina et al., 2008; Skoudis & Zeltser, 2004; Zeltser, 2007). However, this approach can miss anti-analysis techniques incorporated into the unpacking code. Lau and Svajcer (2008) point out that executable packers such as *Themida®* (Oreans Technologies, 2008) will not unpack underlying code if it detects that it is running inside VMWare and that tracing is very useful to uncover the use of anti-analysis techniques. “Tracing offers a means of logging specific events that occur while a process is executing” (Eagle, 2008a, p. 508). Events can include every instruction that is executed, function calls, register activities or any other parameter of interest that changes as the malware is executed.

Sun *et al*. (2008) also employ tracing to locate the OEP of packed software by creating a histogram of the addresses of executed instructions and ordering the histogram by the last time an instruction is executed. “Decryption, decompression and copying appear as large spikes at the start of the histogram, followed by a flat section, of height one, which is usually the OEP" (Sun et al., 2008). The researchers show good results for analyzing non malicious software packed by various packers.

2.20. NEW PARADIGMS FOR MALWARE DETECTION

AV software, that relies on signature matching and heuristics is recognized by AV researchers to be far less than optimal (Mila Dalla et al., 2008; Szewczyk & Brand, 2008; W. Yan et al., 2008; Z. Yan & Inge, 2008; Zhou & Meador Inge, 2008). This has led to a variety of research to be conducted on alternate techniques for malware detection as discussed in the following subsections.

2.20.1. Statistical Structures

Bilar (2005) shows how malware can be classified by analyzing statistical structures. Three perspectives examined by Bilar, includes assembly instructions, Win 32 API Calls and system dependence graphs. Examination of assembly instructions is primarily a static analysis technique where the frequency distribution of operation codes (opcodes) is developed from the disassembly of the binary. Bilar shows that this technique can be useful to provide a quick identification. Just looking at the most frequent opcodes is a weak predictor. Looking at fourteen of the most infrequently used opcodes such as an interrupt (int) and no operation (nop), it may be possible to classify malware. Bilar suggests that root kits make heavy use of software interrupts whilst viruses make use of the nop instruction for padding sleds. Additional work being carried out in this area includes investigating equivalent opcode substitution effects between compilers and types of opcodes.

2.20.2. Win 32 API Calls

Looking at Win 32 API Calls is an active analysis technique that observes the API calls that a program under investigation makes. These calls are recorded and a count vector is saved into a database. These vectors are then compared to known malware vectors in the database if it is determined that the vectors are related. Bilar (2005, p. 25) claims that this vector classification is successful in classification of malware into a family. The Win

32 API call fingerprint is shown by Bilar (2005, p. 27) to be robust, even though various packers were used.

2.20.3. System Dependence Graphs

System Dependence Graphs is a newly developing static analysis technique described by Bilar (2005, p. 31) that represents control, call and data dependencies of a program through graph modeling. Then graph structures can be used as fingerprints, which assist in the process of identification, classification and prediction of behaviour.

2.20.4. Run Time Behaviour Monitoring

Malware detection and analysis by an investigator can be a labor intensive process using static and active techniques. Due to time constraints and the abilities of the investigator, there is a possibility that critical forensic evidence could be overlooked. To this end, automated malware detection and classification tools are being developed. Lee and Mody (2006, p. 3) "propose an automated classification method based on runtime behavioral data and machine learning". Essentially the run time behaviour of a file is represented by a sequence of events, which is stored in a canonical format in a database. Machine learning is used to recognize patterns and similarities, which are then used to classify new objects. Such an automated system is important because human analysis can be inefficient and time consuming (Lee & Mody, 2006). However, development of algorithms, validation training data for the classification system requires the input from manual analysis.

2.20.5. Obfuscation Detection

Obfuscation is used by legitimate software to protect the Intellectual Property (IP) as well as by authors of malware whose intention is to hide the malware from AV software. Wysopal (2009) suggests that the very presence of obfuscated code could indicate the presence of malware. Wysopal says that if the behaviour of software cannot be verified, then the software could have a malicious nature and could violate the privacy of the user.

2.21. IMPLICATIONS OF THE LITERATURE REVIEW

The search of the literature, directly related to the implementation, detection and mitigation of anti-analysis techniques malware employs, reveals a number of lines of enquiry not fully covered in the literature.

Various methodologies exist for analyzing malware. The more effective methodologies take the presence of analysis avoidance techniques into account and encourage the use of mitigation strategies for them. Zeltser (2007) uses a sequential static and dynamic, phased approach, where he discovers something from each phase that assists with progressing to the next phase to discover more about how the malware works. An effective technique to support the detection and mitigation of analysis avoidance techniques could be to use such an incremental static and dynamic spiral approach, where anti forensic techniques are discovered and mitigated as the analyses of the malware progresses from a high level of perspective down to the most detailed perspective.

An extensive range of anti forensic techniques can be implemented in malware. A non-exhaustive list of techniques can include anti-dumping, anti-debugging, anti-disassembling, anti-virtual machine, anti-online analysis, use of root kits, IAT destruction, anti-tool specific and process injection. Techniques are dispersed amongst the literature and generally only exist as code snippets. This leaves the prospect to fully implement the techniques and validate their use against analysis tools. This also includes an opportunity to determine how effective the tools are against such a large number of techniques. It also provides a chance to determine how the use of the techniques can be detected and mitigated. A variance of anti-analysis taxonomies was revealed in the literature and this provides an opportunity to combine the taxonomies into an overall one.

Before detailed analysis of the code of malware can be examined, the malware has to be unpacked and the OEP reached. Packers are used to compress multiple malware files into one file and are unpacked when installed or at run time by run time unpacking routines. Various tools and methods are available to unpack packed malware but are very dependent on knowing which packer was used. This information may come from a packer signature detector, but tools such as *PEiD* are becoming dated, unless the signature database they rely on are updated with signatures of the latest packers. The use of a packer can be determined by measuring the entropy of the malware, which tends to have very high levels of entropy when packed. Malware collected from the internet could be used to determine the prevalence of the use of packers and protectors.

Plugins exist for popular debuggers that assist in hiding the debugger from some of the anti-forensic techniques discussed above, but their coverage of the number of techniques is limited. A variety of scripting languages that are available for use with the debuggers are available and these can be used to detect, log and mitigate the use of these techniques. This opens the door to examine the existing plugins and to discover how effective scripting languages are at extending the tools to detect and mitigate anti-analysis techniques.

The literature search revealed that researchers claim that traditional AV software is far less than ideal at detecting malware and that alternate methods exist. This provides an opportunity to examine their claims.

CHAPTER 3 RESEARCH METHODS

The research questions examined in this thesis were stated in the Introduction chapter of this thesis as:

1. What techniques can malware use to avoid being analyzed?
2. How can the use of these techniques be detected?
3. How can the use of these techniques be mitigated?

These questions, refined whilst searching the literature, clearly initiated this line of research. Hernon (1991, p. 4) describes research as an inquiry process and lists the aims of research to include the "Discovery or creation of knowledge, or theory building". In addition, Hernon says that another aim of research could be the "Testing, confirmation, revision, refutation of knowledge and theory". Alternatively, Hernon says that the aim of the research could be the "Investigation of a problem for local decision making". Without a doubt, all three research questions for this thesis could have any, some or all of these aims. However, for research to be considered to have been conducted with appropriate rigor, and to be accepted as truth, the process and methods used to arrive at the result must be shown to be justifiable, the line of enquiry to be clearly defined, with traceability all the way from the research questions to the resultant conclusions and claims of contribution to knowledge. The research process itself could be considered as the linking activities that the researcher conducts to connect the research questions to the aims and results of the research via a number of intermediatory phases (Bouma & Ling, 2004, p. 5).

3.1. A MODEL OF THE RESEARCH PROCESS

A possible model of the research process that is discussed and represented in diagrammatic form by Oates (2007, p. 23) is reproduced as Figure 3-1.

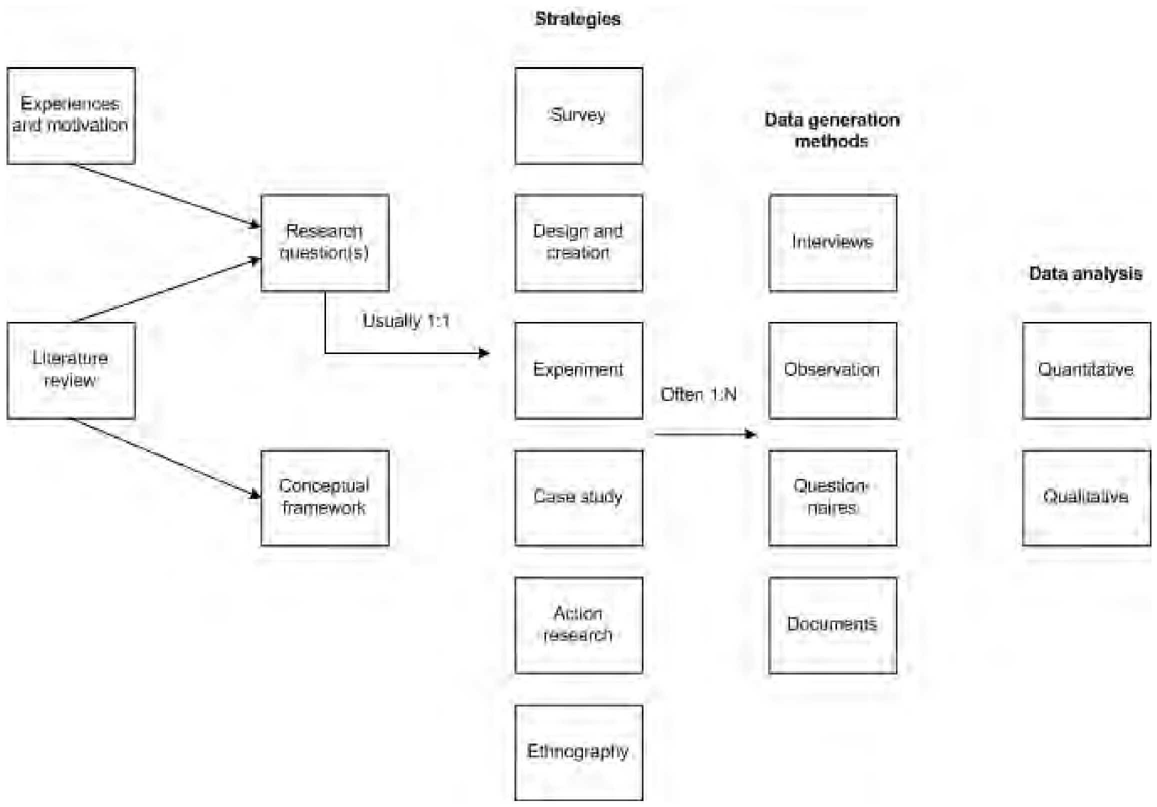


Figure 3-1 Model of the research process showing the variety of paths that can be undertaken (Oates, 2007, p. 23).

The process diagram assists in charting a course to navigate from formulating research questions to discovering answers for the research questions. The particular model presented by Oates shows that experiences, motivation and a literature review are inputs into developing appropriate and meaningful research questions. An objective of this initiating phase of the process is to show why this line of research is important, why it has not been fully addressed in published literature and how the research will be used. The research question is the underlying thread throughout the entire process. After it has been formulated, it is then used to select an appropriate research strategy, data generation method and data analysis method. Research questions clearly have traceability throughout the research process and arriving at answers to an enquiry is dependent upon the selection of an appropriate research strategy, data generation method and data analysis method most appropriate for the questions being asked.

Significant consideration is required to be allocated to the choice of research paradigm before the selection process of research method commences.

Various research paradigms exist to guide the enquiry (Guba & Lincoln, 1994, p. 105; Marshall, 1997, p. 16; Oates, 2007, p. 283). Lincoln and Guba (1994, p. 105) say that “Questions of method are secondary to questions of paradigm, which we define as the basic belief system or worldview that guides the investigator, not only in choices of method but in ontologically and epistemologically fundamental ways .” This is a significant statement, because it emphasizes that in order to conduct research, the researcher must adopt an appropriate and over arching, philosophical viewpoint, referred to as a research paradigm.

3.2. RESEARCH PARADIGMS

Oates (2007, p. 282) describes a paradigm as “a set of shared assumptions or ways of thinking about some aspect of the world”. Various philosophical paradigms exist and have different views about the nature of the world, referred to as ontology, and the way that the knowledge is acquired, referred to as epistemology.

Paradigms can be subdivided further by asking ontological, epistemological and methodological questions (Guba & Lincoln, 1994, p. 108). Epistemology essentially focuses on the theory of knowledge and its acquisition (Carroll & Swatman, 2000). Ontology is concerned with examining the nature of reality from an existence point of view. This philosophical viewpoint asks questions such as “what is?” Epistemology on the other hand, focuses on asking how this knowledge is acquired in the format of questions such as “how do we know what we know?” The methodological question is “how can we come to know it?” (Pickard, 2007, p. 6).

Some common research paradigms include positivism, interpretivism and critical research (Guba & Lincoln, 1994, p. 105; Oates, 2007, p. 283).

3.2.1. Positivism

Oates (2007, p. 283) says positivism is the foundation of the experimental method, which in turn, has two fundamental assumptions:

- The world has order, is regular and is non-random.
- The world can be investigated objectively.

These assumptions are significant because it facilitates the discovery of regularities, patterns and laws through the conduct of experimentation to discover evidence of cause and effect. The discovery process is initiated by the formulation of a hypothesis which is followed by experiments designed to refute or confirm the hypothesis. Confidence in a theory may be gained each time it fails to be refuted. Positivist researchers typically use controlled experiments but they are not limited to the use of controlled experiments as their research strategy. Other strategies such as surveys are also frequently used by this paradigm. Positivists are considered to be reductionist. That is, they study phenomena by breaking them down into simpler components (Easterbrook, Singer, Storey, & Damian, 2008, p. 291).

Guba *et al*. (1994, p. 109) describe the ontology of positivism as a realism and that “an apprehend able reality is assumed to exist, driven by immutable laws and mechanisms”. Guba *et al*. describe the epistemology of positivism as dualist and objectivist. This is because the investigator and the phenomena under investigation are assumed to be independent entities and the investigator studies the object without influencing it, or is influenced by it. Validity is threatened if an influence exists. Guba *et al*. (p. 110) describe the methodology of positivism to be experimental and manipulative. “Questions and/or hypotheses are stated in propositional form and subjected to empirical test to verify them; possible confounding conditions must be carefully controlled (manipulated) to prevent outcomes from being properly influenced” (Guba & Lincoln, 1994, p. 110).

3.2.2. Interpretivism

In contrast to positivism, interpretivism does not seek to prove or disprove a hypothesis. The interpretivist approach tries to understand phenomena through the meanings and values people assign to them. In this way, multiple, subject realities are detailed. Hence, there is no single truth. Different researchers can view the world differently and their values and actions mold the research process. This results in multiple interpretations. Data collected via this paradigm is generally qualitative (Easterbrook et al., 2008, p. 291; Oates, 2007, pp. 292-293).

Guba *et al*. (1994, p. 110) describe the ontology of interpretivism as relativist. This is because realities are interpreted from social experience and intangible mental constructions from individuals or groups that hold the constructions. Constructions from such individuals or groups may be more or less informed than those formed by other individuals or groups. Guba *et al*. (p. 111) describe the epistemology of interpretivism as transactional and subjectivist. That is, the investigator and the object of investigation are assumed to be interactively linked. Guba *et al*. (p. 111) describe the methodology of interpretivism as hermeneutical and dialectical and say that "… constructions can be elicited and refined only through interaction between and among investigator and respondents" .

Williamson (2002) explains that what differentiates interpretivism from positivism is that knowledge can be acquired differently because the natural world is viewed as separate to the social world. The researcher becomes part of the study and loss of the benefit of objectivity obtained from empirical observation may result.

3.2.3. Critical Research

Critical research is similar to interpretivism from the perspective that there are multiple views of reality, but differs by saying that social reality possesses objective properties that interpretivists discount. "Critical researchers seek to identify and challenge the conditions of domination, and the restrictions and unfairness of the status quo and taken-for-granted assumptions" (Oates, 2007, p. 297).

Guba *et al*. (1994) describe the ontology of critical research as historical realism and describe the epistemology of critical research as transactional and subjectivist.

Similar to the description of the epistemology of interpretivism by Guba *et al*., the investigator and the object under investigation are assumed to be interactively linked and the values of the investigator influence the inquiry. The same researchers describe the methodology of critical research as dialogic and dialectical. A dialog is required between the investigator and the subjects of the inquiry and Guba *et al*. (p. 110) say "… dialogue must be dialectical in nature to transform ignorance and misapprehensions (accepting historically mediated structures as immutable) into more informed consciousness ...” .

3.2.4. Research Paradigm Selected for this Research

This research does not consider the social meaning of the phenomena under investigation. This discounts the use of the other identified research paradigms other than the positivist paradigm. The approach selected to address the research questions of this thesis is therefore positivist.

An empirical approach is appropriate because the result should be the same, no matter how it is measured. The use of various tools to perform measurements should produce the same results. This research is reductionist. It is studying the plethora of anti-forensic techniques malware can incorporate by measuring the effectiveness of these techniques on an individual basis together with the effectiveness of being able to detect the use of the techniques and how effectively the use of the techniques can be mitigated. The number and type of techniques employed within any particular collected malware specimen under investigation must be finite.

3.3. EMPIRICAL RESEARCH

“Empirical research seeks to explore, describe, predict, and explain natural, social, or cognitive phenomena by using evidence based on observation or experience” (Sjoberg, Dyba, & Jorgensen, 2007, p. 361). Empirical research involves the collection and interpretation of evidence through methods such as surveys, interviews, experimentation and observation.

Easterbrook *et al.* (2008, p. 290) say that once the research questions have been developed, thought has to be given to the determination of what will be accepted as the empirical truth. If ontology is considered as the nature of the world with respect to knowledge, epistemology is understood as the process in which that knowledge is obtained. This thesis undertakes an empirical approach to obtain knowledge relevant to answering the research questions.

The steps listed by Perry *et al*. (2000, p. 348) to conduct an empirical study are :

- Formulation of an hypothesis or question to test
- observing a situation,
- abstracting observations into data,
- analyzing the data, and
- drawing of conclusions with respect to the tested hypothesis.

There are various types of empirical research in which data can be produced. Easterbrook, Singer, Storey, & Damian (2008, p. 286) explicitly list the five classes of empirical research methods that they believe are most relevant to software engineering as:

- Controlled Experiments (including Quasi-Experiments)
- Case Studies (both exploratory and confirmatory)
- Survey Research
- Ethnographies
- Action Research

A controlled experiment manipulates one or more independent variables to measure the effect on one or more dependent variables to assist the researcher to determine how the variables are related and to identify causality. A hypothesis is used to guide the steps of the experimental design including which variables to include in the study and how they will be measured. This is essentially reductionist and positivist in nature. Complexity is reduced by allowing only a few variables of interest to vary in a controlled manner, whilst holding all other variables constant (Easterbrook et al., 2008, pp. 294-296).

A case study investigates a phenomenon within a context and can reveal causality. Case studies are used where the reductionism of a controlled experiment is inappropriate. This could include when effects may take a long time to appear or where the context plays a role in the phenomena under investigation (Easterbrook et al., 2008, pp. 296-298). To address the research questions of this thesis, a case study could include observing malware analysts in the field and noting how the analysts detect and mitigate anti-forensic techniques over a period of time.

Survey research can be conducted via questionnaires, structured interviews or data logging to identify characteristics of a representative sample from a well defined population. A clear research question is a precondition, the sampling technique must be sound and the survey questions must be designed to yield useful and valid data (Easterbrook et al., 2008, pp. 298-299). To address the research questions of this thesis, a survey could create a questionnaire tailored for malware analysts to determine if they believe the use of anti-forensic techniques are being increasingly used by the malware they are analyzing.

“Ethnography is a form of research focusing on the sociology of meaning through field observation” (Easterbrook et al., 2008, p. 300). To address the research questions of this thesis, ethnography could be used to observe malware analysts create practices and use strategies to detect and mitigate the use of anti-forensic techniques over a period of time.

Action research focuses on solving real world problems. “While most empirical research methods attempt to observe the world as it currently exists, action researchers aim to intervene in the studied situations for the explicit purpose of improving the situation” (Easterbrook et al., 2008, p. 301). The research questions of this research could be addressed by working in a malware research laboratory and interacting with malware analysts.

Selection of the most appropriate research method requires consideration of ontology, epistemology, methodology, resources and the abilities of the researcher with respect to the phenomena under investigation. Empirically based questions can be asked to facilitate comprehension of the ontology of the phenomenon. One of the first steps Easterbrook *et al*. (p. 287) recommends in selecting the research strategy is to clarify the research question. This begins by asking exploratory questions to aid in understanding the phenomena. Such questions assist in the determination of measurable and valid evidence. Table 3-1 re-represents the exploratory questions and the form of the question discussed by Easterbrook *et al*. (p.288) in the form of a table.

Table 3-1 Examples of exploratory research questions

Question	Form of Question
Existence questions	"Does X exist?"
Description and classification questions	"What is X like?" "What are its properties?" "How can it be categorized?" "How can we measure it?" "What is its purpose?" "What are its components?" "How do the components relate to each other?" "What are all the types of X?"
Descriptive-Comparative questions	"How does X differ from Y?"

The research questions in this thesis are fundamentally exploratory in nature and can be answered in a literature review and through empirical methods. Answering these questions assists in progressing to the next stage of questioning where Easterbrook *et al*. (p. 288) says " ... base-rate questions about the normal patterns of occurrence of the phenomena" need to be asked. Base-rate questions help to determine if a particular situation is normal or abnormal. Table 3-2 re-represents the base-rate questions and the form of the question discussed by Easterbrook *et al*. (p. 288) in the form of a table. These questions are appropriate for formulating the research questions, particularly from the perspective of gaining knowledge about how the anti-analysis techniques work and how effective they are.

Table 3-2 Examples of base-rate research questions

Question	Form of Question
Frequency and distribution questions	“How often does X occur?” “What is the average amount of X?”
Descriptive-Process questions	“How does X normally work?” “What is the process by which X happens?” “In what sequence do the events of X occur?” “What are the steps X goes through as it evolves?” “How does X achieve its purpose?”

Relationship questions seek to find out how phenomena are related to each other. Table 3-3 re-represents relationship questions in the form of table discussed by Easterbrook *et al.* (p. 288). Although relevant to future research, relationship questions are considered to be out of the scope for the line of investigation nominated in this thesis.

Table 3-3 Examples of relationship research questions

Question	Form of Question
Relationship questions	“Are X and Y related?” “Do occurrences of X correlate with occurrences of Y?”

Causality questions attempt to identify the relationship between cause and effect. Answering such questions is assisted by having answered the relationship questions presented in Table 3-3. Easterbrook *et al.* (2008, p. 289) points out that it is very important to be able to differentiate correlation and causality. This is because it is harder to demonstrate causality than it is to show correlation. If high values of variable X correlate with high values of variable Y, it could be because X causes Y, or because Y causes X. However, it could also be that some other, common variable is the cause and that neither is the cause of the other. It could also be the case that they co-evolve in complex ways and that no clear cause and effect can be identified (Easterbrook et al., 2008, p. 289).

Table 3-4 re-represents the causality questions discussed by Easterbrook *et al*. (p. 289). Causality questions are considered to be out of scope for this thesis, but remain relevant for future research that extends the line of enquiry developed in this thesis.

Table 3-4 Examples of causality research questions

Question	Form of Question
Causality questions	"Does X cause Y?" "Does X prevent Y?" "What causes Y?" "What are all the factors that cause Y?" "What effect does X have on Y?"
Causality-Comparative questions	"Does X cause more Y than does Z?" "Is X better at preventing Y than is Z?"
Causality-Comparative interaction questions	"Does X or Z cause more Y under one condition but not others?"

3.3.1. Selected Empirical Research Method

All of the empirical research methods listed in the discussion above would be suitable for addressing the research questions of this thesis. However, action research, ethnography, survey and case study would require access to malware researchers desirably working in AV software laboratories for an extended period of time, and preferably, in situ. Such access is not possible for the author at this time. The research questions of this thesis are essentially exploratory in nature. The empirical research method selected for this research is via controlled experiment.

3.4. EXPERIMENTAL STRATEGIES

Various experimental strategies are available. "In academic research, an experiment is a strategy that investigates cause and effect relationships, seeking to prove or disprove a causal link between a factor and an observed outcome" (Oates, 2007, p. 127). The strategy starts with a hypothesis which can then be tested empirically with an experiment designed to prove or disprove the hypothesis. The design of the experiment takes care to remove all factors from the study that could affect the result, apart from the one factor that is considered to cause the outcome of interest. Easterbrook *et al*. (p. 133) says true experiment concentrates on the "… manipulation of the independent variable, pre- and post-test measurement of the dependent variable(s), and control or removal of all other variables".

3.4.1. True Experiment

The experiment needs to consider the variables that can be controlled and those that can be measured. The variables can be classified as either dependent or independent. The dependent variable (effect) changes as a result of a change in the independent variable (cause). Experiments typically manipulate the independent variable and observe the effect on the dependent variable. The idea is to determine the independent variable that causes the change in the dependent variable. The experimental method is essentially positivist and reductionist. "They reduce complexity by allowing only a few variables of interest to vary in a controlled manner, while controlling all other variables" (Easterbrook et al., 2008, p. 295). The aim is to show that only one factor causes the observable change. Ways of controlling variables to assist the determination of the factor are listed by Oates (2007, p. 130) to include:

- Eliminate the factor from the experiment.
- Hold the factor constant if it is not possible to eliminate the factor.
- Use random selection of subjects
- Use control groups
- Make the researchers and subjects blind

Oates (p. 131) says an experiment has good internal validity if the measurements obtained are the result of the experimenter's handling of the independent variable and not due to other factors. Threats listed by Oates (pp. 131-132) to internal validity include:

- Differences between the experimental and control group
- History

- Maturation
- Instrumentation
- Experimental mortality
- Reactivity and experimenter effects

Oates (p. 132) says an experiment has good external validity if the "results are not unique to a particular set of circumstances but are generalizable. That is, the same results can be predicted for subsequent occasions and in other situations". Threats listed by Oates (p. 133) to external validity include:

- Over reliance on special types of participants
- Too few participants
- Non-representative participants
- Non-representative test cases

3.4.2. Quasi Experiment

Quasi-experiments try to remain within the spirit of the true experiment, "but concentrate on observing events in real-life settings where there is a 'naturally occurring' experiment" (Oates, 2007, p. 134). This is because the true experiment endeavors to have nearly complete control over the independent and dependent variable and can exhibit good internal and external validity. Pickard (2007, p. 108) points out that "internal validity is always seen as the greatest threat to quasi-experimental research design; lack of control over intervening variables means it is almost impossible to eliminate rival explanations of any relationship between variables".

In the field, control over variables is harder, and the manipulation of an independent variable is more difficult as well. Therefore, determining cause and effect is not as conclusive as that obtainable from conducting a true experiment (Oates, 2007, p. 134). The quasi-experiment "has some of the components of experimental research, but not all" (Pickard, 2007, p. 107).

Oates (p. 108) explains that there are two types of quasi-experimental research design. The non-equivalent group design and the time series design. The non-equivalent group design is similar to the true experiment, except that the selection of participants is non-random and the study is conducted in the field and not in the laboratory. The time series design is similar to the design of the non-equivalent group, except the observations are made in time intervals. This gives more observational data that can provide detail on progressive change.

3.5. CHOICE OF RESEARCH METHOD

The selected research method to address the research questions is positivist, empirical and quasi-experimental. The independent variable is the individual anti forensic technique under investigation and the dependent variable is the binary result of either detection or non-detection.

3.6. CONCEPTUAL FRAMEWORK

3.6.1. Validation of Techniques

This section discusses the general processes used to address the three research questions of this thesis. The first part of the research design is designed to validate the techniques as described and uncovered in the Literature Review chapter of this thesis. This includes determination of the ability to detect and mitigate these techniques via small quasi experiments. The results from this process are presented in the Validation of Techniques chapter of this thesis.

3.6.2. Collection of Network Based Malware

The *Nepenthes* (Nepenthes, 2006) project is a malware collection tool that works by emulating known vulnerabilities and which then downloads the payload of the malware that attempts to exploit these vulnerabilities. Dr Craig Valli of Edith Cowan University (ECU) has been participating in the Nepenthes project and has been collecting malware using a network of distributed sensors deployed within the geographical locale of Perth, Western Australia. Figure 3-2 is a process diagram depicting how malware is collected and processed by *Nepenthes* and has been adapted from the paper by Valli and Wooten (2007) which outlines how the honeynet was deployed and used to collect malware for analysis purposes.

The process diagram shows that multiple avenues of processing are conducted on the collected malware before results are stored in a SQL database and made available via a web interface. The highlighted process box designates the source of data for the research that was conducted for this thesis using malware collected by the ECU *Nepenthes* malware collection system. By the very nature of the way this malware has been collected via a network interface, the malware is classified as network based. This networked based malware was used a source of data to examine particular types of techniques malware uses to hinder analysis, namely, packers and protectors which is one of the first techniques malware analysts encounter.

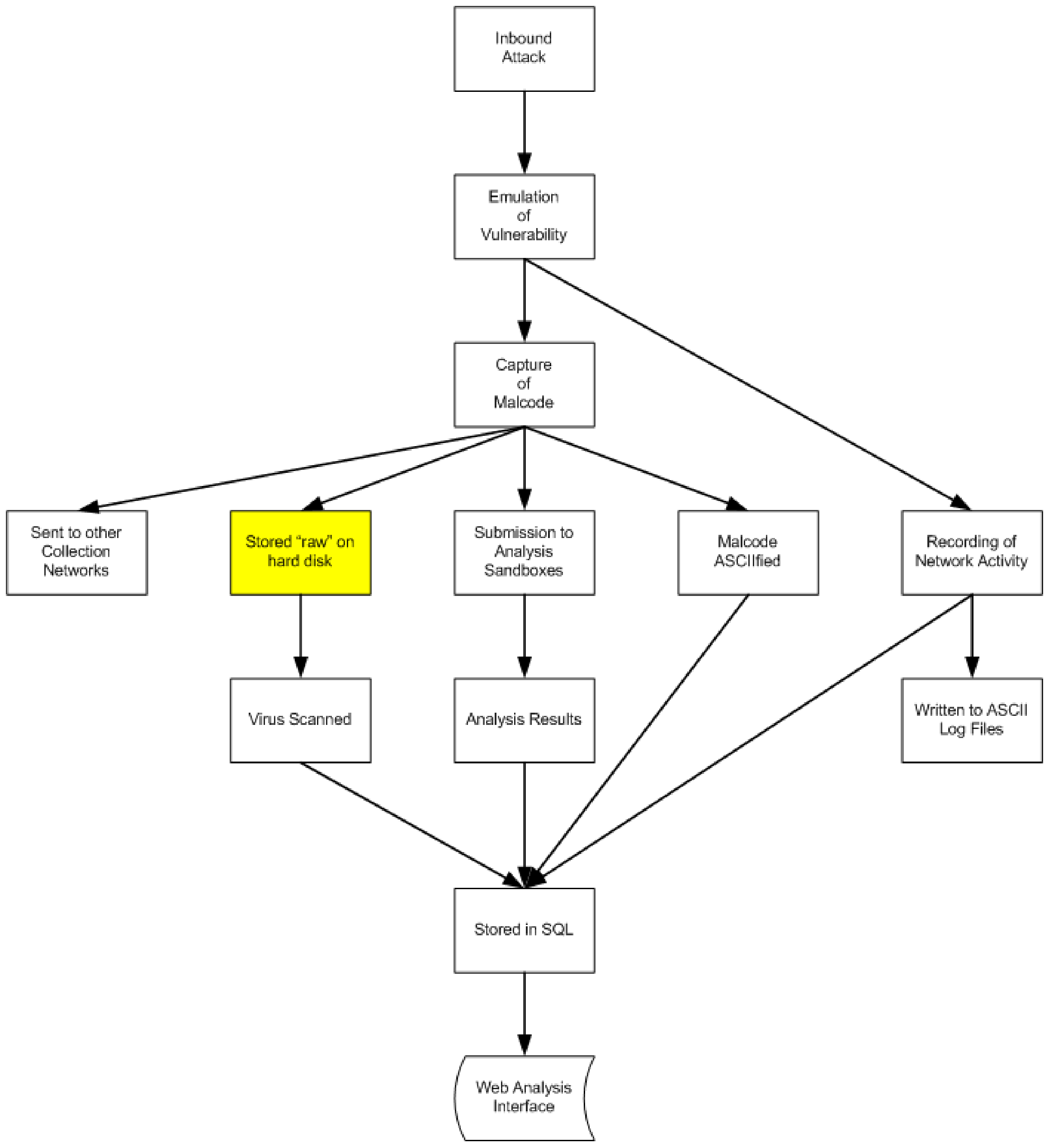

Inbound Attack
Emulation of Vulnerability
Capture of Malcode
Sent to other Collection Networks
Stored "raw" on hard disk
Submission to Analysis Sandboxes
Malcode ASCIIfied
Recording of Network Activity
Virus Scanned
Analysis Results
Written to ASCII Log Files
Stored in SQL
Web Analysis Interface

Figure 3-2 Model of the nepenthes malware collection system depicting the source (highlighted) of malware collected for this research.

3.6.3. Analysis of Collected Malware Packers

The second part of the research design is designed to analyze the use of Packers and Protectors in Microsoft Windows platform, network based malware, collected by the ECU *Nepenthes* sensors. This is also used to support the examination of the research questions, primarily with respect to the ability to detect the use of packers and protectors which is used by malware to hinder analysis. The results of this process are presented in the Analysis of Collected Malware chapter of this thesis.

3.6.4. Risk Mitigation

All steps of the process were conducted on a Linux machine which will not natively run the malware. Downloading the malware from Nepenthes and uploading the malware to the online analysis engines necessitated a connection to the internet. All other analysis work was conducted on a standalone Linux machine without an internet connection to ensure that the malware did not inadvertently interact with the internet. VMWare Virtual Machines were used to run the malware for analysis purposes under Microsoft Windows XP. The advantage of using Virtual Machines was that the state of the Virtual Machine could be restored quickly and easily at any point. Data was transferred between the Virtual Machines and the Linux host using a USB memory device.

3.7. RESEARCH DESIGN

3.7.1. Validate Individual Techniques

This process addresses the exploratory questions outlined in Table 3-1 above. The objective of this process is to validate the requirement that each individual technique prevents code from being analyzed. It also investigates the effectiveness of detection and mitigation methods that can be used against the techniques under investigation.

Inputs – Literature review, research questions, individual techniques.
Outputs – Success or failure result for Technique, Detection and Mitigation.

The steps used were:

1. Write standalone executable programs which employ the individual analysis avoidance technique as identified in the Literature Review section of this thesis.

2. Validate that the technique works by testing the general requirement for each technique, that is, “The use of the technique detects that the program is running in a debugger”.
3. Write a script that will detect the use of each technique.
4. Validate that the detection script correctly identifies each technique.
5. Write a script that will mitigate each technique.
6. Validate that the mitigation script defeats the technique.
7. Analyse results.

3.7.2. Analysis of Collected Malware

The objective of this process is to collect empirical data from the malware collected from the ECU *Nepenthes* honeypot from a variety of analysis tools. This process seeks to assess the effectiveness of Packer Detection tools and methods.

Inputs – Malware specimens from ECU *Nepenthes* sensors.

Outputs – Results from various Packer detection tools and methods.

The steps used to analyze the malware from an empirical perspective were:

1. Download the malware from the ECU *Nepenthes* sensor.
2. Create a directory with the same name as the hash of the collected malware specimen on the analysis machine.
3. Enter the hash into the “MD5 Sum” column of the “Malware Analysis” spreadsheet for each sheet that was used to record the result of each specific type of analysis method that was used.
4. Record the date the malware was collected by *Nepenthes* into the “Nepenthes” sheet.
5. Submit the specimen to *Virus Total* for analysis. Store the html page result in the directory. Virus Total is a site where malware can be submitted and the malware is tested by in excess of 30 AV Engines. Extract information from result and store in “Malware Analysis” spreadsheet in the “Virus Total” sheet. Extract data and store in the appropriate column in the sheet. Count the number of successful detections and store in “Detections” column. Count the

number of engines and store in the “Number of Engines” column. Calculate the detection result and store in column “Detection Result”.

6. Submit the specimen to *Anubis* which is an online dynamic analysis engine. Store the resultant web page into a text file (Anubis) in the directory. Record results into the sheet named “Anubis” in the appropriate columns.
7. Validate the collected malware as malicious or not.
8. Load the unpacked version of the malware into Mandiant’s Red Curtain analysis tool. Record entropy and *PEiD* results directly into the “Red Curtain” sheet in “Entropy” and “PEiD” column of the sheet respectively.
9. Determine effectiveness of Packer detection on validated malware.

CHAPTER 4 VALIDATION OF ANTI-ANALYSIS TECHNIQUES RESULTS

4.1. OVERVIEW

The literature review discussed two fundamental types of analyses appropriate to analyse malware as static and dynamic analysis. Malware tends to be heavily obfuscated to avoid signature based AV software also to defeat static analysis. Analysts generally run the malware under investigation inside a debugger so that instructions are potentially de-obfuscated and revealed at run time. After this, further analysis can commence, however, malware may contain hundreds of thousands of instructions and stepping through every instruction manually can understandably become untenable. This is because the time the analyst can allocated to the analysis is a limited resource. Debuggers have associated scripting languages to perform fundamental analysis tasks in an automated manner to avoid stepping manually through the code.

The literature review revealed that malware can use run time packers that are a stub program embedded in the malware that unpack the original code at run time into memory. Once the malware has been unpacked, the original instructions are executed. The point at which the original code is reached, after the unpacking process is completed, is referred to as the Original Entry Point (OEP). Generally, it is at this point where the program can be dumped from memory and analyzed to determine its functionality, including access to the registry, files, network communications, vectors of attack to other systems and other very useful information to the analyst.

In order to hinder dynamic analysis at such a level, the search of the literature exposed a plethora of techniques malware incorporates into its code to hide functionality. Malware can determine if it is running inside a debugger and then take control of the flow of execution so that it can use deception to hide its true intent and not reveal which files it was going to modify, how it was going to communicate over the network and other malicious activities that could identify it as malicious. This information is also required for disinfection purposes. If a known specimen of malware is detected, the intent of the quarantine process is to remove the files that are known to be associated with the specimen.

The literature associated with anti-analysis techniques very sparsely covers routines to detect the use of these techniques. Detection of the use of anti-analysis techniques was identified in the literature as potentially a very good indicator that the software under investigation is possibly of a malicious nature. Equally, the literature review revealed that mitigation techniques available in popular plugins for dealing with anti forensic techniques such as *OllyAdvanced* and *IDA Stealth* for *OllyDbg* and *IDA Pro* respectively, do not come close to providing coverage for the number of anti-analysis techniques. This provides an opportunity to investigate the methods that can be employed to detect and mitigate the use of anti-analysis techniques.

The purpose of this chapter is three fold. The first part validates a selection of the anti-analysis techniques presented in the literature review. Once the technique has been validated as successful, the implementation of the technique can be used for the next two parts. The second part is used to determine if the use of the same technique can be detected. The third part determines if the use of the same technique can be mitigated. The intention is to produce Objective Quality Evidence (OQE) to directly support answers to the three research questions of this thesis. The OQE is produced by a series of small quasi experiments where strict control over the flow of execution of the programs is maintained and external influences are minimized.

4.2. METHODOLOGY

The fundamental methodology for performing the quasi experiments is as follows:

For each anti-analysis technique under investigation:

1. Implement the technique in as simple a program as possible.
2. Observe if the anti-analysis technique is successful or not.

3. Implement a detection script or employ a detection technique to try to detect the presence of the technique.
4. Observe if the detection technique is successful or not.
5. Implement a mitigation script or technique to try and mitigate the use of the anti-analysis technique.
6. Observe if the mitigation technique is successful or not.

Steps one and two are used to produce OQE to address research question one. That is, “What techniques can malware use to avoid being analyzed?”. Steps three and four are used to produce OQE to address research question two. That is, “How can the use of these techniques be detected?”. Steps five and six are used to produce OQE to address research question three. That is, “How can the use of these techniques be mitigated?” The function of each of these steps is outlined in the following sub sections.

4.2.1. Implement the technique in as simple a program as possible

The literature review presented a wide variety of techniques malware can incorporate to hinder analysis. Code to implement the anti-analysis techniques discussed in the referenced papers exists only as code snippets. That is, as non-functioning and non-complete programs. To progress the examination of the anti-analysis technique and to determine its effectiveness, the code had to be implemented in small standalone programs. The selection of the language to develop the programs in was assembly language. This is because this is the lowest level a programmer can write code in and this is the same language that an analyst would work with when analyzing a malicious program. It has the added benefit of ensuring that the most strict control was obtained over the functioning of the code. That is, it allows control of external variables that could influence the behaviour of the program.

4.2.2. Observe if the anti-analysis technique is successful or not

Once the anti-analysis technique has been implemented, the program is run to determine if it effectively detects the presence of a debugger and alters its path of execution. This can be observed at the debugger level, by stepping through the program and observing each and every instruction as it is executed at the assembly language level. It is intended that the result of each of these tests will either show cause and effect, or not. Figure 4-1 depicts the execution logic of the program and shows the only two possible observable results in a simple flow chart. Either the technique detects the presence of a debugger or it does not.

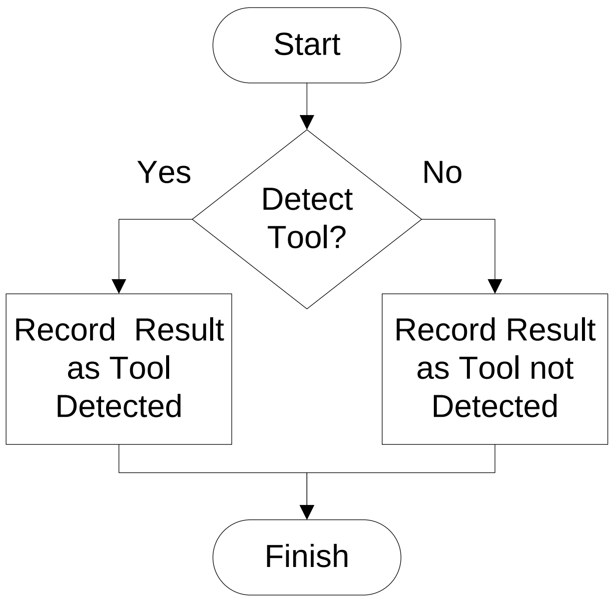

Start
Yes
Detect Tool?
No
Record Result as Tool Detected
Record Result as Tool not Detected
Finish

Figure 4-1 Simple flowchart to record if technique was successful or not in detecting the presence of a tool.

4.2.3. Implement a detection script or employ a detection technique to try to detect the presence of the technique.

The purpose of this step is to implement a debugging script or to use an analysis technique to detect the use of the anti-analysis technique in the developed program. A small variety of scripting languages was used to achieve this, using the two most popular debuggers used in Malware Digital Forensics, *IDA Pro* (Commercial) and *OllyDbg (Non Commercial) (Zeltser, 2007)*. Scripts are written such that they will either detect the technique or not and no unnecessary programming overhead is included. Where scripting languages were not used, features of the debuggers were used instead to detect the use of the technique. Selection of techniques to implement was essentially determined by the techniques implemented in popular anti forensic plugins such as the IDA Stealth plugin for validation purposes. This gave an addition validating mechanism to determine if the technique was successful or not.

4.2.4. Observe if the detection technique is successful or not.

Figure 4-2 depicts the logic of the observable result from conducting the test. It is intended that the result of each of these tests will show cause and effect. The results of each test are recorded as observations, the detection technique either worked or it did not.

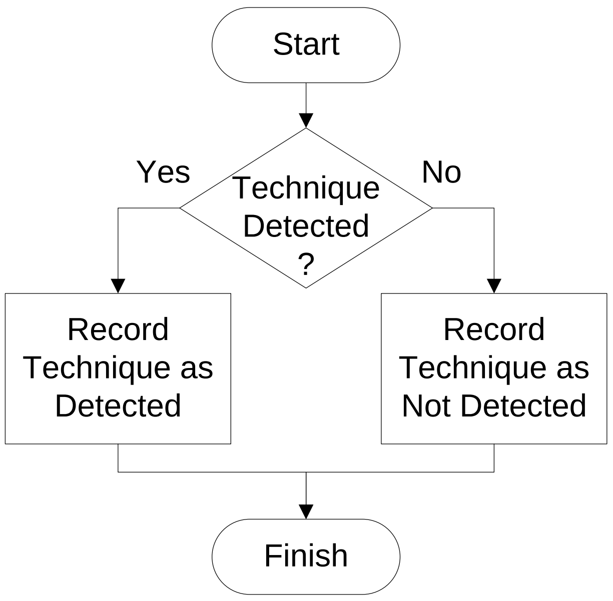

Start
Yes
Technique Detected ?
No
Record Technique as Detected
Record Technique as Not Detected
Finish

Figure 4-2 Simple flow chart depicting logic of recording the result of script or technique to detect implementation of anti-analysis technique.

4.2.5. Implement a mitigation script or technique to try and mitigate the use of the anti-analysis technique.

Scripts were written or techniques were employed to mitigate the use of the anti-analysis technique. Scripts are written such that they will either mitigate the technique or not and no unnecessary programming overhead is included. Where scripting languages were not used, features of the debuggers were used instead to mitigate the use of the technique.

4.2.6. Observe if the mitigation technique is successful or not.

Figure 4-3 depicts the logic of the test of the mitigation script or technique. Either the mitigation technique was successful or not.

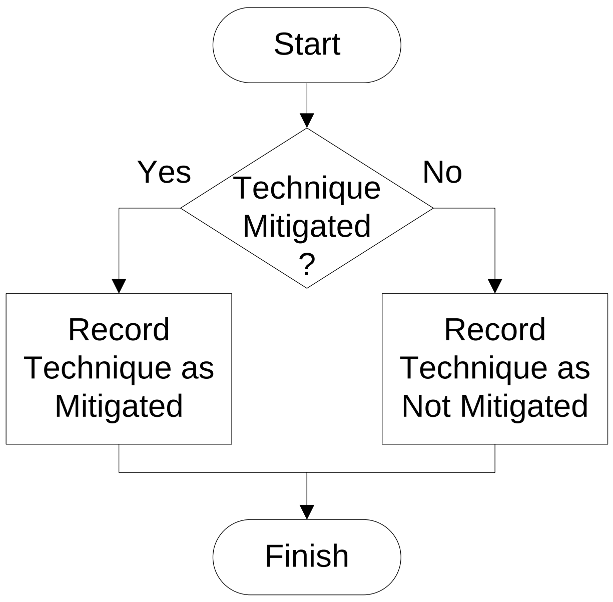

Start
Yes
Technique Mitigated ?
No
Record Technique as Mitigated
Record Technique as Not Mitigated
Finish

Figure 4-3 Simple flow chart depicting the logic of recording the result of the mitigation script or technique.

4.3. KERNEL32 ISDEBUGGERPRESENT() QUASI EXPERIMENT

4.3.1. Implementation of anti-analysis technique

Figure 4-4 demonstrates a call to the `kernel32` DLL function `IsDebuggerPresent()`. `IsDebugger` present will return `1` if the process is being debugged, `0` if not being debugged, and an appropriate message will be displayed. The `ADDR` keyword specifies that pointers to the strings are being passed to the `MessageBox` function.

```
.686
.MODEL flat, stdcall
OPTION CASEMAP:NONE   ;Case sensitive

include windows.inc
include kernel32.inc
includeLib c:\masm32\lib\kernel32.lib
include user32.inc
includeLib c:\masm32\lib\user32.lib

.DATA
     text1 db 'Debugger Not Detected', 0
     caption db 'IsDebuggerPresent',0
     text2 db 'Debugger Detected', 0
.CODE
Start:
     INVOKE IsDebuggerPresent
     TEST EAX,EAX
     JNZ DebuggerDetected
     INVOKE MessageBox, 0, ADDR text1, ADDR caption, MB_OK
     JMP Finish
DebuggerDetected:
     INVOKE MessageBox, 0, ADDR text2, ADDR caption, MB_OK
Finish:
     INVOKE ExitProcess, 0
End Start
```

Figure 4-4 Listing of implementation of kernel32 IsDebuggerPresent technique.

4.3.2. Effectiveness of anti-analysis technique observation

The debugger was detected when the program was run in *OllyDbg* and *IDA Pro*.

4.3.3. Implementation of detection of analysis avoidance technique

The use of functions can be easily detected from a static analysis point of view in *IDA Pro*. The *IDA Python* script in Figure 4-5 shows how the name of a function can be detected. It should be noted that this is a very simple example and that malware can obfuscate function names so that detection is not so easy. The function prints to the screen, but could just as easily write to a file or a port. It should be noted that this script works with the static disassembly. A script can also be written that will work inside the debugger as it runs. This approach facilitates dynamic analysis and even allows decisions to be made about the control flow of the program as it runs. The reality is that a function found from a disassembly may never be actually called. This can be determined by checking to see what other functions (cross references) call the function of interest. The compromise is that with a static analysis, the analyst is not actually running malicious code. However, with a dynamic analysis (running in the debugger), the malicious code is actually interacting with the system.

```
# detectFunction(functionToFind)
# detect the presence of a particular function
# input  : functionToFind = function to find as string
# output : True if function found, False otherwise
def detectFunction(functionToFind):
    found = False
    # get the segments starting address
    ea = ScreenEA()
    # loop through all the functions in the segment
    for function_ea in Functions(SegStart(ea), SegEnd(ea)):
        if GetFunctionName(function_ea) == functionToFind:
            found = True
            print hex(function_ea), GetFunctionName(function_ea)
    return found

def main():
     detectFunction("IsDebuggerPresent")

if __name__ == "__main__":
    main()
```

Figure 4-5 IDA Python function detection script used for static analysis.

4.3.4. *Effectiveness of detection of technique observation*

The detection script effectively detected the use of the technique.

4.3.5. *Implementation of mitigation technique*

The mitigation technique employed was the use of the selection of the *OllyAdvanced* option to detect IsDebuggerPresent in *OllyDbg* and to use the IsDebuggerPresent flag in *IDA Stealth*.

4.3.6. *Effectiveness of mitigation technique observation*

The *OllyAdvanced* option and the *IDA Stealth* option were effective in mitigating the technique in the implemented program in Figure 4-4.

4.4. PEB ISDEBUGGED() QUASI EXPERIMENT

4.4.1. Implementation of anti-analysis technique

In such a simple example as shown in the listing in Figure 4-4, the `IsDebuggerPresent()` function call shows up in the import table and can be detected. Since the API function call itself is simply reading the second byte of the Process Environment Block (PEB) at offset 2, a stealthy version can attempt to do this itself directly instead of calling the `IsDebuggerPresent` API function as shown in the listing of Figure 4-6. Offset `+30` from the Thread Environment Block (TEB) data structure points to the PEB of the current process. Because a `BYTE` is being transferred to `EAX`, it must be extended with zeros (`MOVZX`) to fill the register.

```
.686
.MODEL flat, stdcall
OPTION CASEMAP:NONE   ;Case sensitive

include windows.inc
include kernel32.inc
includeLib c:\masm32\lib\kernel32.lib
include user32.inc
includeLib c:\masm32\lib\user32.lib

.DATA
     text1 db 'Debugger Not Detected', 0
     caption db 'IsDebugged',0
     text2 db 'Debugger Detected', 0
.CODE
Start:
     ASSUME FS:NOTHING
     MOV EAX, DWORD PTR FS:[30h]
     MOVZX EAX, BYTE PTR [EAX+2]  ;mov with zero extend
     TEST EAX,EAX
     JNZ DebuggerDetected
     INVOKE MessageBox, 0, addr text1, addr caption, MB_OK
     JMP Finish
DebuggerDetected:
     INVOKE MessageBox, 0, addr text2, addr caption, MB_OK
Finish:
     INVOKE ExitProcess, 0
End Start
```

Figure 4-6 Listing of implementation of PEB!IsDebugged technique

4.4.2. Effectiveness of anti-analysis technique observation

The use of the technique effectively detected the presence of *OllyDbg* and *IDA Pro.*

4.4.3. Implementation of detection of analysis avoidance technique

The IsDebuggerPresent flag is an option in *IDA Stealth* that can be used to detect the use of this technique. An alternative to using *IDA Stealth* is to patch the IsDebugged field of the Process Environment Block (PEB) using the *IDC script* in Figure 4-7, partially extracted from an example from Eagle (2008b). Although Eagle's technique is effective at mitigation, some additional modification is required to check if the malware is using this detection method.

```
#include <idc.idc>

static main() {
   auto globalFlags, func, end;
   // run to the entry point
   RunTo(BeginEA());
   // launch the debugger, but suspend
   GetDebuggerEvent(WFNE_SUSP, -1);
   //ebx points to peb on entry.  This is only true at BeginEA,
not main
   PatchByte(EBX + 2, 0);           //PEB!IsDebugged = 0;
   // resume the debugger
  GetDebuggerEvent(WFNE_CONT , -1);
}
```

Figure 4-7 IDC script PatchIsDebuggerPresent.idc to patch IsDebuggerPresent flag in PEB.

Another way to detect that this technique is being used, is to check when the PEB is being accessed. One way to do this is to check the second operand for each instruction to see if it is accessing the PEB at FS:[30h] as shown in the listing in Figure 4-8.

```
// simple example to find a pattern dynamically
#include <idc.idc>

static main() {
  auto code;
  EnableTracing(TRACE_STEP, 1);
  findPattern(GetEventEa(), "fs:30h");
  for (code = GetDebuggerEvent(WFNE_ANY | WFNE_CONT, -1); code >
0;
            code = GetDebuggerEvent(WFNE_ANY | WFNE_CONT, -1))
{
     findPattern(GetEventEa(), "fs:30h");
  }
  EnableTracing(TRACE_STEP, 0);
}

// if pattern found in second operand, print a short message
static findPattern(addr, pattern)
{
  auto oper1, oper2, mnem;
  mnem = GetMnem(addr);
  oper1 = GetOpnd(addr, 0);
  oper2 = GetOpnd(addr, 1);
  if (strstr(oper2, pattern) >= 0) {
      Message("Found %s\n", pattern);
     Message("%x %s %s, %s\n", addr, mnem, oper1, oper2);
  }
  return 0;
}
```

Figure 4-8 IDC script to find a pattern at run time.

4.4.4. *Effectiveness of detection of technique observation*

Both the manual detection technique discussed above and the detection scripts were very effective at detecting the use of the anti-analysis technique. If the *OllyAdvanced* option to detect IsDebuggerPresent is selected when the code in Figure 4-6 is run, *OllyDbg* will be detected because the call to the function IsDebuggerPresent is never called. This emphasizes the importance of understanding the limitations of the functionality of tools and the likelihood of workarounds to have been discovered and implemented to mitigate detection methods used by analysts.

4.4.5. *Implementation of mitigation technique*

The PEB can be viewed in *OllyDbg* by pressing Ctrl+G (Goto Expression) in the data window and entering FS:[30]. Highlight the offset at 0x02 (remembering to start at 0), press the space bar to pull up the editor, and change the 0x01 to 0x00. This emphasizes a significant difference between *IDA Pro* and *OllyDbg*. It is much easier to patch code with *OllyDbg* than with *IDA Pro* and save the modified binary. *OllyDbg* is working with the actual, original binary, whereas *IDA Pro* is working with an analyzed version of the original binary that is stored in a database, but can still be patched and run.

4.4.6. *Effectiveness of mitigation technique observation*

The use of the mitigation technique was effective.

4.5. PEB NTGLOBALFLAGS() QUASI EXPERIMENT

4.5.1. *Implementation of anti-analysis technique*

The DWORD located at offset 0x68 in the PEB contains flags that define how various APIs will be used by the loaded program, and certain flags are set if the process is being run in a debugger. These flags are listed in Figure 4-9.

```
FLG_HEAP_ENABLE_TAIL_CHECK        (0x10)
FLG_HEAP_ENABLE_FREE_CHECK        (0x20)
FLG_HEAP_VALIDATE_PARAMETERS      (0x40)
```

Figure 4-9 NTGlobal Flags used to detect if program is running inside a debugger

The NtGlobalFlag will be set to 0x00 in a program that is not being debugged. If the program is being debugged, the NtGlobalFlag will be set to 0x70 which shows that the above flags are set. These flags can be set by the call to the ntdll function LdrpInitializeExecutionOptions(). The listing in Figure 4-10 demonstrates this technique.

```
.686
.MODEL flat, stdcall
OPTION CASEMAP:NONE   ;Case sensitive

include windows.inc
include kernel32.inc
includeLib c:\masm32\lib\kernel32.lib
include user32.inc
includeLib c:\masm32\lib\user32.lib

.DATA
     text1 db 'Debugger Not Detected', 0
     caption db 'NtGlobalFlags',0
     text2 db 'Debugger Detected', 0
.CODE
Start:
     ASSUME FS:NOTHING
     MOV EAX, DWORD PTR FS:[30h]
     MOVZX EAX, BYTE PTR [EAX+68h]
     CMP EAX, 70h
     TEST EAX,EAX
     JNZ DebuggerDetected
     INVOKE MessageBox, 0, addr text1, addr caption, MB_OK
     JMP Finish
DebuggerDetected:
     INVOKE MessageBox, 0, addr text2, addr caption, MB_OK
Finish:
     INVOKE ExitProcess, 0
End Start
```

Figure 4-10 Listing of implementation of PEB!NTGlobalFlags technique to detect presence of debugger.

4.5.2. Effectiveness of anti-analysis technique observation

The use of the technique effectively detected the presence of *OllyDbg* and *IDA Pro. OllyDbg* was detected, unless the *OllyAdvanced* NtGlobal flag option was enabled. Equally, *IDA Pro* was detected until the NtGlobalFlag (Patch global heap flag) option was selected.

4.5.3. Implementation of detection of analysis avoidance technique

To detect the use of this technique, the pattern searching script in Figure 4-8 can be used to notify the analyst about code access to the PEB. The pattern searching script could be modified to cater for the various permutations that are possible.

4.5.4. *Effectiveness of detection of technique observation*

The pattern matching technique in Figure 4-8 effectively detected the use of the technique using *IDA Pro*.

4.5.5. *Implementation of mitigation technique*

The listing in Figure 4-11 is partially extracted from an example by (Eagle, 2008b) and shows how the NtGlobalFlag can be successfully patched at run time using the *IDC* scripting language in *IDA Pro*.

```
#include <idc.idc>

static main() {
   auto globalFlags, func, end;
   RunTo(BeginEA());
   GetDebuggerEvent(WFNE_SUSP, -1);
   globalFlags = Dword(EBX + 0x68) & ~0x70;
   PatchDword(EBX + 0x68, globalFlags);
}
```

Figure 4-11 IDC Script to patch NtGlobalFlags at run time to avoid detection of debugger.

4.5.6. *Effectiveness of mitigation technique observation*

Use of the script in Figure 4-11effectively mitigated the use of the technique.

4.6. HEAP FLAGS QUASI EXPERIMENT

4.6.1. *Implementation of anti-analysis technique*

When the first heap of a program is created, its Flags will be set to 0x02 to designate that the heap can grow and the ForceFlags field will be set to 0x00. However, when a process is being debugged, “these flags are usually set to 0x50000062 (depending on the NTGlobalFlag) and 0x40000060 (which is Flags AND 0x6001007D)” (Yason, 2007, p.5). The following heap flags in Figure 4-12 are also set when a heap is created on a debugged process.

```
HEAP_TAIL_CHECKING_ENABLED (0X20)
HEAP_FREE_CHECKING_ENABLED (0X40)
```

Figure 4-12 Heap flags that are set when a process is being debugged. These can be used to detect the presence of a debugger.

Falliere (2007, p.3) says that checking the `ForceFlags` field in a heap header at offset `0x10` can be used to detect the presence of a debugger. This technique is implemented in the listing in Figure 4-13.

```
.686
.MODEL flat, stdcall
OPTION CASEMAP:NONE   ;Case sensitive

include windows.inc
include kernel32.inc
includeLib c:\masm32\lib\kernel32.lib
include user32.inc
includeLib c:\masm32\lib\user32.lib

.DATA
      text1 db 'Debugger Not Detected', 0
      caption db 'Heap Flags',0
      text2 db 'Debugger Detected', 0
.CODE
Start:
      ASSUME FS:NOTHING
      MOV EAX, DWORD PTR FS:[30h]
      MOV EAX,  [EAX+18h]  ;process heap
      MOV EAX, [EAX+10h] ; heap flags
      TEST EAX,EAX
      JNZ DebuggerDetected
      INVOKE MessageBox, 0, addr text1, addr caption, MB_OK
      JMP Finish
DebuggerDetected:
      INVOKE MessageBox, 0, addr text2, addr caption, MB_OK
Finish:
      INVOKE ExitProcess, 0
End Start
```

Figure 4-13 Listing of implementation of HeapFlags detection technique.

4.6.2. Effectiveness of anti-analysis technique observation

The use of the technique effectively detected the presence of *OllyDbg* and *IDA Pro.*

4.6.3. Implementation of detection of analysis avoidance technique

To detect the use of this technique, the pattern searching script from Figure 4-8 can be used to detect when the PEB is being accessed. However, it should be noted that it would be very easy to further obfuscate the operand to access the PEB.

4.6.4. Effectiveness of detection of technique observation

The use of the detection technique proved to be effective.

4.6.5. Implementation of mitigation technique

Falliere (2007, p.3) suggests two ways to mitigate the use of this technique as follows:

1. Create a non-debugged process, and attach the debugger once the process has been created. An easy solution is to create the process suspended, run until the entry-point is reached, patch it to an infinite loop, resume the process, attach the debugger, and restore the original entry-point.
2. Edit the registry key:
`HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options`
"Create a subkey (not value) names as your process name, and under this subkey, a String value `GlobalFlags` set to nothing" (Falliere, 2007, p.3).

Yason (2007, p.5) says that a solution is to patch the `PEB.NTGlobalFlag` and `PEB.HeapProcess` flag to the values as if the process is not being debugged. Yason provides an *`OllyScript`* to patch the flags that is reproduced as follows in the listing in Figure 4-14. The assembly language feel is very evident in *`OllyScript`* syntax and serves as a very interesting contrast to *`IDAPython`* and *`IDC script`*. A variety of *`OllyScripts`* can be found on most reverse engineering web sites and can be used to see how particular analysis techniques work and if desired, transform the algorithm into another scripting language such as *`IDAPython`* to work with *`IDA Pro`*.

```
var peb
var patch_addr
var process_heap

// retrieve PEB via a hardcoded TEB address (first thread:
// 0x7ffde000)
mov peb, [7ffde000+30]

//patch PEB.NtGlobalFlag
lea patch_addr, [peb+68]
mov [patch_addr], 0

//patch PEB.ProcessHeap.Flags/ForceFlags
mov process_heap, [peb+18]
lea patch_addr, [process_heap+0c]
mov [patch_addr], 2
lea patch_addr, [process_heap+10]
mov [patch_addr], 0
```

Figure 4-14 OllyScript to patch Heap Flags

4.6.6. Effectiveness of mitigation technique observation

The technique was mitigated when the `Heap Flag` option of *`IDA Stealth`* was checked. The script in Figure 4-14 effectively mitigated the technique in *`IDA Pro`*. Setting the Heap Flags option in *`OllyAdvanced (v1.26)`* did not help in mitigating this case, the debugger was still detected.

4.7. NTQUERYINFORMATIONPROCESS() QUASI EXPERIMENT

4.7.1. Implementation of anti-analysis technique

The `NtQueryInformationProcess` call is used to retrieve information about the running process. Its prototype is shown in Figure 4-15.

```
NTSTATUS WINAPI NtQueryInformationProcess(
  __in       HANDLE ProcessHandle,
  __in       PROCESSINFOCLASS ProcessInformationClass,
  __out      PVOID ProcessInformation,
  __in       ULONG ProcessInformationLength,
  __out_opt  PULONG ReturnLength
);
```

Figure 4-15 NtQueryInformationProcess call used to retrieve information about the running process

The `PROCESSINFOCLASS` enumeration can be set with a value of `7` to retrieve the port number of the debugger for the process. The process is being debugged if the return value is non zero. An example implementation of this technique by ap0x (2006) is shown in the listing in Figure 4-16.

```
.386
.model flat, stdcall
option casemap :none   ; case sensitive

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data
       DbgNotFoundTitle db "Debugger status:",0h
       DbgFoundTitle db "Debugger status:",0h
       DbgNotFoundText db "Debugger not found!",0h
       DbgFoundText db "Debugger found!",0h
       ntdll db "ntdll.dll",0h
       zwqip db "NtQueryInformationProcess",0h
.data?
       NtAddr dd ?
       MinusOne dd ?
.code

start:

; MASM32 antiOlly example
; coded by ap0x
; Reversing Labs: http://ap0x.headcoders.net
;    This    example    can    detect    Olly    by    using
NtQueryInformationProcess API.
MOV [MinusOne],0FFFFFFFFh
PUSH offset ntdll ;ntdll.dll
CALL LoadLibrary
PUSH offset zwqip ;NtQueryInformationProcess
PUSH EAX
CALL GetProcAddress
MOV [NtAddr],EAX
MOV EAX,offset MinusOne
PUSH EAX
MOV EBX,ESP
PUSH 0
PUSH 4
PUSH EBX
PUSH 7
PUSH DWORD PTR[EAX]
CALL [NtAddr]
POP EAX
TEST EAX,EAX
JNE @DebuggerDetected
PUSH 40h
PUSH offset DbgNotFoundTitle
PUSH offset DbgNotFoundText
PUSH 0
```

```
CALL MessageBox
JMP @exit
  @DebuggerDetected:
PUSH 30h
PUSH offset DbgFoundTitle
PUSH offset DbgFoundText
PUSH 0
CALL MessageBox
  @exit:
PUSH 0
CALL ExitProcess
end start
```

Figure 4-16 Implementation of NtQueryInformationProcess technique to detect the presence of a debugger (ap0x, 2006)

4.7.2. Effectiveness of anti-analysis technique observation

The use of the technique effectively detected the presence of *OllyDbg* and *IDA Pro.*

4.7.3. Implementation of detection of analysis avoidance technique

The use of particular functions (where the use of the function is not obfuscated) can be easily detected in *IDC* by the use of the function call LocByName() which takes the name of the function to search for as a parameter and returns the address of the function which serves to detect the use of the function.

4.7.4. Effectiveness of detection of technique observation

The use of the technique was effectively detected using the function call LocByName() in *IDC*.

4.7.5. Implementation of mitigation technique

The NtQueryInformationProcess is a wrapper around the ZwQueryInformationProcess system call. The debugger will be found until the *OllyAdvanced* option ZwQueryInformationProcess is enabled. The NtQueryInformationProcess signature is as follows in Figure 4-17.

```
NTSTATUS NTAPI NtQueryInformationProcess (
     HANDLE               ProcessHandle,
     PROCESSINFOCLASS     ProcessInformationClass,
     PVOID                ProcessInformation,
     ULONG                ProcessInformationLength,
     PULONG               ReturnLength
}
```

Figure 4-17 Signature of NtQueryInformationProcess

IDA Stealth has an option to mitigate this technique using the NTQueryInformationProcess option.

Once the address of the function has been found, the function can be mitigated by setting a breakpoint on the return from NtQueryInformationProcess. An algorithm presented by Eagle (2008a, p. 534) using *IDA Pro* is as follows:

- Locate the address of NtQueryInformationProcess.
- Create a function at the address.
- Find the end address of the function.
- Find the beginning of the return instruction by subtracting three from the end address and set a breakpoint at this address.
- Add a condition function on the breakpoint and set the breakpoint's attributes so that execution is prevented from stopping on the breakpoint.

The listing in Figure 4-18 is extracted from an example by (Eagle, 2008b) that implements the algorithm described above.

```
#include <idc.idc>

//handle a return from NtQueryInformationProcess
#define ProcessDebugPort 7
static bpt_NtQueryInformationProcess() {
   auto p_ret;
   if (Dword(ESP + 8) == ProcessDebugPort) {
      //test ProcessInformationClass
      p_ret = Dword(ESP + 12);
      if (p_ret) {
         PatchDword(p_ret, 0);  //fake no debugger present
      }
   }
}

static main() {
   auto globalFlags, func, end;
   RunTo(BeginEA());
   GetDebuggerEvent(WFNE_SUSP, -1);

//   func = LocByName("ntdll_NtQueryInformationProcess");
   func = LocByName("ntdll_ZwQueryInformationProcess");
   MakeFunction(func, BADADDR);
   end = GetFunctionAttr(func, FUNCATTR_END) - 3;
   AddBpt(end);
   SetBptAttr(end, BPT_BRK, 0);  //don't stop
   SetBptCnd(end, "bpt_NtQueryInformationProcess()");

}
```

Figure 4-18 Listing of NtQueryInformationProcess avoidance technique (Eagle, 2008b)

A code snippet from Yason (2007, p.7) that uses NtQueryInformationProcess is reproduced in Figure 4-19:

```
; using ntdll!NtQueryInformationProcess (ProcessDebugPort)
lea   eax,[.dwReturnLen]
push  eax        ; ReturnLength
push  4          ; ProcessInformationLength
lea   eax, [.dwDebugPort]
push  eax        ; ProcessInformation
push  ProcessDebugPort ; ProcessInformationClass (7)
push  0xffffffff ; ProcessHandle
call  [NtQueryInformationProcess]
cmp   dword [.dwDebugPort], 0
jne   .debugger_found
```

Figure 4-19 Code snippet using NtQueryInformationProcess (Yason, 2007, p.7)

An example *OllyScript* presented by Yason (2007, p.7) is reproduced in Figure 4-20. It shows how a breakpoint can be set where NtQueryInformationProcess() returns and then patches ProcessInformation to 0 when the breakpoint is hit.

```
var        bp_NtQueryInformationProcess

// set a breakpoint handler
eob        bp_handler_NtQueryInformationProcess

// set a breakpoint where NtQueryInformationProcess returns
gpa        “NtQueryInformationProcess”, “ntdll.dll”
find       $RESULT, #c21400#          //retn 14
mov        bp_NTQueryInformationProcess, $RESULT
bphws      bp_NTQueryInformationProcess, “x”
run

bp_handler_NtQueryInformationProcess:
// ProcessInformationClass == ProcessDebugPort ?
cmp        [esp+8], 7
jne        bp_handler_NtQueryInformationProcess_continue

// patch ProcessInformation to 0
mov        patch_addr, [esp+c]
mov        [patch_addr], 0

// clear breakpoint
bphwc      bp_NtQueryInformationProcess

bp_handler_NtQueryInformationProcess_continue:
run
```

Figure 4-20 OllyScript to Patch ProcessInformation (Yason, 2007, p.7)

4.7.6. Effectiveness of mitigation technique observation

The mitigation techniques were observed to be very effective.

4.8. KERNEL32 CHECKREMOTEDEBUGGERPRESENT() QUASI EXPERIMENT

4.8.1. Implementation of anti-analysis technique

This call has two parameters, a process handle, and a pointer to a BOOLEAN variable that will be set to TRUE if it is found that a debugger is attached to the process. The signature of this call is as follows in Figure 4-21.

```
BOOL CheckRemoteDebuggerPresent (
     HANDLE     hProcess,
     PBOOL      pbDebuggerPresent
)
```

Figure 4-21 Signature of CheckRemoteDebuggerPresent

The call chain for this function is via the `ntdll` function `NtQueryInformationProcess` which queries the `DebugPort` field of the `EPROCESS` kernel structure. An example listing (ap0x, 2006) is provided in Figure 4-22 that uses the `CheckRemoteDebuggerPresent` function call.

```
.386
.model flat, stdcall
option casemap :none   ; case sensitive

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data
DbgNotFoundTitle db "Debugger status:",0h
DbgFoundTitle db "Debugger status:",0h
DbgNotFoundText db "Debugger not found!",0h
DbgFoundText db "Debugger found!",0h
krnl db "kernel32.dll",0h
chkrdbg db "CheckRemoteDebuggerPresent",0h
.data?
IsItPresent dd ?
.code

start:

; MASM32 antiRing3Debugger example
; coded by ap0x
; Reversing Labs: http://ap0x.headcoders.net
; CheckRemoteDebuggerPresent is function similar to
; IsDebuggerPresent.
; This function is available only in Windows NT and it
; outputs TRUE or FALSE value if debugger is present
; in selected process.

; Load the function via GetProcAddress

PUSH offset krnl ;kernel32.dll
CALL LoadLibrary
PUSH offset chkrdbg ;CheckRemoteDebuggerPresent
PUSH EAX
CALL GetProcAddress
```

```
; IsItPresent variable will store the result
PUSH offset IsItPresent
PUSH -1
CALL EAX
MOV EAX,DWORD PTR[IsItPresent]
TEST EAX,EAX
JNE @DebuggerDetected
PUSH 40h
PUSH offset DbgNotFoundTitle
PUSH offset DbgNotFoundText
PUSH 0
CALL MessageBox
JMP @exit
  @DebuggerDetected:
PUSH 30h
PUSH offset DbgFoundTitle
PUSH offset DbgFoundText
PUSH 0
CALL MessageBox
  @exit:
PUSH 0
CALL ExitProcess
end start
```

Figure 4-22 Listing of CheckRemoteDebuggerPresent technique to find presence of remote debugger (ap0x, 2006)

4.8.2. Effectiveness of anti-analysis technique observation

The use of the technique effectively detected the presence of *OllyDbg* and *IDA Pro.*

4.8.3. Implementation of detection of analysis avoidance technique

The use of this technique can be detected by locating calls to the CheckRemoteDebuggerPresent function call as exemplified by the routine presented in Figure 4-23.

4.8.4. Effectiveness of detection of technique observation

Detection of the use of the technique proved to be effective.

4.8.5. Implementation of mitigation technique

A technique to detect and patch the use of this technique with the *Immunity Debugger* is provided by BoB (2007) in the procedure listed in Figure 4-23.

```
#-----------------------------------------------------------------
----------
# CheckRemoteDebuggerPresent ..
# Note: This Api calls ZwQueryInformationProcess Api,
# so usually no need to patch both ..

def Patch_CheckRemoteDebuggerPresent(imm):
    deb = imm.getAddress( "kernel32.CheckRemoteDebuggerPresent"
)
    # Just incase on Win2k .. ;)
    if (deb <= 0):
        imm.Log( "No CheckRemoteDebuggerPresent to patch .." )
        return

    imm.Log( "Patching CheckRemoteDebuggerPresent ..", address =
deb )
    imm.writeMemory( deb, imm.Assemble( " \
        Mov   EDI, EDI                                   \n \
        Push  EBP                                        \n \
        Mov   EBP, ESP                                   \n \
        Mov   EAX, [EBP + C]                             \n \
        Push  0                                          \n \
        Pop   [EAX]                                      \n \
        Xor   EAX, EAX                                   \n \
        Pop   EBP                                        \n \
        Ret   8                                          \
    " ) )
```

Figure 4-23 Implementation of CheckRemoteDebuggerPresent detection technique (BoB, 2007)

After detection, the procedure patches the program by assembling new instructions to replace the original instructions. The assembly language commands and assembled instructions appear as follows in Figure 4-24.

```
MOV   EDI, EDI            8B FF
PUSH  EBP                 55
MOV  EBP, ESP             8B EC
MOV  EAX, [EBP + 0Ch]     8B 45 0C
PUSH 0                    6A 00
POP  [EAX]                8F 00
XOR  EAX, EAX             33 C0
POP  EBP                  5D
RET   8                   C2 08 00
```

Figure 4-24 Resultant patched program after running CheckRemoteDebuggerPresent detection script.

Once the start address of the `CheckRemoteDebuggerPresent` function is found in the `Kernel32` DLL, memory can be over written with the new instructions. This can be done manually through a debugger, or through a script. Figure 4-25 provides an equivalent example written in *`IDC script`*. A variety of other anti anti debugging techniques in BoB's script include:

- `IsDebuggerPresent`
- `ZwQueryInformationProcess`
- `CheckRemoteDebuggerPresent`
- `PEB.IsDebugged`
- `PEB.ProcessHeap.Flag`
- `PEB.NtGlobalFlag`
- `PEB.Ldr`
- `GetTickCount`
- `ZwQuerySystemInformation`
- `FindWindowA`
- `FindWindowW`
- `FindWindowExA`
- `FindWindowExW`
- `EnumWindows`

```
#include <idc.idc>

# detect and patch CheckRemoteDebuggerPresent

static main() {
   auto addr;
   RunTo(BeginEA());
   GetDebuggerEvent(WFNE_SUSP, -1);
   addr = LocByName("kernel32_CheckRemoteDebuggerPresent");
   if (addr != BADADDR){
       Message("CheckRemoteDebuggerPresent at address %x\n", addr);
       patchCheckRemoteDebuggerPresent(addr);
   }
}

static patchCheckRemoteDebuggerPresent(addr) {
   PatchDword(addr, 0x8B55FF8B);
   PatchDword(addr + 4, 0x0C458BEC);
   PatchDword(addr + 8, 0x008F006A);
   PatchDword(addr + 12, 0xC25DC033);
   PatchWord(addr + 16, 0x0008);
}
```

Figure 4-25 CheckRemoteDebuggerPresent detection and mitigation IDC Script (Dynamic)

4.8.6. *Effectiveness of mitigation technique observation*

The implemented mitigation techniques proved to be very effective. The debugger was detected when run inside *OllyDbg*, but was mitigated when the *OllyAdvanced* ZwQuerySystemInformation option was enabled. *IDA Pro* was detected when the program was executed. The *NTQueryInformation* process (which includes CheckRemoteDebuggerPresent) checkbox must be selected in *IDA Stealth* to prevent its discovery by the anti-analysis technique.

4.9. UNHANDLED EXCEPTION FILTER QUASI EXPERIMENT

4.9.1. *Implementation of anti-analysis technique*

Windows has a chained Structured Exception Handler (SEH) mechanism to pass exceptions to handlers instead of crashing the program if possible. Malware can take advantage of SEH to gain control of the malware to detect it is being debugged. The malware throws an exception deliberately, and if its own SEH does not handle the exception, it can deduce that it is being debugged. *OllyDbg* does have a setting to not handle exceptions and to pass exceptions to the process being debugged. Exceptions are handled in the following way for Windows XP SP2, Windows 2003 and Windows Vista (Falliere, 2007, p.5):

- Pass control to the per process Vectored Exception Handler if any.
- Otherwise, pass control to the per thread SEH which is pointed to by FS:[0] in the thread that generated the exception.
- If not processed by the previous two steps, the final SEH in the chain will call the kernel32 function UnhandledExceptionFilter which is set by the system. This function will determine what to do next dependent upon whether the program is being debugged or not. If not being debugged, a user defined filter function will be called, that is set by the kernel32 function SetUnhandledExceptionFilter. If it is being debugged, the program is terminated.

Two types of exception handlers are (Gordon, n.d.) :

- Final exception handler.
- Per thread exception handler.

The final exception handler is set up in the main thread by a call to the API function `SetUnhandledExceptionFilter` which replaces the top level exception handler that Win32 places at the top of each thread and process. If an exception occurs after this call “in a process that is not being debugged, and the exception makes it to the Win32 unhandled exception filter, that filter will call the exception filter function specified by the `lpTopLevelExceptionFilter` parameter”(+Pumpqara, n.d.). A modified version of an example developed by +Pumqara is shown in Figure 4-26.

```
.686
.model flat, stdcall
option casemap:none

include c:\masm32\INCLUDE\Windows.inc
include c:\masm32\INCLUDE\user32.inc
include c:\masm32\INCLUDE\kernel32.inc
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib

.data
caption db "SetUnhandledExceptionFilter",0
text4 db "Return Point from Handler", 0
text1 db "In Handler",0

.code
ExceptionHandler proc
      INVOKE MessageBox, 0, addr text1, addr caption, MB_OK
      ; get the EXCEPTION_POINTERS structure from the stack
      MOV EAX, DWORD PTR [ESP+4] ;
      ; from the EXCEPTION_POINTERS structure, get the pointer
      ; to the CONTEXT structure
      MOV EAX, [EAX+4] ; CONTEXT
       ASSUME EAX:PTR CONTEXT
; change the regEip member of the CONTEXT to the safe address
      MOV [EAX].regEip, OFFSET SafeAddress  ; Change regEip
      ; Set EXCEPTION_CONTINUE_EXECUTION flag in EAX
      XOR EAX,EAX
      DEC EAX
      RETN 4      ; Normalize stack and return
ExceptionHandler endp

start:
       ; register the exception handler
      INVOKE SetUnhandledExceptionFilter,offset ExceptionHandler
      ; force a divide by 0 exception
      XOR EAX,EAX
      DIV EAX

SafeAddress:
      INVOKE MessageBox, 0, addr text4, addr caption, MB_OK
      INVOKE ExitProcess,0
end start
```

Figure 4-26 Listing of implementation of SetUnhandledExceptionFilter technique.

4.9.2. Effectiveness of anti-analysis technique observation

This technique was found to be very effective. If the program is run normally, the exception handler will be called after the deliberate divide by zero exception, and will return to the location of SafeAddress. If the program is run in *OllyDbg*, the program will not enter the exception handler and will crash, unless the debug options are set to pass the exceptions to the program. *IDA Pro* performs in a very similar manner and can also be set to pass exceptions to the application. Many packers use this technique to make the analysis process more difficult. This is because if the program is being debugged, the top level exception handler is never called. Only the per thread or per process handler is called.

4.9.3. Implementation of detection of analysis avoidance technique

Detection of this technique was accomplished by searching for a call to `SetUnhandledExceptionFilter` function call.

4.9.4. Effectiveness of detection of technique observation

The detection technique proved to be effective.

4.9.5. Implementation of mitigation technique

The type of exception that is raised could be examined as well as the handler and patched out if it assists the analysis.

4.9.6. Effectiveness of mitigation technique observation

The mitigation technique proved to be effective.

4.10. NTSETINFORMATIONTHREAD() QUASI EXPERIMENT

4.10.1. Implementation of anti-analysis technique

A thread can be hidden from a debugger by using the `ntdll` function `NtSetInformationThread`. This is usually used for setting the priority of a thread, but can be used to prevent debugging events from being sent to the debugger. It's prototype is as follows in Figure 4-27.

```
NTSYSAPI NTSTATUS NTAPI NtSetInformationThread(
IN HANDLE ThreadHandle,
IN THREAD_INFORMATION_CLASS ThreadInformationClass,
IN PVOID ThreadInformation,
IN ULONG ThreadInformationLength
);
```

Figure 4-27 NtSetInformationThread signature

The ThreadInformationClass has to be set to 0x11 to hide the thread, which essentially detaches the thread. The following listing, an extension of an example by (Falliere, 2007, p.7) demonstrates this technique.

```
.386
.model flat,stdcall
option casemap:none
include c:\masm32\include\windows.inc
include c:\masm32\include\user32.inc
include c:\masm32\include\kernel32.inc
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\user32.lib

.data
LibName db "ntdll.dll",0
FunctionName db "NtSetInformationThread",0
DllNotFound db "Cannot load library",0
AppName db "Load Library",0
FunctionNotFound db "Function not found",0
strAllOk db "Debugger Not Found", 0

.data?
hLib dd ?           ;  the handle of the library (DLL)
FunctionAddr dd ?   ; the address of the function

.code
start:
  invoke LoadLibrary,addr LibName
  .if eax==NULL
    invoke MessageBox,NULL,addr DllNotFound,addr AppName,MB_OK
     .else
       mov hLib,eax
       invoke GetProcAddress,hLib,addr FunctionName
        .if eax==NULL
 invoke MessageBox,NULL,addr FunctionNotFound,addr AppName,MB_OK
         .else
           mov FunctionAddr,eax
           push 0
           push 0
           push 11h
           push -2
           call [FunctionAddr]
    invoke MessageBox, NULL, addr strAllOk, addr AppName, MB_OK
         .endif
          invoke FreeLibrary,hLib
       .endif
    invoke ExitProcess,NULL
end start
```

Figure 4-28 Listing of implementation of NtSetInformationThread technique.

4.10.2. *Effectiveness of anti-analysis technique observation*

When run outside a debugger, the `MessageBox` will display the message that the debugger was not found. If stepped in *OllyDbg*, the thread will be detached, and an error will be displayed that access is denied when trying to exit from the debugger, effectively detecting the presence of the debugger.

4.10.3. *Implementation of detection of analysis avoidance technique*

The use of this technique can be detected by locating calls to `NtSetInformationThread`.

4.10.4. *Effectiveness of detection of technique observation*

The detection technique proved to be effective.

4.10.5. *Implementation of mitigation technique*

The *OllyAdvanced* option `ZwSetInformationThread` can be set to mitigate this technique or when the *IDA Stealth* plugin `NtSetInformationThread` option is selected.

4.10.6. *Effectiveness of mitigation technique observation*

IDA Pro was not detected when run without breakpoints, but detached the thread when stepped through with the debugger. The debugger was not detected when the *IDA Stealth* plugin `NtSetInformationThread` option was selected. If the first breakpoint is set one instruction (or more) beyond the call to the function, the breakpoint is reached ok, effectively mitigating the technique.

4.11. KERNEL32 CLOSEHANDLE() AND NTCLOSE()QUASI EXPERIMENT

4.11.1. *Implementation of anti-analysis technique*

The presence of a debugger can be detected by making use of the `ZwClose` system call. `CloseHandle` indirectly makes use of this call. Calling `ZwClose` with an invalid handle will generate a `STATUS_INVALID_HANDLE` exception.

Falliere (2007, p.7) says that "the only proper way to bypass the CloseHandle anti-debug is to either modify the system call data from ring 0, before it is called, or set up a kernel hook." The listing in Figure 4-29, an extension of an example provided by (Falliere, 2007, p.7) demonstrates this technique.

```
.686
.MODEL flat, stdcall
OPTION CASEMAP:NONE   ;Case sensitive

include windows.inc
include kernel32.inc
includeLib c:\masm32\lib\kernel32.lib
include user32.inc
includeLib c:\masm32\lib\user32.lib

.DATA
      text1 db 'Debugger Not Detected', 0
      caption db 'Heap Flags',0
      text2 db 'Debugger Detected', 0
.CODE
Start:
      PUSH OFFSET Finish
      PUSH 1234h       ; invalid handle
      CALL CloseHandle
      INVOKE MessageBox, 0, addr text1, addr caption, MB_OK
      JMP Finish
DebuggerDetected:
      INVOKE MessageBox, 0, addr text2, addr caption, MB_OK
Finish:
      INVOKE ExitProcess, 0
End Start
```

Figure 4-29 Listing of Kernel32 CloseHandle technique to detect presence of debugger

4.11.2. Effectiveness of anti-analysis technique observation

The program runs fine outside a debugger, but inside *OllyDbg*, the STATUS_INVALID_HANDLE exception was raised. *IDA Pro* behaved in a very similar manner.

4.11.3. Implementation of detection of analysis avoidance technique

The call to CloseHandle is easy enough to find for detection purposes. An example script to locate functions and their cross references adapted and modified from an example by Eagle (2008a, p.271) is shown in Figure 4-30.

```
#include <idc.idc>

// locate functions and their cross references
// adapted from an example by Chris Eagle, p.271
// The IDA Pro Book

static findFunction(func) {
  auto f, addr, xref, source;
  f = LocByName(func);
  if (f == BADADDR) {
    Message("%s not located\n", func);
  }
  else {
    for (addr = RfirstB(f); addr != BADADDR; addr = RnextB(f,
addr)) {
      xref = XrefType();
      if (xref == fl_CN || xref == fl_CF) {
        source = GetFunctionName(addr);
        Message("%s is called from 0x%x in %s\n", func, addr,
source);
      }
    }
  }
}

static main() {
   // add functions to find
   findFunction("CloseHandle");
}
```

Figure 4-30 Listing of findFunction script adapted from Eagle (2008a, p.271)

4.11.4. Effectiveness of detection of technique observation

The detection technique was found to be effective.

4.11.5. Implementation of mitigation technique

IDA Stealth has an NtClose option that can be used to mitigate this technique.

4.11.6. Effectiveness of mitigation technique observation

The mitigation technique was found to be effective.

4.12. USER-MODE TIMERS QUASI EXPERIMENT

4.12.1. Implementation of anti-analysis technique

Packers and debug detection routines take advantage of the fact that code running in a debugger is going to take longer to execute than when not running in a debugger. The routines measure the time elapsed and compare it with a normal run time value. If it took longer to run than expected, then it is probably running in a debugger. The `RDTSC` (Read Time Stamp Counter) instruction can be used before and after a routine to determine how much time elapsed.

The `kernel32` DLL has a function called `GetTickCount` that returns with the number of milliseconds elapsed since the system was started. A `SharedUserData` data structure is always located at address `0x7FFE0000` and contains the fields `TickCountLow` and `TickCountMultiplier`.

The following listing, in Figure 4-31, shows an full implementation of a partial example presented by Yason (2007, p. 8). It shows how the RDTSC instruction can be used to determine if the program could be being stepped in a debugger.

```
; this code uses the RDTSC instruction to get the time stamp
; before and after a section of timed code to determine if it
; is being debugged.

.686
.MODEL flat, stdcall
OPTION CASEMAP:NONE   ;Case sensitive

Include windows.inc
Include kernel32.inc
IncludeLib c:\masm32\lib\kernel32.lib
Include user32.inc
IncludeLib c:\masm32\lib\user32.lib

.DATA
     text1 db 'Debugger Not Detected', 0
     caption db 'RDTSC',0
     text2 db 'Debugger Detected', 0
.CODE
Start:
     ; result of RDTSC returned in EDX:EAX
     RDTSC
     PUSH EAX
     PUSH EDX
     ; just a delay to simulate some function
     MOV ECX, 10
L1:  NOP
     LOOP L1
     ; get time stamp again
     RDTSC
     ; work out the delta
     POP EBX
     CMP EDX, EBX
     JA DebuggerDetected
     POP EBX
     SUB EAX, EBX
     CMP EAX, 500h
     JA DebuggerDetected
     INVOKE MessageBox, 0, addr text1, addr caption, MB_OK
     JMP Finish
DebuggerDetected:
     INVOKE MessageBox, 0, addr text2, addr caption, MB_OK
Finish:
     INVOKE ExitProcess, 0
End Start
```

Figure 4-31 Listing of implementation of RDTSC technique to detect presence of a debugger.

4.12.2. *Effectiveness of anti-analysis technique observation*

This technique proved to be effective at detecting the presence of a debugger.

4.12.3. *Implementation of detection of analysis avoidance technique*

This use of this technique can be found by locating the instruction `RDTSC`.

4.12.4. *Effectiveness of detection of technique observation*

Locating calls to the instruction `RDTSC` proved to be effective

4.12.5. *Implementation of mitigation technique*

A simple solution to this technique would be to identify where the timing checks are being performed in the code, and then set a breakpoint before the first time delta measurement and then perform a run instead of a step until the breakpoint is hit (Yason, 2007, p.9). Alternatively the result returned from a call to `GetTickCount` and modify the return value. Yason says that *OllyAdvanced* installs a kernel mode driver that sets the Time Stamp Disable bit (TSD) in the `CR4` control register which will trigger a General Protection (GP) exception if the `RDTSC` instruction is executed in a privilege level other than 0. The Interrupt Descriptor Table (IDT) is setup so that the GP exception is hooked and the execution of the RDTSC is filtered. Yason emphasises that this driver may cause instability to the system.

4.12.6. *Effectiveness of mitigation technique observation*

OllyAdvanced has two anti `RDTSC` options, but the debugger was still detected. The most effective mitigation strategy was to locate the calls to the function and patch out appropriately.

4.13. KERNEL32 OUTPUTDEBUGSTRINGA() QUASI EXPERIMENT

4.13.1. *Implementation of anti-analysis technique*

Falliere (2007, p.7) reports that he encountered this technique whilst examining files packed with *ReCrypt v0.80*. If `OutputDebugStringA` is called with a valid ASCII string under the control of a debugger, the return value will the address of the string passed as a parameter. When not run in a debugger, the return value should be 1. This technique is demonstrated in the listing in Figure 4-32. Yason (2007, p.26) says that this technique is specific to *OllyDbg* because it is vulnerable to a format string bug.

```
; this code will detect the presence of OllyDbg v1.1 and
; v2.0 alpha by exploiting a string format vulnerability

.686
.MODEL flat, stdcall
OPTION CASEMAP:NONE   ;Case sensitive

include windows.inc
include kernel32.inc
includeLib c:\masm32\lib\kernel32.lib
include user32.inc
includeLib c:\masm32\lib\user32.lib

.DATA
     text1 db 'Debugger Not Detected', 0
     caption db 'OutputDebugStringA',0
     text2 db 'Debugger Detected', 0
     textString db 'My Test String', 0
.CODE
Start:
     XOR EAX,EAX
     INVOKE OutputDebugString, addr textString
     CMP EAX, 1
     JNE DebuggerDetected
     INVOKE MessageBox, 0, addr text1, addr caption, MB_OK
     JMP Finish
DebuggerDetected:
     INVOKE MessageBox, 0, addr text2, addr caption, MB_OK
Finish:
     INVOKE ExitProcess, 0
End Start
```

Figure 4-32 Listing of implementation of OutputDebugStringA to detect presence of a debugger.

4.13.2. Effectiveness of anti-analysis technique observation

The technique worked in *OllyDbg* v1.10 and *OllyDbg* v2.00 (alpha2) and it was found that the technique also worked in *IDA Pro*.

4.13.3. Implementation of detection of analysis avoidance technique

This technique can be detected by adding the following line to the main function in the listing in Figure 4-30:

```
     findFunction(“OutputDebugStringA”);
```

A technique that works to detect and then patch the return result from OutputDebugStringA is provided in Figure 4-33 which was extracted and modified from an example by (Eagle, 2008b).

```
#include <idc.idc>

static main() {
   auto addr, funcName, end;
   funcName = "kernel32_OutputDebugStringA";
   // run to entry point
   RunTo(BeginEA());
   // wait until process is suspended
   GetDebuggerEvent(WFNE_SUSP, -1);
   // locate address of function
   addr = LocByName(funcName);
   if (addr != BADADDR) {
      Message("%s found at %x\n", funcName, addr);
      MakeFunction(addr, BADADDR);
      end = GetFunctionAttr(addr, FUNCATTR_END) - 3;
      AddBpt(end);
      SetBptAttr(end, BPT_BRK, 0);  //don't stop
      //fix   the   return   value   as   expected   in   non-debugged
processes
      SetBptCnd(end, "EAX = 1");
   } else {
      Message("%s not found\n", funcName);
   }
}
```

Figure 4-33 Script to patch result of OutputDebugStringA function call to hide presence of debugger.

4.13.4. Effectiveness of detection of technique observation

The use of the detection techniques was found to be effective.

4.13.5. Implementation of mitigation technique

This technique can be mitigated by enabling the OutputDebugString option in *IDA Stealth*. Alternatively, the listing in Figure 4-33 can be employed.

4.13.6. Effectiveness of mitigation technique observation

The mitigation techniques were found to be effective.

4.14. ROGUE INT3 QUASI EXPERIMENT

4.14.1. *Implementation of anti-analysis technique*

The idea of this technique is to insert INT3 opcodes into the binary to trick the debugger into thinking it is one of the software breakpoints it has inserted into the binary being debugged. Control will be given to an exception handler when the INT3 is encountered in a program that is not being debugged and the program continues executing. Debuggers typically handle these debugger interrupts themselves. The exception handler of the malware can set flags so that it can determine if it is running in a debugger if the exception handler is not entered. Yason (2007, p.7) says that the kernel32 DLL function DebugBreak() internally invokes an INT3 and this can be used instead. An example presented by ap0x (2006) is presented in Figure 4-34. It sets the value of EAX to 0xFFFFFFFF (via the CONTEXT record) in the exception handler to flag the fact that the exception handler has been entered. The purpose of the context record is to contain the state of a thread. The context record that is passed to an exception handler contains the current state of the thread that threw the exception (Yason, 2007, p.8). Yason (2007, p.7) points out that the kernel32 DLL function DebugBreak() internally invokes INT3, and some packers use this call instead of using INT3 directly.

```
.386
.model flat, stdcall
option casemap :none   ; case sensitive

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data
msgTitle db "Execution status:",0h
msgText1 db "No debugger detected!",0h
msgText2 db "Debugger detected!",0h
.code

start:

; MASM32 antiRing3Debugger example
; coded by ap0x
; Reversing Labs: http://ap0x.headcoders.net
; This code takes advantage of debugger not handling INT3
; instructions correctly. If we set a SEH before INT3 executing
; INT3 instruction will fire SEH. If debugger is present it
; will just walk over INT3 and go straight forward.
; If debugger is not present exception will occur and execution
; will be handled by SEH.

; Set SEH
ASSUME FS:NOTHING
PUSH offset @Check
PUSH FS:[0]
MOV FS:[0],ESP

; Exception
INT 3h

PUSH 30h
PUSH offset msgTitle
PUSH offset msgText2
PUSH 0
CALL MessageBox

PUSH 0
CALL ExitProcess

; SEH handling
@Check:
POP FS:[0]
ADD ESP,4

PUSH 40h
PUSH offset msgTitle
PUSH offset msgText1
PUSH 0
CALL MessageBox

PUSH 0
CALL ExitProcess

end start
```

Figure 4-34 Listing of implementation of INT3 technique to detect the presence of a debugger (ap0x, 2006)

4.14.2. Effectiveness of anti-analysis technique observation

This code successfully detects that it is running in *`OllyDbg`.*

4.14.3. Implementation of detection of analysis avoidance technique

This technique can be detected by searching the code for the `INT3` instruction.

4.14.4. Effectiveness of detection of technique observation

Searching for the presence of `INT3` instructions was found to be effective.

4.14.5. Implementation of mitigation technique

This technique can be mitigated in a couple of different ways. The first solution was to allow the interrupts to be automatically passed to the exception handler by setting the debugging options to pass *`INT3`* breaks and Single-step breaks to the program. Another method was to identify the exception handler address (in *`OllyDbg`*, `View > SEH Chain`) and then set a breakpoint on the exception handler. Then the exception can be passed to the exception handler by pressing Shift + F9, and the code of the exception handler can be traced. Note that you have to step through (or set a breakpoint) the code until the SEH is installed before you can see it in the SEH window.

4.14.6. Effectiveness of mitigation technique observation

The mitigation techniques were found to be effective. A software breakpoint exception was raised in *`IDA Pro`* when the program was run and an option to pass the exception to the program is offered via a dialog box. If the exception is not passed to the program, the debugger was detected, otherwise the debugger is not detected. Essentially, this technique can be mitigated by setting an option to pass breakpoint exceptions to the program.

4.15. "ICE" BREAKPOINT QUASI EXPERIMENT

4.15.1. Implementation of anti-analysis technique

The `Ice` breakpoint is an undocumented Intel instruction that can be used to detect programs that are being debugged. Its opcode is `0xF1`. This instruction generates a `SINGLE_STEP` exception when executed and the debugger will not call the exception handler and execution will not continue as expected. An example implementation is shown in the listing in Figure 4-35, which is a very simple modification to the example developed by ap0x which was shown above in the listing of Figure 4-35.

```
.386
.model flat, stdcall
option casemap :none   ; case sensitive

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

    .data
msgTitle db "Execution status:",0h
msgText1 db "No debugger detected!",0h
msgText2 db "Debugger detected!",0h
    .code

start:

; Set SEH
ASSUME FS:NOTHING
PUSH offset @Check
PUSH FS:[0]
MOV FS:[0],ESP

; Exception
db 0F1h

PUSH 30h
PUSH offset msgTitle
PUSH offset msgText2
PUSH 0
CALL MessageBox

PUSH 0
CALL ExitProcess

; SEH handling
@Check:
POP FS:[0]
ADD ESP,4

PUSH 40h
PUSH offset msgTitle
PUSH offset msgText1
PUSH 0
CALL MessageBox

PUSH 0
CALL ExitProcess

end start
```

Figure 4-35 Listing of implementation of Ice Breakpoint technique to detect the presence of a debugger.

4.15.2. *Effectiveness of anti-analysis technique observation*

This technique was successful with *`OllyDbg`* and *`IDA Pro`*

4.15.3. *Implementation of detection of analysis avoidance technique*

The use of this technique can be found by searching for the opcode `F1h`.

4.15.4. *Effectiveness of detection of technique observation*

The detection technique was found to be effective.

4.15.5. *Implementation of mitigation technique*

This technique can be overcome by setting the debugging options to pass single-step breaks to the program.

4.15.6. *Effectiveness of mitigation technique observation*

The mitigation technique was found to be effective.

4.16. INTERRUPT 2DH QUASI EXPERIMENT

4.16.1. *Implementation of anti-analysis technique*

Interrupt `2Dh` will raise a breakpoint exception if the program is not being debugged. Note how this is different to the other examples. If a debugger is attached, there will not be an exception. This technique is demonstrated in the listing shown in Figure 4-36.

```
.386
.model flat, stdcall
option casemap :none   ; case sensitive

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data
msgTitle db "Execution status:",0h
msgText1 db "No debugger detected!",0h
msgText2 db "Debugger detected!",0h
.code

start:
; Set SEH
ASSUME FS:NOTHING
PUSH offset @Check
PUSH FS:[0]
MOV FS:[0],ESP

; Exception
INT 2DH
POP FS:[0] ; clear the SEH
ADD ESP, 4

INVOKE  MessageBox, 0, offset msgText2, offset msgTitle, 30h
JMP Finish

; SEH handling
@Check:
POP FS:[0]
ADD ESP,4
INVOKE  MessageBox, 0, offset msgText1, offset msgTitle, 40h

Finish:
PUSH 0
CALL ExitProcess

end start
```

Figure 4-36 Listing showing use of INT 2DH to raise an exception if the program is not being debugged.

4.16.2. Effectiveness of anti-analysis technique observation

This was effective in both *OllyDbg* and *IDA Pro*.

4.16.3. Implementation of detection of analysis avoidance technique

This technique can be detected by search for Interrupt 2Dh.

4.16.4. Effectiveness of detection of technique observation

The detection technique proved to be effective.

4.16.5. Implementation of mitigation technique

This technique can be overcome by setting the debugger options to pass all exceptions to the program being debugged.

4.16.6. Effectiveness of mitigation technique observation

The mitigation technique proved to be effective.

4.17. POPF AND THE TRAP FLAG QUASI EXPERIMENT

4.17.1. Implementation of anti-analysis technique

The trap flag in the Flags register is used to control the tracing of a program. If the trap flag is set, an instruction that is being executed will raise a `SINGLE_STEP` exception. Falliere (2007, p.10) says that this can be used to thwart tracers. A working implementation using Falliere’s snippet of code is given in the listing of Figure 4-37. This will have no effect on the flags register of a program that is being traced. The debugger will process the exception that is raised, and the associated exception handler will not be executed.

```
.386
.model flat, stdcall
option casemap :none   ; case sensitive

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data
msgTitle db "Execution status:",0h
msgText1 db "No debugger detected!",0h
msgText2 db "Debugger detected!",0h
.code
start:
; Set SEH
ASSUME FS:NOTHING
PUSH offset @Check
PUSH FS:[0]
MOV FS:[0],ESP
; Exception
PUSHF
MOV EAX, 100h
MOV  [ESP], EAX
POPF
INVOKE  MessageBox, 0, offset msgText2, offset msgTitle, 30h
JMP Finish
; SEH handling
@Check:
POP FS:[0]
ADD ESP,4
INVOKE  MessageBox, 0, offset msgText1, offset msgTitle, 40h
Finish:
PUSH 0
CALL ExitProcess
end start
```

Figure 4-37 Listing of implementation POPF and the Trap Flag technique to detect the presence of a debugger.

4.17.2. Effectiveness of anti-analysis technique observation

OllyDbg and *IDA Pro* were detected if the exception was not passed to the program.

4.17.3. Implementation of detection of analysis avoidance technique

The use of this technique can be detected by examining exceptions.

4.17.4. Effectiveness of detection of technique observation

This technique proved to be effective.

4.17.5. Implementation of mitigation technique

This was defeated by passing all raised exceptions to the program being debugged.

4.17.6. Effectiveness of mitigation technique observation

This technique proved to be effective.

4.18. SUMMARY OF VALIDATION OF TECHNIQUES RESULTS

The literature review revealed a large and wide variety of techniques malware can incorporate to hinder analysis and avoid detection. A subset of these techniques were implemented and validated in small, standalone programs. All of the implemented techniques were observed to be effective at detecting the presence of a debugger, namely *IDA Pro* and *OllyDbg*. After ensuring that the anti-analysis technique was effective, small scripts were developed or sourced to determine if the use of the technique could be detected. All of the implemented detection techniques were observed to be effective. Mitigation scripts were then developed or sourced to determine if the use of the technique could be mitigated. All of the implemented mitigation techniques or scripts were observed to be effective. A summary of the results is provided in Table 4-1.

Table 4-1 Validation of Techniques Results

Technique	Implemented in Code	Debugger Detection		Technique Detectable	Technique Mitigatable
		IDA Pro	OllyDbg		
IsDebuggerPresent	✓	✓	✓	✓	✓
IsDebugged	✓	✓	✓	✓	✓
NtGlobalFlags	✓	✓	✓	✓	✓
Heap Flags	✓	✓	✓	✓	✓
NtQueryInformationProcess	✓	✓	✓	✓	✓
CheckRemoteDebuggerPresent	✓	✓	✓	✓	✓
UnhandledExceptionFilter	✓	✓	✓	✓	✓
NtSetInformationThread	✓	✓	✓	✓	✓
CloseHandle	✓	✓	✓	✓	✓
User Mode Timers	✓	✓	✓	✓	✓
OutputDebugString	✓	✓	✓	✓	✓
INT 3	✓	✓	✓	✓	✓
ICE Breakpoint	✓	✓	✓	✓	✓
INT 2DH	✓	✓	✓	✓	✓
POPF	✓	✓	✓	✓	✓

These results provide a significant measure of validation for the anti-analysis techniques discussed in the literature review (Falliere, 2007; Ferrie, 2008; Yason, 2007).

The ability to detect the use of anti-analysis techniques provides confidence in being able to implement an application to detect malware, as suggested by Wysopal (2009) who said that the use of such techniques could be a very good indicator of the program under investigation to possibly have a malicious nature. This is important, because the literature review represented claims that existing malware detection paradigms are less than effective and that a new approach is required.

The literature review showed that the coverage of anti-analysis techniques in popular plugins was limited. These results show that mitigation scripts can be very useful to extend the coverage of such plugins to aid in the analysis of malicious software and to hide the presence of analysis tools.

Significant programming and operating system knowledge is required to detect and mitigate the techniques malware can incorporate to avoid analysis, as evidenced in the programs and scripts used to derive the results in this chapter. The conduct of this research led to the identification of a Malware Analysis Body of Knowledge by Valli and Brand (2008) that attempts to identify an appropriate spectrum of knowledge required to analyse malicious software. A key component of the MABOK is the treatment of anti-analysis techniques. The MABOK is discussed in greater detail in section 6.7.9 of this thesis.

CHAPTER 5 ANALYSIS OF COLLECTED MALWARE RESULTS

5.1. OVERVIEW

The purpose of this chapter is to present the results of examination of network based malware collected by the ECU *Nepenthes* sensors as described in the Conceptual Framework, section 3.6 of this thesis. It examines claims in the literature review that existing approaches to the detection of malware is less than effective and supports research question two, that is, "How can the use of these techniques be detected?". In general, the literature review discussed three techniques common to AV software to detect malicious software as signature recognition, heuristics and file integrity checking. To this end, the effectiveness of existing virus signature detection is examined as well as an examination of the effectiveness of heuristics.

An additional facet of this chapter is an examination into the use of run time packers which are arguably, one of the most fundamentally used analysis avoidance techniques. Methods that can be used to detect the use of run time packers, include packer signature recognition and by measures of entropy (randomness) in the code.

898 malware samples were collected by the ECU *Nepenthes* sensors between June 25 2007 and August 9 2008. All samples were collected by the *Nepenthes* system which emulates known vulnerabilities that network based malware takes advantage of, to install malware on the vulnerable computer.

5.2. VIRUS SIGNATURES

5.2.1. Anubis

All 898 samples were submitted to *Anubis* and of these, 738 (82.2%) were able to be analyzed and 160 (17.8%) were not able to be analyzed. Of the 738 specimens of malware that were able to be analyzed, 544 virus signatures were able to be determined by the *Ikarus* virus scanner. This represents a detection rate by the `Ikarus` virus scanner of 73.7%. This is the solitary virus scanner `Anubis` uses. The results of `Ikarus` are shown in Table 5-1 and it is clearly dominated by the `Allaple` (Anonymous, n.d.-a) worm. Variants exist of most types of malware and these variants were grouped together where possible in the results in Table 5-1.

Table 5-1 Ikarus Virus Scanner results showing high incidence of Allaple worm in the collected malware specimens.

Ikarus Signature	***Count***	***%***
Allaple	422	77.57
Virut	30	5.51
PoeBot	17	3.13
Rbot	16	2.94
Agent	10	1.84
Nepoe	8	1.47
SdBot	7	1.29
Delf	6	1.10
WinFixer	4	0.74
VanBot	4	0.74
Hupigon	3	0.55
NSPM	3	0.55
Lovesan	2	0.37
ProcessHijack	2	0.37
IRCBot	2	0.37
oda	1	0.18
Lineage	1	0.18
AHKD	1	0.18
Adload	1	0.18
Zlob	1	0.18
Klone	1	0.18
Slaper	1	0.18
Sasser	1	0.18

5.2.2. Virus Total

`Virus Total` is an online virus scanner site that accepts uploaded files which are then processed by up to 36 virus scanner engines. Results for the submission of a particular collected specimen to `Virus Total` are given in Table 5-2 as an example. It shows that 33 of the 36 virus engines recognized the signature of the malware. This particular specimen was collected on October 17 2007 and the analysis was conducted on September 4 2008.

Table 5-2 Virus Total results from a single submission showing disparity in signatures by different vendors.

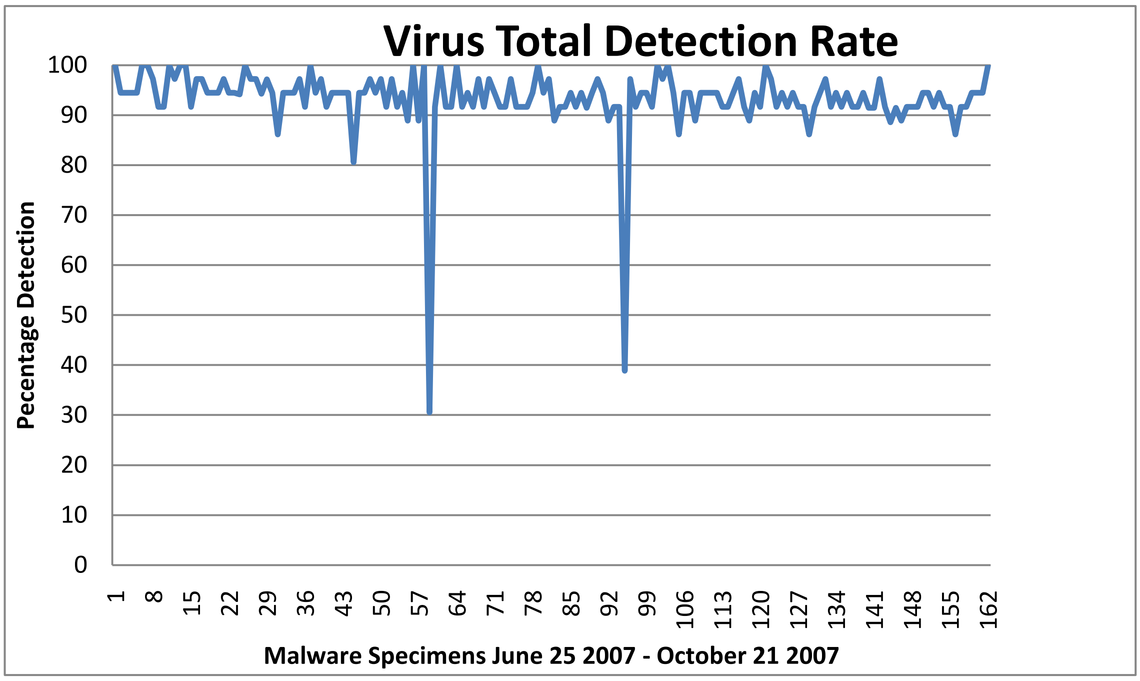
Anti Virus Scanner	*Result*
AhnLab-V3	Win32/Allaple.worm.B
AntiVir	WORM/Allaple.Gen
Authentium	W32/RAHack.A.gen!Eldorado
Avast	Win32:Allaple
AVG	Worm/Allaple.B
BitDefender	Win32.Worm.Allaple.Gen
CAT-QuickHeal	I-Worm.Allaple.gen
ClamAV	Worm.Allaple-311
DrWeb	Trojan.Starman
eSafe	Suspicious File
eTrust-Vet	Win32/Mallar
Ewido	-
F-Prot	W32/RAHack.A.gen!Eldorado
F-Secure	Net-Worm.Win32.Allaple.b
Fortinet	W32/ALLAPLE.E!worm
GData	Net-Worm.Win32.Allaple.b
Ikarus	Net-Worm.Win32.Allaple.a
K7AntiVirus	Net-Worm.Win32.Allaple.a
Kaspersky	Net-Worm.Win32.Allaple.b
McAfee	W32/RAHack
Microsoft	Worm:Win32/Allaple.A
NOD32v2	a variant of Win32/Allaple.Gen
Norman	Allaple.gen
Panda	W32/Rahack.gen
PCTools	Worm.Allaple.Gen
Prevx1	-
Rising	Worm.Win32.Allaple.a
Sophos	W32/Allaple-F
Sunbelt	Worm.Win32.Allaple.JF
Symantec	W32.Rahack.W
TheHacker	-
TrendMicro	WORM_ALLAPLE.IK
VBA32	Net-Worm.Win32.Allaple
ViRobot	Worm.Win32.Allaple.Gen
VirusBuster	Worm.Allaple.Gen
Webwasher-Gateway	Worm.Allaple.Gen

The differing naming conventions used by each of the 36 Anti Virus scanners is clearly evident.

162 specimens, collected between June 25 2007 and October 21, were submitted to `Virus Total`. This had been a period of over a year for most of the specimens since they had been collected. Only 17 of the 162 samples (10.4%) were detected by all of the Anti Virus scanners. The results of this test are plotted in Figure 5-1. It indicates that 100% detection by all Anti Virus scanners is not achieved nearly a year after collection and suggests

that the 73.7% virus detection rate by *Ikarus* may be considered within the norm.

Figure 5-1 Virus Total detection rate plot showing less than ideal detection results.



5.3. MALWARE FUNCTIONALITY

Anubis reports contain a summary of the functionality of the malware it analyzes. Functionality reported by *Anubis* for the submitted samples included various combinations of the following reports:

- Performs Address Scan.
- Auto Start Capabilities.
- Creates Files in the Windows System Directory.
- Changes Security Settings of Internet Explorer.
- Joins IRC Network.

Address scans are performed by malware to locate other targets on the network to attack. Auto start capabilities are generally changes made to the registry to ensure that the malware is activated each time the computer is restarted. Malware is generally packed when it is initially loaded onto a vulnerable machine, and usually consists of multiple files which are then copied to various locations in the Windows System directory with the hidden attribute set and given file names that very closely resemble legitimate file names to provide additional camouflage. Security settings are changed in *Internet Explorer* so that more malware can be downloaded from sites without warnings. IRC networks are used by Bots to accept remote commands from a BotNet. Table 5-3 lists the results of the high level malicious activities the malware performed. Note that various combinations of activities are possible.

Table 5-3 Submitted malware functionality results

Malware Function	***Occurrence Count***
Performs Address Scan	301
Auto start capabilities	282
Creates files in the Windows system directory	276
Changes security settings of Internet Explorer	135
Joins IRC network	58

The *Allaple* (Anonymous, n.d.-a) worm was the most representative specimen collected by the sensors. Four variants of this worm were detected, including *Allaple.A*, *Allaple.B*, *Allaple.D* and *Allaple.E*. The number of detections for each variant is presented in Table 5-4.

Table 5-4 Allaple variants detection results

Allaple Variant	**Detections**
Allaple.A	340
Allaple.B	63
Allaple.D	1
Allaple.E	18
Totals	422

Table 5-5 presents the functionality detected by *Anubis* of the variants.

Table 5-5 Functionality of Allaple variants results

Malware Function	Allaple. A	Allaple. B	Allaple. D	Allaple. E
Performs Address Scan	164	29	1	13
Autostart Capabilities	119	22	0	3
Creates Files in the Windows System Directory	112	21	0	4
Changes security settings of Internet Explorer	9	0	0	3
Joins IRC network	0	0	0	0

In contrast, Table 5-6 shows the *Allaple* specimens where *Anubis* did not record any activity at all, even though these particular specimens' run time was around 150 seconds as depicted in Figure 5-2.

Table 5-6 Allaple variants showing no activity recorded

Allaple Variant	No Activity	Ikarus Detections	%
Allaple.A	62	340	18.24%
Allaple.B	13	63	20.63%
Allaple.D	0	1	0.00%
Allaple.E	2	18	11.11%
Totals	77	422	18.25%

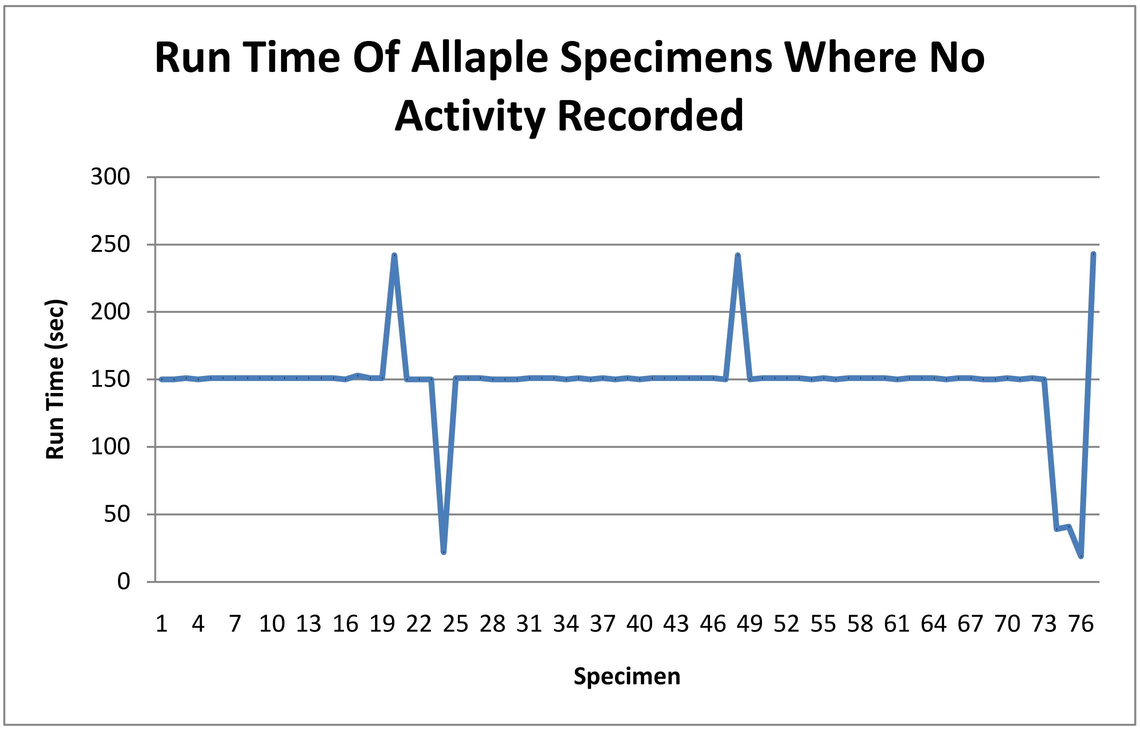


Figure 5-2 Run time Of Allaple specimens where no activity is recorded could indicate deception.

5.4. PACKER ANALYSIS

Anubis uses *SigBuster* as its packer signature detector during the time this research has been conducted and it is not publicly available. The only way to use *SigBuster* is to upload a malware specimen to *Anubis* and have the file analyzed online. Of the 738 samples that were able to be analyzed, the *SigBuster* packer detector recognized 543 signatures of packers, as listed in Table 5-7.

Table 5-7 SigBuster detected packer signature results

SigBuster Signature	Count	%
Allaple_Polymorphic_Packer vna SN: 1647	444	60.16
eXpressor v1.4.5 SN:225	21	2.85
Signature_Safe v2. SN:49	9	1.22
PolyCrypt_PE v2005.06.01 SN:391PolyCrypt_PE v2.1.4b/2.1.5 SN:1150	8	1.08
Themida vna SN:732	6	0.81
eXpressor V1.4 SN: 1654	4	0.54
UPX All_Versions SN:1634	4	0.54
UPX All_Versions SN:1634EXE_Cryptor v2.2X SN:193	4	0.54
Allaple_Polymorphic_Packer vna SN: 1647UPX All_Versions SN:1634	3	0.41
FSG V1.3x SN:1637	3	0.41
Unknown_packer vna SN: 1671Allaple_Polymorphic_Packer vna SN: 1647	3	0.41
ASProtect v1.2x-1.3x SN:137	2	0.27
ASProtect v1.2x-1.3x SN:137ASProtect v2.1/2.2(exe) SN:1424	2	0.27
DotFix NiceProtect vna SN: 1655	2	0.27
EXE_Cryptor v2.2X SN:193	2	0.27
eXpressor V1.4 SN: 1654Unknown_packer vna SN: 1679	2	0.27
eXpressor V1.4 SN: 1654UPX All_Versions SN:1634	2	0.27
PKLITE32 v1.1 SN:1153	2	0.27
Unknown_packer vna SN: 1660	2	0.27
eXpressor V1.4 SN: 1654UPX All_Versions SN:1634EXE_Cryptor v2.2X SN:193	1	0.14
Expressor v1.4 SN: 1672	1	0.14
Expressor v1.4 SN: 1672UPX All_Versions SN:1634	1	0.14
FSG V1.3x SN:1637EXE_Cryptor v2.2X SN:193	1	0.14
NsPack All_Versions SN:1635	1	0.14
PE_Compact v2.0x SN:1610FSG V1.3x SN:1637	1	0.14
PE_Compact v2.X SN:660FSG V1.3x SN:1637	1	0.14
PE_Pack v1.0 SN:1399	1	0.14
PE_Pack v1.0 SN:72	1	0.14
PE_Pack v1.0 SN:72	1	0.14
Unknown_metamorphic_packer vna SN: 1658	1	0.14
Unknown_packer vna SN: 1654	1	0.14
Unknown_packer vna SN: 1654UPX All_Versions SN:1634EXE_Cryptor v2.2X SN:193	1	0.14
Unknown_packer vna SN: 1671Unknown_packer vna SN: 1679	1	0.14
Unknown_packer vna SN: 1676Signature_Safe v2. SN:49	1	0.14
Unknown_packer vna SN: 1679	1	0.14
UPX_xor_stub vna SN:1612	1	0.14
Xtreme_Protector v1.05 SN:78	1	0.14
Total		**73.58**

The *SigBuster* results are dominated by the *Allaple_Polymorphic_Packer vna SN: 1647* signature with 444 occurrences, followed very distantly by variations of *Expressor* with 32 occurrences. In contrast, *Mandiant Red Curtain (MRC)* uses *PEiD* as its packer signature detector which is publicly available together with its database of signatures. The *PEiD* database consists of over 400 signatures. *MRC* only detected 43 known signatures, the results of which are displayed in Table 5-8.

Table 5-8 PEiD signature results indicating disparity in signature matching with those performed by SigBuster.

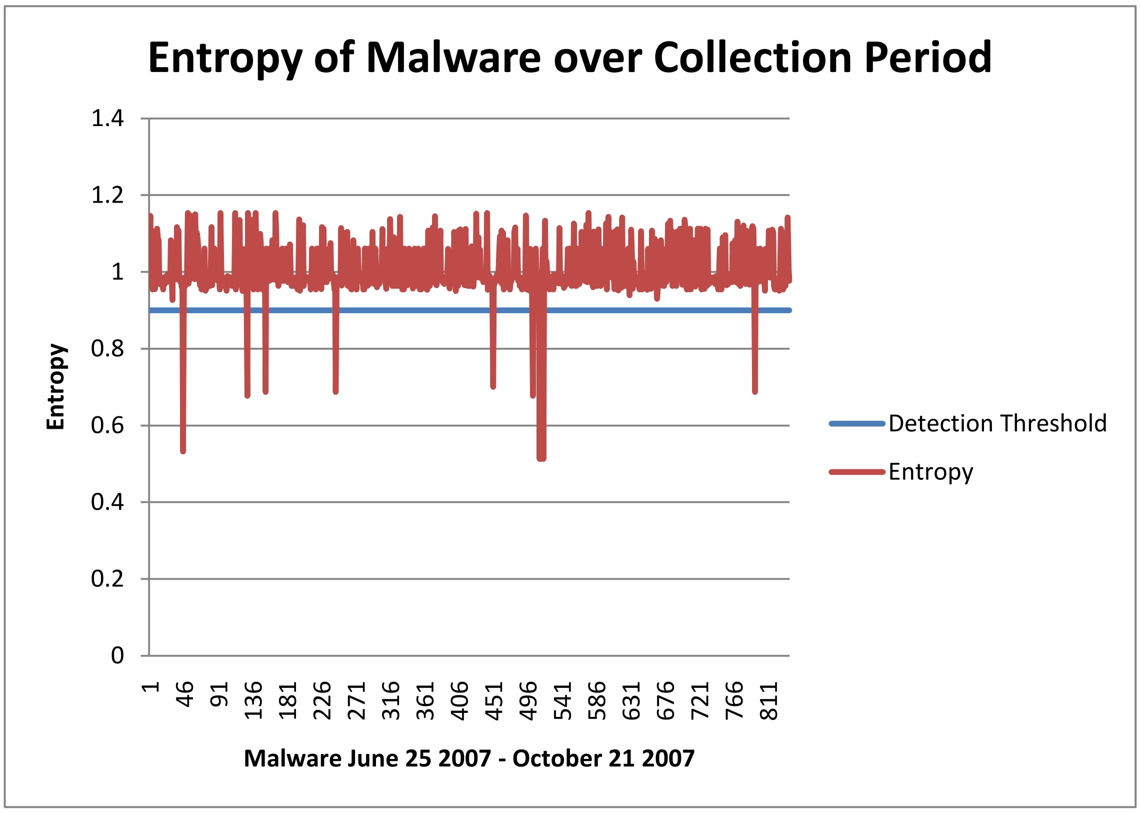
PEiD Signature	Count	%
PECompact v2.x	11	1.49
CodeSafe v2.0	9	1.22
Anticrack Software Protector v1.09 (ACProtect)	4	0.54
Borland Delphi v6.0 - v7.0	3	0.41
Microsoft Visual Basic v5.0 / v6.0	3	0.41
UPX v0.89.6 - v1.02 / v1.05 - v1.22	3	0.41
ASProtect v1.23 RC1	2	0.27
UPX v1.03 - v1.04	1	0.14
PKLITE32 v1.1	1	0.14
PEtite v1.4	1	0.14
NeoLite vx.x	1	0.14
Symantec Visual Cafe v3.0	1	0.14
Microsoft Visual C++ v5.0/v6.0 (MFC)	1	0.14
Xtreme-Protector v1.05	1	0.14
UPX-Scrambler RC v1.x	1	0.14
Total		**5.83**

The significant contrast in results between Table 5-7 and Table 5-8 could be attributed to *SigBusters* ability to detect the *AllAple Polymorphic Packer* which is not in the *PEiD* database of signatures. It is also observed that not a single signature matched between *PEiD* and *SigBuster*.

An alternative method for the detection of the use of a packer is through measuring the entropy (randomness) of the program. *MRC* employs this technique as one of the criteria it uses to develop a risk score to identify malicious software. Packed code has a higher value of entropy than unpacked code. *MRC* uses a value of 0.9 as a threshold to signal files of interest. *MRC* was used to scan the directory of malware and returned 838 results. Of these, 829 returned a measurement of entropy of greater than, or equal to 0.9. This represents 98.9% of the files. In comparison, when *MRC* was used to scan the `C:\Windows\System32` directory of an uncompromised system, 19 files out of 671 executable files returned an entropy of greater than, or equal to 0.9. This represents 2.83% of the files. The entropy method appears to be very successful for the detection of runtime packed files.

Figure 5-3 displays the entropy of the malware that was collected during the period June 25 2007 to October 21 2007 and clearly shows the high level of entropy of the malware.

Figure 5-3 Graph indicating high measures of entropy of malware exceeding accepted threshold



5.5. SUMMARY OF COLLECTED MALWARE RESULTS

The results in this chapter support claims that signature based virus detection is less than ideal. The *Ikarus* Virus Scanner used by *Anubis* only detected 73.7% of malware collected by the ECU *Nepenthes* sensors, even though the malware had been in the wild for a period of up to a year. In addition, the specimens were clearly malicious because they had arrived on the sensors by deliberately exploiting an emulated vulnerability and installed software uninvited.

The functionality of the malware determined by *Anubis* clearly demonstrated malicious intent. This does provide a good indicator of the nature of the malware, however, running the software to determine its nature gives control of the malware to employ deception techniques and does provide an opportunity to the malware to do damage to the system.

Measures of entropy showed to be a very good method to determine if the malware is packed. The results also showed that two different packer signature determination tools provided very different results. This is significant because identification of a packer signature assists in the determination of the appropriate unpacking algorithm to employ to unpack the malware to arrive at the OEP so that detailed analysis can commence.

CHAPTER 6 DISCUSSION

6.1. DISCUSSON OF VALIDATION OF ANTI-ANALYSIS TECHNIQUES RESULTS

Only a subset of the techniques discussed in the Literature Review of this thesis were validated due to the extensive number of techniques uncovered through a search of the literature and software reverse engineering sites. The validation process included implementing individual techniques in simple, standalone programs, running the program in two popular debuggers (*IDA Pro* and *OllyDbg*) and then observing whether or not, the debugger was able to be detected. The simple nature of the validating program was designed to ensure that no other factor was present to account for the behaviour of the program. This was followed by writing or sourcing a detection and mitigation script or method and observing the result. A summary of the results of the techniques that were validated is presented in Table 6-1. The check symbol (✓) designates that the technique was successful, whilst the use of the cross symbol (×) would have been used to designate failure of the technique.

All of the implemented techniques were successful in detecting the presence of the two debuggers. The use of these anti-analysis techniques was successfully detected and mitigated using scripts or via manual methods. The techniques that were not implemented and discussed by other researchers (Eagle, 2008b; Falliere, 2007; Ferrie, 2008; Yason, 2007) appear to be sound, and this researcher is confident that these techniques would also be able to be detected and mitigated successfully.

Documentation for scripting languages to support the validation activity was found to be sparse and mostly focused on function definitions. Learning how to implement scripts to perform a particular function was attained by examination of existing scripts from reverse engineering software web sites such as *Tuts4You* (T. Rogers, 2008) and analyzing how they were implemented. Scripting languages provide a rich set of functionality and are essential for analysis of malware that employs anti-analysis techniques. The same scripting languages are also extremely useful for detection and mitigation of anti-analysis techniques.

Table 6-1 Validation of techniques results showing validity of technique and the ability to detect and mitigate the techniques.

Technique	Implemented in Code	Debugger Detection		Technique Detectable	Technique Mitigatable
		IDA Pro	*OllyDbg*		
IsDebuggerPresent	✓	✓	✓	✓	✓
IsDebugged	✓	✓	✓	✓	✓
NtGlobalFlags	✓	✓	✓	✓	✓
Heap Flags	✓	✓	✓	✓	✓
NtQueryInformationProcess	✓	✓	✓	✓	✓
CheckRemoteDebuggerPresent	✓	✓	✓	✓	✓
UnhandledExceptionFilter	✓	✓	✓	✓	✓
NtSetInformationThread	✓	✓	✓	✓	✓
CloseHandle	✓	✓	✓	✓	✓
User Mode Timers	✓	✓	✓	✓	✓
OutputDebugString	✓	✓	✓	✓	✓
INT 3	✓	✓	✓	✓	✓
ICE Breakpoint	✓	✓	✓	✓	✓
INT 2DH	✓	✓	✓	✓	✓
POPF	✓	✓	✓	✓	✓

6.2. DISCUSSION OF COLLECTED MALWARE ANALYSIS RESULTS

The *Nepenthes* sensors work by emulating known vulnerabilities and allowing network based malware to install itself on the vulnerable computer. It could be considered that any software that takes advange of such vulnerabilities and installs itself on a computer over the internet, uninvited, be categorised as malicious. The malware collected by the ECU Nepenthes sensors was validated as malicious software by its behaviour, however the detection rate by the *Ikarus* virus detector employed by *Anubis* was approximately 73.7%. A detection rate of much lower than 100% may not be an unusual result when compared with other virus detectors results as performed when the malware was submitted to *Virus Total,* which employs up to thirty six AV engines from various vendors .

A continuous subset of the malware collected between June 25 and October 21 2007 was submitted to *VirusTotal* on or around September 04 2008.

Even though each malware specimen was submitted to 36 virus detectors, approximately one year after collection and submission to online virus collection agencies, only 93.7% of the virus engines agreed that the specimens were malicious. This indicates that AV software may provide less than ideal detection ability and supports the claims by other researchers (Mila Dalla et al., 2008; Szewczyk & Brand, 2008; W. Yan et al., 2008; Z. Yan & Inge, 2008; Zhou & Meador Inge, 2008).

The specimens were dominated by the *Allaple* worm at approximately 77.57% of the total number of specimens that could be analyzed. Approximately 18% of these specimens recorded no activity when run inside the sandbox *Anubis* provides, even though the average run time was 148 seconds. Although purely speculation at this point in time, this could indicate specimens that have detected the presence of *Anubis* or other analytical tools and used deception to not perform malicious activity to avoid being detected and remains to be investigated. These samples should be flagged for special consideration for determining if they were using anti online analysis techniques that have been documented in this research.

98.9% of the specimens indicated very high levels of entropy which means they were packed or protected. Packing and protecting is typically used by malware to mitigate detection by Anti Virus software. The two packer detectors did not agree on the names of any of the packer signatures they detected. *SigBuster* provided a name for 73.58% of the specimens and *PEiD* gave a name for 5.83% of the specimens. Knowledge of the name of the packer greatly assists malware analysis because the appropriate unpacking algorithm can be used to unpack the malware to arrive at the OEP. Using packers that are not recognised by packer detectors assists the malware from not being analyzed in detail and certainly implies that automated unpacking based on recognition of a name could produce a lot of false positives. Measurement of entropy appears to be a very successful method of detecting packed and protected malware as supported by other researchers (Ebringer & Sun, 2008; Lyda & Hamrock, 2007) .

6.3. RESEARCH QUESTION 1 - WHAT TECHNIQUES CAN MALWARE USE TO AVOID BEING ANALYZED?

This research question is essentially exploratory in nature. It was answered by

- Uncovering techniques from a review of the literature.
- Implementation of the techniques.
- Validation of the techniques through quasi experimentation.

The literature review uncovered an extensive range of techniques, mostly published by three key researchers (Falliere, 2007; Ferrie, 2008; Yason, 2007) who each provide their own, differing taxonomies of techniques. Note that these papers have only been published within the past year or two of this research and this could be indicative of the problems encountered by the increased spectrum of techniques malware can now employ to hinder analysis. Their work is supplemented by other researchers (Anthracene, 2006; Gordon, n.d.; Rolles, 2007; Smidgeonsoft, 2005; Smith & Quist, 2006; xC, 2007) whose online articles focus on more individual techniques and provide greater detail with respect to implementation and analysis. The work of Rolles in particular, focuses on leading edge techniques such as malware that uses its own virtual machines to avoid detailed analysis. Such malware is difficult to analyse because the custom virtual machines have their own instruction sets and these customised instruction sets have to be determined before detailed analysis can commence. A proposed taxonomy by the author of this research combines elements of the taxonomies of Falliere, Ferrie and Yason appears in Table 6-2, in an attempt to provide a more complete coverage of techniques. Note that each technique listed in the taxonomy is the highest level stratum and could be further stratified.

Table 6-2 Taxonomy of anti-analysis techniques

Technique	Description
Anti Emulation	A range of techniques exist to detect that the malware is running inside popular VM's such as `VMWare` or `Virtual PC`.
Anti Online Analysis	A variety of techniques exist for malware to determine if it is running in a online analysis engine such as `Anubis` or `Norman Sandbox`.
Anti Hardware	Techniques that target hardware such as the CPU including the debug registers.
Anti Debugger	Target the way Debuggers work and take advantage of these to take control of the flow of execution.
Anti Disassemblers	Target the way Disassemblers work and take advantage of this to produce a false disassembly.
Anti Tools	Detect the presence of specific analysis tools and enter a deceptive mode.
Anti Memory	Target the way memory is used when a process is being debugged and take advantage of this as well as the way processes can be dumped from memory including stolen bytes.
Anti Process	Target the way processes are handled when being debugged and take advantage of this including structured exception handling.
Anti-analysis	Target the way analysis is conducted. Use junk code, code camouflage, check sum checks, destruction of the Import Address Table and other deceptive techniques to make analysis harder.
Packers and Protectors	Use run time packers and protectors to obfuscate code and data and make it hard to unpack to find the original entry point. This includes packers that use their own virtual machines such as `HyperUnpackme2`.
Rootkits	Insert rootkits at `Ring 0` to take control of the way the operating system manages processes and use deception to hide malicious processes.

The existing literature only provided snippets of code and these snippets had to be implemented in standalone programs for the purpose of validation. Each technique that was validated was implemented in isolation to provide as much control as possible over the environment and written in assembly language. Assembly language was used because collected malware, such as that collected by the ECU honeypot (Valli & Wooten, 2007) are binaries and are in assembly language in their disassembled state. Validation was conducted by employing quasi experiments where the effects on common debuggers such as *IDA Pro* and *Ollydbg* were observed. All of the techniques that were implemented were determined to be valid and prevented analysis.

Malware is often packed or protected to hinder analysis by anti-virus software or static analysis. One of the first steps the malware analyst performs after detection of the virus signature is the detection of the packer used to pack the malware. Determination of the name of the packer allows the analyst to apply the appropriate algorithm to unpack the malware. Hundreds of different packers exist and range from using simple techniques through to very complex techniques that use Virtual Machines. Unpacking can be conducted by automated scripts or with manual methods to arrive at the OEP. Even simple packers can be customized by malware authors to disrupt automated scripts and hence hinder analysis. Additionally, the unpacking routines can contain the analysis avoidance techniques discussed and validated in this thesis. A common technique is to cause a divide by zero exception during the unpacking process to give control to the malware so that it can determine if it is running inside a debugger. If it detects it is running inside a debugger, the malware can take control and exit the program.

Malware can use anti-forensic techniques at any time and use deception to hide its real purpose. If it does not perform any malicious action while it is being analyzed, it may be accepted on the system as being safe. Then once free from analysis, it can perform its original, malicious objective.

6.4. RESEARCH QUESTION 2 – HOW CAN THE USE OF THESE TECHNIQUES BE DETECTED?

This research question was also mostly exploratory in nature, particularly with respect to how the technique can be detected. A number of the techniques that were discussed in the literature review were validated in small quasi-experiments where a single technique was implemented and its behaviour was observed and empirically recorded from conducting controlled experiments. Once the technique was validated, scripts were written or sourced particular to the two debuggers that were being used to detect the use of the technique. There are a variety of plugins for the popular debuggers whose purpose is to hide the debugger from malware that uses these techniques, but these plugins only provide a very small subset of anti-anti-forensic functionality and generally do not log the detection event. This necessitated the development of scripts that not only hide the debugger but also log the detection event. It was found that all of the techniques that were implemented could be detected using scripts or by manual methods. A very good source for discussing the development of these detection scripts for *IDA Pro* are discussed by Eagle (2008b), but the number and scope of the scripts is relatively small compared to the number of techniques revealed from the literature review. A much higher number of scripts are available from software reverse engineering sites such as the *Tuts4You* web site maintained by Rogers (2008). However, the scripts at such sites are written for debuggers such as *OllyDbg* and either have to be rewritten into *IDC* or *IDAPython* scripts for *IDA Pro*, or the analyst must be prepared to use multiple debugging tools and multiple scripting languages. It is therefore highly advisable for malware analysts to develop or source detection scripts and have a library of suitable scripts at their disposal. It is also advisable for malware analysts to develop or have access to malware analysts with scripting skills particular to popular debuggers.

It was noted that plugins for *OllyDbg* and *IDA Pro* such as *Olly Advanced* and *IDA Stealth* focus mostly on hiding the debuggers. This researcher recognizes three limitations of these plugins. The first is that the number of techniques that are currently mitigated by the plugins is limited. This is exemplified by the discrepancy in the large number of techniques uncovered in the literature review compared to the limited number of techniques available in the plugins. The second is that the techniques focus on hiding the debugger only. Other methods and approaches are required to cover the other techniques that are available as specified in the taxonomy in Table 6-2 above. This means that extensive knowledge of techniques and tools including acknowledgement of their limitations is required to mitigate the anti forensic techniques that malware has at its disposal to employ. The third limitation uncovered in this research is that the plugins do not provide notification through logs that particular techniques were detected. This limitation does not assist the collection of forensic evidence.

A review of the literature on malware analysis methodologies (Skoudis & Zeltser, 2004; Zeltser, 2007) found that the most effective methodologies take the presence of analysis avoidance techniques into account. Zelter's incremental, static and dynamic spiral approach for analyzing malware from a high level of detail down to a low level of detail provided an effective methodology to discover and mitigate analysis avoidance techniques as the analysis progresses. Zelter's methodology uses an iterative and recursive technique to traverse through the phases of static analysis, molding the environment for conitnued dynamic analysis. Zelter's methodology begins by performing a basic static analysis of the malware specimen such as performing a virus scan, determining the type of file and the type of packer used. This is followed by setting up a suitable environment to examine the specimen in, such as Windows XP in a Virtual Machine. This is followed by running the malware and observing its behaviour. The methodology continues to spiral in from obtaining information from a low level of detail, down to a highly detailed level. A graphical representation of Zelter's methodology is depicted in Figure 6-1. An extended model of Zelter's spiral analysis methodology is represented by Figure 6-2. The advantage of extending Zelter's spiral analysis methodology is that when anti forensic techniques are encountered, they can be detected and mitigated before proceeding with the analysis. This appears to be a far superior approach to that discussed by (Skoudis & Zeltser, 2004) who neglects to include a strategy for detecting and mitigating anti forensic techniques.

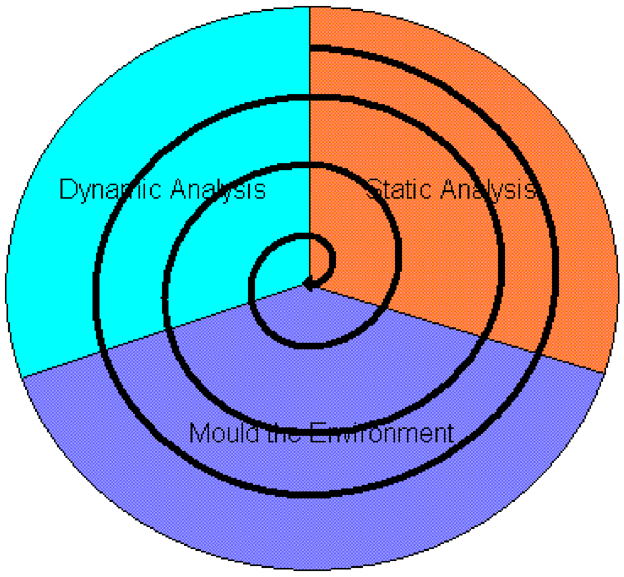


Figure 6-1 Graphical representation of Zelter's analysis methodology showing spiral nature through phases.

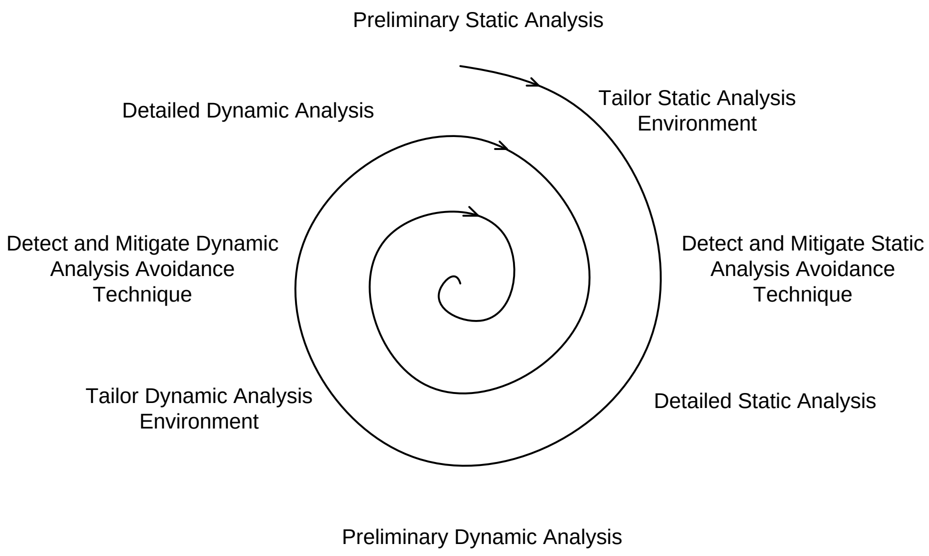

Preliminary Static Analysis
Tailor Static Analysis Environment
Detect and Mitigate Static Analysis Avoidance Technique
Detailed Static Analysis
Preliminary Dynamic Analysis
Tailor Dynamic Analysis Environment
Detect and Mitigate Dynamic Analysis Avoidance Technique
Detailed Dynamic Analysis

Figure 6-2 Extended analysis methodology to cater for anti-forensic techniques. Anti-analysis techniques are mitigated as they are detected.

The analysis that was conducted in this research showed that the measurement of the entropy of the malcode is very effective at detecting if a packer has been used. The analysis of the collected malware via two different, packer signature detectors also provided very different signature results. Generally, once the packer signature has been determined, the appropriate algorithm can be applied to unpack the malware to arrive at the OEP. However, if conflicting packer signatures are determined from two or more packer signature detectors, both algorithms may have to be applied to arrive at the OEP, and there is no guarantee that either one of them is correct. This has implications with respect to wasting the time of the analyst and certainly benefits the malware writer whose objective is to prevent or hinder analysis of the malcode.

6.5. RESEARCH QUESTION 3 – HOW CAN THE USE OF THESE TECHNIQUES BE MITIGATED?

This research question was also mostly exploratory in nature and answered through empirical results gained from controlled quasi-experiments. It was found through quasi experimentation of the techniques that were selected to be validated, that the techniques could be mitigated once they had been detected. However, although popular debugging tools such as *OllyDbg* and *IDA Pro* have plugins to help hide the debugger such as *Olly Advanced* and *IDA Stealth* respectively, their coverage of techniques is relatively limited given the much larger number of techniques that are available in contrast to the number of techniques covered by the plugins. Additionally, these plugins concentrate mostly in hiding the debugger leaving a considerable lack of overall mitigation coverage for the remainder of the techniques. This leaves considerable work to be done in providing mitigation coverage for the remaining techniques in tools and scripts.

6.6. LIMITATIONS OF THE STUDY

6.6.1. Methodology

Although the research questions were answered via the literature review, validated through quasi-experimentation and detection of the use of packers and protectors in collected malware, other research methods would be of assistance, particularly to assist in triangulation to gain an improved perspective of this phenomena. This could include a case study where observations are made of how malware analysts in the field detect and mitigate anti-forensic techniques. It could include survey research conducted via questionnaires and structured interviews of malware analysts in the field to find data to address a hypothesis such as “Is Malware increasingly using anti forensic techniques”. Conceivably, given the complexity of malware, teams of malware analysts have specialties and work together. A ethnography could be conducted to find meaning through field observation of malware analysts and how they work together and detect and mitigate anti-forensic techniques. Additionally, action research could be conducted to interact with malware analysts in the field to assist in improving the processes and methodologies associated with countering anti-forensic techniques in malware.

Although detection and mitigation scripts were validated against the implemented techniques, they were not used against the collected malware because of the restraints of the research questions and limitations of time. This remains as an activity to pursue.

6.7. DISCUSSION OF CONTRIBUTION TO KNOWLEDGE

This research claims to contribute to the body of knowledge associated with the anti-analysis techniques malware can incorporate to hinder forensic analysis. These claims are discussed in the following subsections.

6.7.1. Confirmation that anti-analysis techniques are very effective

This research shows that a variety of techniques are available to authors of malware to hinder the malware forensic analyst from fully discovering the capabilities of the malware. Malware can use these techniques to detect if it is being analyzed and can then use deception to hide its true intent (Brand, 2007; Eagle, 2008a; Falliere, 2007; Grugq, n.d.; Yason, 2007). This research shows how these techniques work, how the use of these techniques can be detected and how they can be mitigated. This line of research that combines these three aspects has not been located in existing research.

6.7.2. Anti-analysis techniques can be detected and mitigated

This research shows that the use of scripting for debuggers and disassemblers extends the functionality of the tools to facilitate the detection and mitigation of analysis avoidance techniques employed by malware. This research recommends that the development of debugger and disassembly scripting skills is a requisite to being able to detect and counter analysis avoidance techniques of malware. This contribution exists at the current front line of research in the detection of malware.

6.7.3. Confirmation that virus signature detection is less than ideal

An examination of a sample of the malware specimens collected for the purposes of this research shows that even though the majority of the malware collected had been "in the wild" for up to, or exceeding one year, the unanimous detection by a collection of thirty six AV detection engines was only 10.4%. The particular AV engine used by the *Anubis* (International Secure Systems Lab et al., 2008) online virus analyzer only recorded a 73.7% detection rate. This is a significant and potentially alarming result. It indicates that even though it is accepted computer security policy to run AV software, detection of all malware could be highly unlikely. This is supports the findings of other researchers (Masood, 2004; Mohandas, n.d.; Skoudis & Zeltser, 2004; Szewczyk & Brand, 2008).

6.7.4. Malware extensively uses Packers and Protectors

Runtime packers are utilized by network based malware to compress malware and to act as a counter measure to signature based AV software via obfuscation (Sun et al., 2008). The packed malware has to be unpacked to be able to perform a detailed static analysis because packed malware obfuscates the malware code. Knowledge of the packer used, assists in the process of unpacking because the appropriate unpacking methodology can be employed. Software tools are available that attempt to determine the name of the packer that was used to pack the malware. This research shows that two popular packer detectors that were used by this researcher did not agree on the names of any of the packers that were used. This is significant because it indicates uncertainty could be associated with the determined packer signatures and more in depth analysis is required to validate the type of packing that was conducted. The line of this research was extended to examine entropy (randomness) measurements of the packed malware as a method of determining if the collected malware was packed or not. Entropy measurements are shown in this research to be a very good indicator that malware has been packed.

6.7.5. Support for a new paradigm for malware detection

AV software typically uses signature matching and recognition of heuristics to detect malware. This approach generally requires the malware to have been collected “from the wild”, analyzed and signatures downloaded to client computers to approach any level of effectiveness. Significant damage to computers could occur between the time of collection and signature updates have been performed. In addition, it is very unlikely that AV software will detect custom malware that has not been set loose on the internet, but targeted against an individual or a corporation because it will not have been analyzed and a signature will not have been obtained by an AV company. AV software that uses this approach is seen to be fighting a losing battle in the literature and from this research (Mila Dalla et al., 2008; Zhou & Meador Inge, 2008). This research supports a proposal for a new paradigm for malware detection. In particular, this research proposes that detection of deception and anti-analysis techniques in software should flag the software as potentially malicious and delegate for further in depth analysis or removal.

6.7.6. Identification of analysis tool deficiencies

A number of software tools are utilized by malware forensic analysts. Static analysis and dynamic analysis are two methodologies that can be used to analyse the malware (Aquilina et al., 2008). Software disassemblers and debuggers such as *IDA Pro* (Hex-Rays, 2008) and *OllyDBg* (Yuschuk, 2008) can be used to perform a detailed analysis of the malware code and provide an internal view of the malwares functionality (Valli & Brand, 2008). This is referred to as static analysis. In contrast, dynamic analysis runs the malware and observes the interaction of the running malware with the computer from an external point of view. A number of plug-ins that extend the functionality of *IDA Pro* and *OllyDBg* include *IDA Stealth* (Newger, 2008) and *Olly Advanced* (MaRKuS, 2006) respectively to work with malicious code that employ anti-analysis techniques. The intention of such plug-ins is to provide functionality to hide their associated tools from the malware they are analyzing. The research in this thesis shows that the number of anti forensic techniques covered by such plug-ins is much less than the number of techniques that are available to be implemented by malware. In addition, this research shows that although the plug-ins successfully hides the debugger or disassembler, the tools do not provide any information to the analyst about having detected the use of analysis avoidance techniques. This is significant because detection of the use of anti-analysis techniques in software may be of assistance to a digital forensic investigator to show that deception was used to hide malicious intent.

6.7.7. Determination of suitable malware analysis methodology

Essentially, types of malware analysis fall under two main categories, dynamic analysis and static analysis. Dynamic analysis means the code is run and its behaviour and interaction with the computer it is running on, and the interaction with inter connected computers is observed. Static analysis means that the code is not run, but the code itself is analyzed to determine the functionality and capability of the code. Generally, dynamic analysis is easier to perform than static analysis but malware can more easily employ deception to hide its true intent without the analyst being aware of it. In reality, both types of analysis can be subverted. This research recommends that given the deceptive nature of malware, a combination of dynamic and static analysis is best performed in a sequential manner to mitigate analysis avoidance techniques. Fundamentally, this means that an initial high level static analysis of the malware is first performed. Using this information, a high level dynamic analysis is conducted using the information from the first static analysis to setup a suitable working environment. Information gathered from this phase is used as an input to conduct a more detailed static analysis, mitigating analysis avoidance counter measures in the malware. This process of dynamic analysis following static analysis is then followed, spiraling in from a high level of perspective until a low level of perspective of the malware is attained. This is very much along the lines recommended by Zeltser (2007), but explicitly adds the search for anti forensic techniques and subsequent mitigation as the analysis proceeds.

6.7.8. Development of a taxonomy of analysis avoidance techniques

This research amalgamates existing anti-analysis technique taxonomies into a single taxonomy as shown in Table 6-2 (Falliere, 2007; Ferrie, 2008; Yason, 2007). This is envisaged as being potentially very useful for classification purposes.

6.7.9. Malware Analysis Body of Knowledge

This research has shown that malware does make extensive use of packers and protectors to hinder analysis. This research has also shown that the recursive and iterative approach outlined by Zeltser (2007) to analyse malware is the most effective methodology to detect and mitigate anti-analysis techniques as they are uncovered to continue analysis. Combination of these two findings led to a proposed analysis process that incorporates a learning taxonomy and is reproduced from the paper by Valli *and Brand* (2008, p. 3) as Figure 6-3. Research remains to be done on developing the learning taxonomy that incorporates anti-analysis techniques into the malware analysis process. This research could possibly be continued with surveys, case studies and ethnographies with AV software company malware analysts and malware academic researchers. Nothing on this particular research front has been able to be ascertained from known, existing research. This line of research would also benefit from a study of learning taxonomies such as Bloom’s learning taxonomy which divides educational objectives into three domains, affective, cognitive and psychomotor (Anderson et al., 2001).

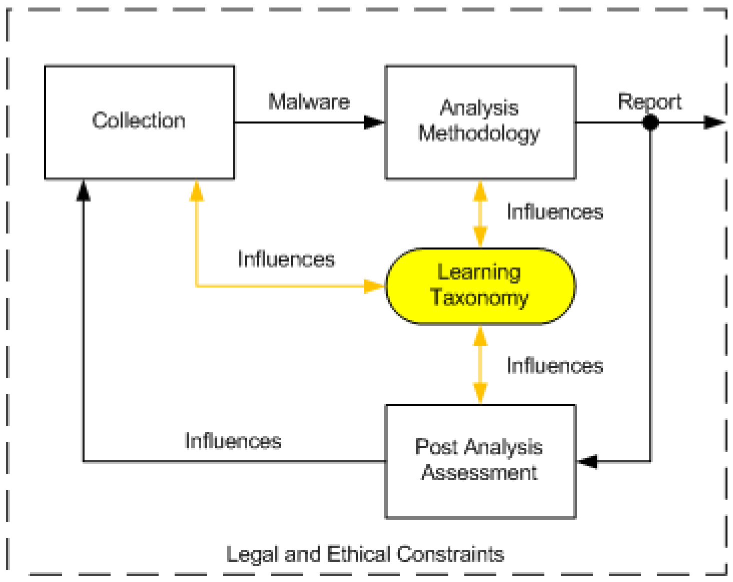


Figure 6-3 Malware analysis process incorporating a learning taxonomy that assists in the development of the MABOK.

The paper by Valli and Brand (2008) identified a Malware Analysis Body of Knowledge (MABOK) that could “be used as a framework for competency development and assessment for the field of malware analysis” (Valli & Brand, 2008, p. 2). Essentially this is because malware analysis is recognised to be difficult and a very broad knowledge domain is required to undertake detailed, in-depth analysis of malware. A knowledge domain identified by Valli and Brand (2008, p. 4) essentially from the research conducted for this thesis, is reproduced as Figure 6-4. Essentially, the diagram shows eight, high level categories of knowledge that are required to undertake malware analysis. The next lower stratum identifies numerous sub-domains of knowledge that could also be broken down into even more sub-domains.

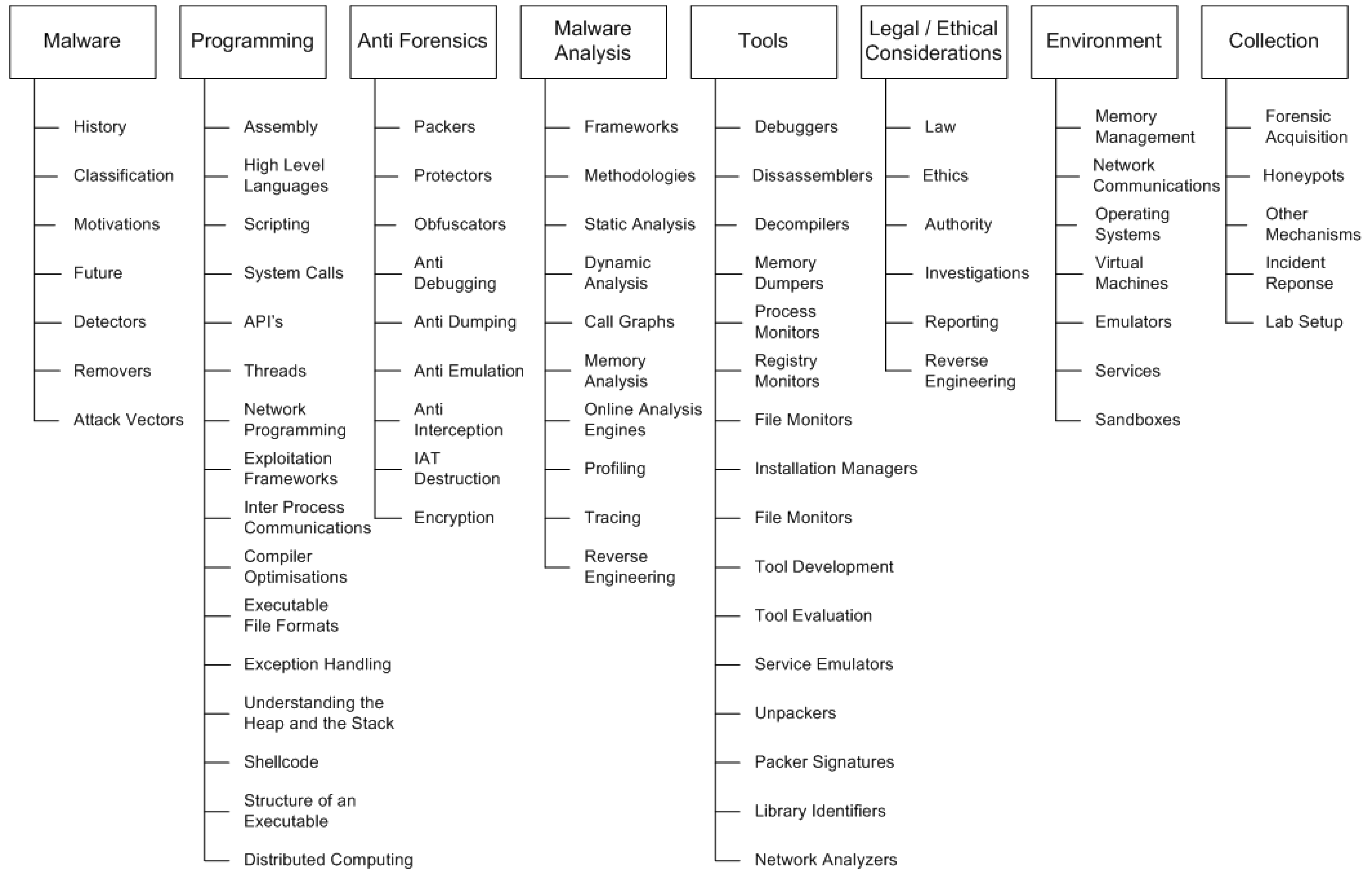


Figure 6-4 Model of the learning domain of the Malware Analysis Body of Knowledge (MABOK)

6.8. FUTURE RESEARCH

The lines of enquiry examined in this research could be extended in a number of avenues, as outlined in the following sub sections.

6.8.1. Hypothesis

Future research could include addressing a hypothesis such as:

- *Network based malware is increasingly using anti forensic techniques*.

This could be conducted by examining the network based malware collected by the ECU *Nepenthes* honeypot using the analysis avoidance detection and mitigation scripts presented in this thesis using a positivist, empirical, quasi experimental research methodology as outlined in this thesis.

6.8.2. Plugin Development

This research noted that plugins such as *IDAStealth* and *OllyAdvanced* provide coverage for only a subset of analysis avoidance techniques Additional research could be conducted on extending the coverage of techniques of such plugins. A limitation of the existing plugins is that their focus is on hiding the debugger and do not have the ability to detect and log the use of anti-analysis techniques. The detection and logging of techniques as they are discovered during forensic analysis of malware could assist in the collection of evidence suitable for a court of law.

6.8.3. Collation of Techniques

This research revealed an extensive range of analysis avoidance techniques that is distributed amongst research papers, hacking and reverse engineering sites. Detection and mitigation techniques are not represented any where near the same extent in academic literature or on hacking and reverse engineering sites. A very useful contribution to the field of malware analysis research could be to collate analysis avoidance techniques together with their corresponding detection and mitigation techniques into a central library and to develop an encompassing taxonomy.

6.8.4. Improved Packer Signature Detection

Packer signature detection has been revealed in this research to be an area that requires further and most likely, continual research. This also extends to the area of unpacking packed malware as well. This is because malware can use multiple packers not only from a sequential sense, for example, pack the entire malware specimen with packer A and then pack the result with packer B, but firstly pack sections of code with packer A and then pack the result with packer B. This last scenario is another deception trick that is generally only uncovered once manual analysis is conducted. It is possible that an automated analysis process may miss the second (or third, or more) level of packing. This remains an area of research that lacks published work.

6.8.5. A New Paradigm for Malware Detection

This research has shown that AV software to be less than fully effective at detecting malware. Research could continue into investigating a new paradigm for malware detection, particularly by detecting the use of anti-analysis techniques in scanned software and flagging it for more detailed attention.

6.8.6. A Model for Automating the Spiral Analysis Methodology

The spiral analysis methodology depicted in Figure 6-2 was proposed as a suitable process to follow to detect and mitigate anti-forensic techniques employed by malware in a very manual, labor intensive manner. This same methodology is presented in Figure 6-5 in the form of a process diagram that could be implemented in software to more automate the malware analysis process where anti-forensic techniques need to be detected and mitigated. It shows malware under investigation as the input to the process that employs the spiral analysis methodology. A central control supervisor processor is responsible for managing each step and phase of the analysis, where recording, processing and reporting is managed or delegated to a sub process. The supervisor function interacts with each phase by providing control over the constituent steps in each phase. It also acts as the recipient of data which is produced by each phase which is required to make decisions on how to tailor the subsequent phases. In addition to assisting the forensic analyst, such a process could be a supplementary tool, or a replacement, for traditional signature and heuristic based anti-virus software. This is because detection of the use of deception techniques could be a very good indicator of malicious intent as argued by this research. Continuation of this line of research into automating the analysis process is left to be researched.

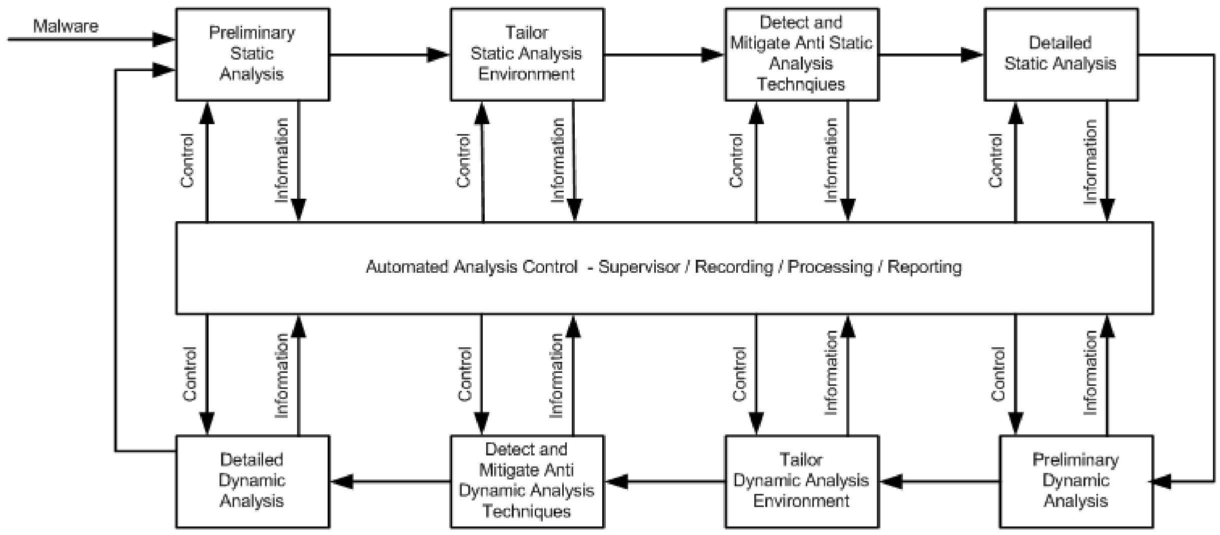

Malware
Preliminary Static Analysis
Tailor Static Analysis Environment
Detect and Mitigate Anti Static Analysis Technqiues
Detailed Static Analysis
Control
Information
Automated Analysis Control - Supervisor / Recording / Processing / Reporting
Detailed Dynamic Analysis
Detect and Mitigate Anti Dynamic Analysis Techniques
Tailor Dynamic Analysis Environment
Preliminary Dynamic Analysis

Figure 6-5 Proposed process model to automate the spiral analysis methodology which recursively and iteratively detects and mitigates static and dynamic anti-analysis techniques

CHAPTER 7 CONCLUSION

7.1. ANALYSIS AVOIDANCE TECHNIQUES OF MALWARE

AV software generally employs heuristics and signature matching to detect the presence of malware. Determination of the signatures and heuristics of malware is performed by analysts and sent out in updates to the signature files anti-virus software depends on to detect its presence. It is not uncommon for these updates to be conducted multiple times per day because of the large number of new malicious threats that appear each day on the internet. AV software has been shown in this research to be less than fully effective and this supports the claims of other AV researchers. Malware can employ a variety of techniques to avoid detection by anti-virus software and hinder the analysis conducted by analysts. This is because malware is becoming increasingly profit driven and more likely to incorporate stealth and deception techniques to avoid detection.

Malware has an extensive range of anti-forensic techniques that it can incorporate into its overall functionality to hinder analysis. This can include, but is not limited to the following taxonomy of techniques:

- Anti emulation
- Anti online analysis
- Anti hardware
- Anti debugger
- Anti disassembler
- Anti tools
- Anti memory
- Anti process
- Anti-analysis
- Packers and Protectors
- Rootkits

The overall aim of malware that incorporates these techniques is to defeat the signature and heuristic based nature of anti-virus software and to hinder the forensic analyst by making detailed analysis time consuming and difficult. This research has validated a number of these techniques and all proved to be effective. This research has also shown that these techniques can be detected and their use mitigated so that detailed forensic analysis can be conducted. However, it remains a time consuming activity, based on methodology and analysis that requires a very broad range of knowledge and a significant skill set. Competence with scripting languages associated with the popular debuggers is a requisite to being able to detect and mitigate these techniques, particularly when new techniques arise. This is because the coverage of the techniques in existing plugins and scripts is limited. Plugins tend to concentrate on hiding the debugger, or mitigate only a small number of the anti-analysis techniques that are available. This is identified as a limitation analysts must be aware of. Existing analysis scripts for some tools are more prevalent than for other tools. In either case, the forensic analyst will need the ability to create or modify existing scripts to conform to the requirements of the tools that the forensic analyst has validated as forensically sound to employ.

7.2. CONTRIBUTION TO KNOWLEDGE

This research contributes to the body of knowledge directly related to the anti-analysis techniques malware incorporates into its code, from a variety of perspectives, as outlined in the following sub sections.

7.2.1. Confirmation that anti-analysis techniques are very effective

This research shows that a variety of techniques are available to authors of malware to hinder the malware forensic analyst from fully discovering the capabilities of the malware.

7.2.2. Anti-analysis techniques can be detected and mitigated

This research shows that the use of scripting for debuggers and disassemblers extends the functionality of the tools to facilitate the detection and mitigation of analysis avoidance techniques employed by malware.

7.2.3. Confirmation that virus signature detection is less than ideal

An examination of a sample of the malware specimens collected for the purposes of this research shows that even though the majority of the malware collected had been “in the wild” for up to, or exceeding one year, the unanimous detection by a collection of thirty six AV detection engines was only 10.4%.

7.2.4. Malware extensively uses Packers and Protectors

Runtime packers are utilized by network based malware to compress malware and to act as a counter measure to signature based AV software via obfuscation (Sun et al., 2008). Entropy measurements are shown in this research to be a very good indicator that malware has been packed.

7.2.5. Support for a new paradigm for malware detection

This research supports a proposal for a new paradigm for malware detection. In particular, this research proposes that detection of deception and anti-analysis techniques in software should flag the software as potentially malicious and delegate for further in depth analysis or removal.

7.2.6. Identification of a Malware Body of Knowledge

The knowledge required to analyse malware is extensive. A Malware Analysis Body of Knowledge (MABOK) has been identified from the conduct of this research, to include anti-forensics as a very significant component.

7.2.7. Identification of analysis tool deficiencies

The research in this thesis shows that the number of anti forensic techniques covered by such plug-ins is much less than the number of techniques that are available to be implemented by malware. In addition, this research shows that although the plug-ins successfully hides the debugger or disassembler, the tools do not provide any information to the analyst about having detected the use of analysis avoidance techniques. This is significant because detection of the use of anti-analysis techniques in software may be of assistance to a digital forensic investigator to show that deception was used to hide malicious intent.

7.2.8. Determination of a suitable malware analysis methodology

This research outlines a suitable methodology for analyzing malware that incorporates anti-analysis techniques.

7.2.9. Development of a taxonomy of analysis avoidance techniques

This research amalgamates existing anti-analysis technique taxonomies into a single taxonomy.

7.3. LINKING OF CONTRIBUTIONS TO KNOWLEDGE

Malware can use anti-forensic techniques and use deception to hide its real purpose whilst being analyzed. If it does not perform any malicious action while it is being analyzed, it may be accepted on the system as being safe, or excluded from the evidence collection process. Then once free from analysis, the malware can perform its original, malicious objective. Some considerations must be made in order to closely analyze malware. Firstly, totally relying on AV software to classify the malware could be a mistake because signature based detection is far less than ideal. It is unlikely to recognize customized malware that has not been analyzed before. This leads to necessity of the digital forensic analyst to analyze the malware manually. It must be noted that a significant number of anti-analysis techniques exist covering the entire spectrum of the computational mechanics of computers. These techniques are very effective at hindering analysis. This can be compounded by additional factors. This includes deficiencies in analysis tools that do not cover the number of anti-analysis techniques that are available. It is made more difficult by the number of packers and protectors that malware can use. This makes it hard because a typical technique to unpack the malware is to use known algorithms to let the malware unpack itself to reach the OEP. In doing so, control is given to the malware and an opportunity exists for the malware to detect that is being analyzed and to employ deception. An additional consideration is that a very extensive knowledge of programming, debugging and operating system internals is required that arguably exceeds the level attained even by competent software engineers. On the positive side, the use of anti-analysis techniques can be detected and mitigated, given significant analysis skills have been attained. This can be assisted by using an appropriate methodology where static and dynamic methods are combined in such a way that the view of the malware transitions from a high level of detail down to a low level of detail, mitigating the anti-analysis techniques as analysis progresses in a spiral analysis methodology. Although legitimate software uses anti-analysis techniques to protect itself from reverse engineers, malware is almost certain to use anti-analysis techniques. So much so, the detection of the use of anti-analysis techniques may be a very good indicator of the presence of malware.

7.4. LESSONS LEARNED FROM RESEARCH APPROACH AND CONDUCT

The selected research method to address the research questions was positivist, empirical and quasi experimental. The research questions were essentially exploratory in nature. Validation of the techniques, followed by their detection and mitigation, was conducted in a series of controlled quasi-experiments. This effectively answered the research questions. Other empirical methods such as action research, ethnography, survey and case study could have been used, but would have required access to malware researchers desirably working in AV software laboratories for an extended period of time, and preferably, in situ. Such access is not possible for this researcher at this time. A combination of these methods would not have necessarily enhanced the validity of the results but would have undoubtedly contributed to answering the research questions. Triangulation would have been assisted by using additional tools to validate the results as would have using multiple analysts to perform the quasi experiments.

7.5. RESEARCH IMPLICATIONS

A number of significant implications have arisen from this research. A large number of anti-analysis techniques were uncovered and found to be very effective when implemented in small stand alone programs. These same techniques could be detected and mitigated by the development of scripts and plugins. Existing analysis tools serve primarily to hide the tools from being counter detected by the malware and cover a small minority of the available techniques malware can use to hinder analysis. These tools do not provide functionality to log or record detection of analysis avoidance techniques. Logging or recording of these techniques may be of great use to the digital forensic investigator when analyzing malware whilst investigating a case. Functionality can be added to existing tools by custom development of scripts and plugins. Knowledge of analysis avoidance techniques and being able to script and develop plugins adds to a body of knowledge, the MABOK, identified by this research. The MABOK covers the knowledge domain required to analyse malware and will be useful for assessment and skill development for analysts working with malware. In addition, this research shows an appropriate methodology should be employed by the forensic analyst to detect and mitigate these anti-analysis techniques as analysis continues.

This research supports claims that AV software performs at a less than ideal level and that a new paradigm is warranted. This research recommends that any software that employs anti-analysis techniques be treated as suspicious. This is because a characteristic employed by nearly all malware examined in this research employed anti forensic techniques, primarily packers and protectors.

Deficiencies in existing tools and plugins were found in the tools used for this research with respect to handling anti forensic techniques. This exemplifies the need for analysts to be able to conduct manual analysis and to not rely on automated tools. In addition, this emphasizes the importance of possessing the ability to be able to extend the functionality of the tools on an as required basis.

This research can be continued on a number of fronts. Firstly, it could continue the search for anti forensic techniques employed by the malware that was collected for the purposes of this research. Such a line of enquiry could use the existing detection and mitigation scripts as a foundation and continue in the development and use against the collected malware. This work could use a hypothesis such as “malware is increasingly using anti forensic techniques” and show the use of the techniques over time for collected malware.

Another line of enquiry would be to use the detection of anti forensic techniques as a new paradigm for AV software. This would very much suit the application of the true experiment research methodology.

REFERENCES

+Pumpqara. (n.d.). My Own Tricks to Detect OllyDbg. Retrieved Feb 10, 2008, from http://pe-lib.sourceforge.net/pumqara/html/code_protection/Detecting%20OllyDbg/DetectOllyDbg+Pum.htm

ANAKiN. (2005). PE-Pack.

Anderson, L., Krathwohl, D., Airasian, P., Cruikshank, K., Mayer, R., Pintrich, P., et al. (2001). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. New York: Longman.

Anonymous. (1999). PrettyPark.

Anonymous. (2003). SpyBot.

Anonymous. (2004). PolyBot.

Anonymous. (2005). GpCoder.

Anonymous. (n.d.-a). Allaple.

Anonymous. (n.d.-b). Bagle.

Anonymous. (n.d.-c). exetk.

Anonymous. (n.d.-d). ExtremBot.

Anonymous. (n.d.-e). ForBot.

Anonymous. (n.d.-f). HideWindow.

Anonymous. (n.d.-g). HyperUnpackMe2.

Anonymous. (n.d.-h). Kuang.

Anonymous. (n.d.-i). MLSRH.

Anonymous. (n.d.-j). Morphine.

Anonymous. (n.d.-k). MyDoom.

Anonymous. (n.d.-l). RBot.

Anonymous. (n.d.-m). Tibs.

Anthracene. (2006). Unpacking with Anthracene. Retrieved August 21, 2007, from http://www.tuts4you.com/download.php?list.18

ap0x. (2006). OpenRCE Anti Reverse Engineering Techniques Database. Retrieved Feb 08, 2008 from https://www.openrce.org/reference_library/anti_reversing

Aquilina, J., Casey, E., & Malin, C. (2008). *Malware Forensics Investigating and Analyzing Malicious Code*. Burlington, MA: Syngress.

Arnold, B., Chess, D., Morar, J., Segal, A., & Swimmer, M. (2000). An Environment for Controlled Worm Replication and Analysis. Retrieved March 18, 2007 from http://www.research.ibm.com/antivirus/SciPapers/VB2000INW.htm

ASPack Software. (2008). ASProtect.

Aycock, J. (2006). *Computer Viruses and Malware*. New York: Springer

Bart, & Xtreeme. (2005). FSG.

Bilar, D. (2005). Statistical Structures: Fingerprinting Malware for Classification and Analysis. Retrieved September 2, 2006 from www.blackhat.com/presentations/bh-usa-06/BH-US-06-Bilar.pdf

Bitsum Technologies. (2008). PECompact.

Blinkinc. (2003). Shrinker.

BoB. (2007). HideDebug.

Bouma, G., & Ling, R. (2004). *The Research Process, Fifth Edition*. Oxford UK: Oxford University Press.

Brand, M. (2007). *Forensic Analysis Avoidance Techniques of Malware.* Paper presented at the 5th Australian Digital Forensics Conference, Edith Cowan University, Mount Lawley Campus, Western Australia.

Canavan, J. (2005). The Evolution of Malicious IRC Bots. Retrieved Dec 28, 2007, from www.symantec.com/avcenter/reference/the.evolution.of.malicious.irc.bots.pdf

Carroll, J. M., & Swatman, P. A. (2000). *Structured-case: A methodological framework for building theory in information systems research*. Paper presented at the European Conference on Information Systems.

CG SoftLabs. (2008). Stud_PE.

Chess, B., & West, J. (2007). *Secure Programming with Static Analysis*. Upper Saddle River, NJ: Addision-Wesley.

Chouchane, M., Walenstein, A., & Lakhotia, A. (2007). *Statistical signatures for fast filtering of instruction-substituting metamorphic malware*. Paper presented at the Proceedings of the 2007 ACM workshop on Recurring malcode.

Collberg, C., Thomborson, C., & Low, D. (1998). *Manufacturing cheap, resilient, and stealthy opaque constructs.* Paper presented at the Annual Symposium on Principles of Programming Languages, San Diego, California, United States

Combs, G. (2008). Wireshark.

Compuware. (2008). SoftIce.

Craig, P. (2006). Unpacking Malware, Trojans and Worms. from www.security-assessment.com/files/presentations/Ruxcon_2006_-_Unpacking_Virus, _Trojans_and_Worms.pdf

Diabl0. (2005). MyTob.

Dunham, K. (2006). Criminalization of Code. from http://www.nortel.com/corporate/events/2006a/collateral/battling_malware_01_18_06/criminalization.pdf

Eagle, C. (2008a). *The IDA Book*: No Starch Press.

Eagle, C. (2008b). The IDA Book Examples. from http://www.idabook.com/examples/

Eagle, C. (n.d.). Honeynet Scan of the Month 32 Analysis. from http://honeynet.org/scans/scan32/sols/1-Chris_Eagle/analysis.html

Easterbrook, S., Singer, J., Storey, M., & Damian, D. (2008). Selecting Empirical Methods for Software Engineering Research. In D. Sjoberg, F. Shull & J. Singer (Eds.), *Guide to Advanced Empirical Software Engineering*: Springer.

Ebringer, T., & Sun, L. (2008). *A Fast Randomness Test that Preserves Local Detail*. Paper presented at the Virus Bulletin 2008.

Eilam, E. (2005). *Reversing : Secrets of Reverse Engineering*. Indianapolis: Wiley Publishing, Inc.

Erdélyi, G. (2008). IDA Python.

Falliere, N. (2006). Anatomy of a Malware. Retrieved October 20, 2007, from http://www.securityfocus.com/infocus/1893

Falliere, N. (2007). Windows Anti-Debug Reference. Retrieved October 1, 2007 from http://www.securityfocus.com/infocus/1893

Farwell, J. (2004). The Heart of the Matter, What Makes Antivirus Software Tick? Retrieved March 17, 2007, from http://www.smartcomputing.com/editorial/article.asp?article=articles/2004/s1511/24s11/24s11.asp&articleid=23737&guid=

Ferrie, P. (2008). *Anti-Unpacker Tricks*. Paper presented at the 2nd International Caro Workshop. from http://www.datasecurity-event.com/uploads/unpackers.pdf

FireDaemon Technologies Ltd. (2009). FireDaemon.

Foundstone. (2008). Fport.

G-RoM, Lorian, & Stone. (1999). ProcDump.

Gembe, A. (2002). AgoBot.

Gordon, J. (n.d.). Win32 Exception handling for assembler programmers. Retrieved Feb 10, 2008 from http://win32assembly.online.fr/Exceptionhandling.html

Grugq. (n.d.). The Art of Defiling, Defeating Forensic Analysis on UNIX File Systems.

Guba, E., & Lincoln, Y. (1994). Competing Paradigms in Qualitative Reseach. In N. Denzin & Y. Lincoln (Eds.), *Handbook of Qualitative Research*. London: SAGE Publications Ltd.

Harbour, N. (2007). Stealth Secrets of the Malware Ninjas. Retrieved October 20, 2007 from https://www.blackhat.com/presentations/bh-usa-07/Harbour/Presentation/bh-usa-07-harbour.pdf

Hernon, P. (1991). The Elusive Nature of Research in LIS. In C. McClure & P. Hernon (Eds.), *Library and Information Science Research: Perspectives and Strategies for Improvement* (pp. 3-14). Norwood, NJ: Ablex Publishing.

Hex-Rays. (2008). IDA Pro.

Hoglund, G., & Butler, J. (2005). *Rootkits: Subverting the Windows Kernel*. Upper Saddle River, NJ: Addison Wesley Professional.

Holt, T. (2007). The Market for Malware. Retrieved May 04, 2008, from http://www.dc414.org/download/confs/defcon15/Speakers/Holt/Presentation/dc-15-holt.pdf

Immunity. (2008). Immunity Debugger.

Innes, S., & Valli, C. (2006). *Honeypots: How do you know when you are inside one?* Paper presented at the 4th Australian Digital Forensics Conference, Edith Cowan University, Perth, Western Australia.

Insecure.org. (2008). nmap.

International Secure Systems Lab, Vienna University of Technology, Eurecom France, & UC Santa Barbara. (2008). Anubis: Analyzing Unknown Binaries. Retrieved October 4, 2008, from http://anubis.iseclab.org/

iroffer.org. (n.d.). IrOffer.

Jibz, Qwerton, Snaker, & XineohP. (2006). PEiD.

Kaspersky Labs. (2007a). Network Worms. Retrieved October 7, 2007 from http://www.viruslist.com/en/virusesdescribed?chapter=152540408

Kaspersky Labs. (2007b). Trojan Programs. Retrieved October 7, 2007 from http://www.viruslist.com/en/virusesdescribed?chapter=152540408

Kessler, G. (2007). Anti-Forensics and the Digital Investigator. Retrieved May 04, 2008, from http://scissec.scis.ecu.edu.au/conference_proceedings/2007/forensics/01_Kessler_Anti-Forensics.pdf

Kleiman, D. (2007). *The Official CHFI Study Guide (Exam 312-49) for Computer Hacking Forensic Investigators*. Burlington, MA: Syngress Publishing Inc.
Kotadia, M. (2006). Beware 'suicidal' malware, says CyberTrust. Retrieved August 27, 2006 from http://software.silicon.com/malware/0,3800003100,39160966,00.htm
Koziol, D., Litchfield, D., Aitel, D., Anley, C., Eren, S., Mehta, N., et al. (2004). *The Shellcoder's Handbook*. Indianapolis: Wiley Publishing Inc.
Krack. (2006). Defeating Norman Sandbox. Retrieved July 21, 2006 from http://www.ryan1918.com/viewtopic.php?t=2676&highlight=defeat
Larsson, L. (2007). Meeting the Swedish Bank Hacker. Retrieved April 14, 2007 from http://computersweden.idg.se/2.2683/1.93344
Lau, B., & Svajcer, V. (2008). Measuring Virtual Machine Detection in Malware Using DSD Tracer. *Journal in Computer Virology*.
Lee, T., & Mody, J. (2006). Behavioural Classification. Retrieved 16 March, 2006, from secureitalliance.org/blogs/microsoft/attachment/1244.ashx
Lyda, R., & Hamrock, J. (2007). Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security and Privacy, 5*(2), 40-45.
MackT. (2008). Import REConstructor.
Mandiant. (2007). Red Curtain. Retrieved October 20, 2007, from http://www.mandiant.com/mrc
Mardam-Bey, K. (1995). mIRC.
MaRKuS. (2006). Olly Advanced.
Marshall, P. (1997). *Research Methods*. Plymouth, United Kingdom: How To Books.
Masood, S. G. (2004). Malware Analysis for Administrators. Retrieved 17 March, 2007 from http://www.securityfocus.com/infocus/1780
Microsoft. (2007). Virtual PC.
Microsoft. (2008a). Explorer.exe.
Microsoft. (2008b). windbg.
Microsoft. (2008c). Windows Sysinternals.
Mila Dalla, P., Mihai, C., Somesh, J., & Saumya, D. (2008). A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst., 30*(5), 1-54.
Mohandas, R. (n.d.). Hacking the Malware – A reverse-engineer's analysis. Retrieved 17 March 2007, from geocities.com/rahulmohandas/hacking_the_malware.pdf
Nepenthes. (2006). Nepenthes – Finest Collection. Retrieved March 16, 2004 from http://nepenthes.mwcollect.org
Newger, J. (2008). IDA Stealth Plugin.
Newman, R. (2006). *Cybercrime, identity theft, and fraud: practicing safe internet - network security threats and vulnerabilities*. Paper presented at the Proceedings of the 3rd annual conference on Information security curriculum development.
NIST. (2004). Computer Security Incident Handling Guide. Retrieved 15 Sept 2005, from http://csrc.nist.gov/publications/nistpubs/800-61/sp800-61.pdf
Norman. (2008). Submit file for Sandbox analysis. Retrieved April 12, 2008, from http://www.norman.com/microsites/nsic/Submit/en-us

Northfox. (2004). MEW.
Oates, B. (2007). *Researching Information Systems and Computing*. London: Sage Publications Ltd.
Oberhumer, M., Molnár, L., & Reiser, J. (2008). UPX.
OG. (2007). Defeating Anubis File Analyzer. Retrieved Jul 21, 2007 from http://www.ryan1918.com/viewtopic.php?t=8654
Oreans Technologies. (2008). Themida.
Perry, D., Porter, A., & Votta, L. (2000). *Empirical studies of software engineering: a roadmap.* Paper presented at the International Conference on Software Engineering Limerick, Ireland.
Pickard, A. (2007). *Research Methods in Information*. London: Facet Publishing.
Pietrek, M. (n.d.). PEdump.
Porras, P., Saidi, H., & Yegneswaran, V. (2007). A Multi-perspective Analysis of the Storm (Peacomm) Worm. Retrieved Dec 7, 2007 from http://www.cyber-ta.org/pubs/StormWorm/SRITechnical-Report-10-01-Storm-Analysis.pdf
random, killa, & acpizer. (1999). PECRYPT32.
Reversing Labs. (2008). RLPack.
Rogers, M. (2006). Panel session at CERIAS 2006 Information Security Symposium. *Journal*. Retrieved from http://www.cerias.purdue.edu/symposium/2006/materials/pdfs/antiforensics.pdf
Rogers, T. (2008). Tuts4You. from www.tuts4you.com
Rolles, R. (2007). Defeating HyperUnpackMe2 With an IDA Processor Module. Retrieved Feb 28, 2008 from http://www.openrce.org/articles/full_view/28
Rubenking, N. J. (2007). Jaaaane! Get Me off This Crazy Thing! Retrieved March 26, 2007 from http://0-proquest.umi.com.library.ecu.edu.au/pqdweb?did=1204578291&sid=2&Fmt=3&clientId=7582&RQT=309&VName=PQD
Sandboxie. (2008). About Sandboxie. Retrieved April 12, 2008, from http://www.sandboxie.com/
Schiller, C., Binkley, J., Harley, D., Evron, G., Bradley, T., Willems, C., et al. (2007). *Botnets : The Killer Web App*. Rockland: Syngress Publishing Inc.
Schwittay, B. (2006). Towards Automating Analysis in Computer Forensics. Retrieved Dec 05, 2007 from http://pi1.informatik.uni-mannheim.de/filepool/theses/diplomarbeit-2006-schwittay.pdf
sd. (2002). SDBot.
Seculab. (2008). neoGuard.
SHaG. (2006). OllyScript.
Shub-Nigurrath. (2006). HideDebugger.
Silicon Realms. (2008). Armadillo.
Sjoberg, D., Dyba, T., & Jorgensen, M. (2007). *The Future of Empirical Methods in Software Engineering Research.* Paper presented at the International Conference on Software Engineering.
Skoudis, E., & Zeltser, L. (2004). *Malware Fighting Malicious Code*. New Jersey: Prentice Hall.
Smidgeonsoft. (2005). SetUnhandledExceptionFilterTrick. Retrieved Feb 11, 2008, from http://www.openrce.org/forums/posts/45
Smith, S., & Quist, D. (2006). Hacking Malware: Offense is the new Defense. Retrieved July 24, 2007

from http://www.offensivecomputing.net/dc14/valsmith__dquist_hacking_malware_us06.pdf
Sofpro. (2008). PC Guard.
Sourcefire. (2008). snort.
Stargazer. (2006). Anti-Sandbox code with norman. Retrieved Apr 12, 2008, from http://my.stargazer.at/2006/11/07/anti-sandbox-code-anhand-von-norman/
Stewart, J. (2006). OllyPerl.
Sub7Crew. (n.d.). SubSeven.
Sukhai, N. (2004). *Hacking and cybercrime*. Paper presented at the Proceedings of the 1st annual conference on Information security curriculum development.
Sun, L., Ebringer, T., & Boztas, S. (2008). Hump-and-Dump: efficient generic unpacking using an ordered address execution histogram. *Journal*. Retrieved from http://www.datasecurity-event.com/uploads/hump_dump.pdf
Szewczyk, P., & Brand, M. (2008). *Malware Detection and Removal: An Examination of Personal Anti-Virus Software.* Paper presented at the 6th Australian Digital Forensics Conference, Edith Cowan University, Mount Lawley Campus, Western Australia.
Team Cymru. (2006). Cybercrime: an epidemic. *Queue, 4*(9), 24-35.
Tenable Network Security. (2008). nessus.
TGM. (2004). Telock.
TH-DJM, M. (2006). Olly Advanced.
Thrasher. (2007). Anti Sandboxie. Retrieved July 21, 2007 from http://www.ryan1918.com/viewtopic.php?t=11045
Trend Micro Incorporated (2007). Trend Micro Annual Threat Report: Cybercriminals are Working Faster Than Ever. *Journal*. Retrieved from http://trendmicro.mediaroom.com/index.php?s=43&item=700
Valli, C., & Brand, M. (2008). *Malware Analysis Body of Knowledge.* Paper presented at the 6th Australian Digital Forensics Conference, Edith Cowan University, Mount Lawley Campus, Western Australia.
Valli, C., & Wooten, A. (2007). An Overview of ADSL Homed Nepenthes Honeypots In Western Australia. *Proceedings of The 5th Australian Digital Forensics Conference*, 204-209.
Vilhonen, V. (2007). OllyPython.
Virus Bulletin (2008). Blended Threat. *Journal*. Retrieved from http://www.virusbtn.com/resources/glossary/blended_threat.xml
Virus Total. (2007). Virus Total. Retrieved March 17, 2006 from http://www.virustotal.com/en/virustotalf.html
Virus Total. (2008). Virus Total. Retrieved October 4, 2008, from http://www.virustotal.com/en/virustotalf.html
VirusList.com (2009). Trojan Programs. *Journal*. Retrieved from http://www.viruslist.com/en/virusesdescribed?chapter=152540521#arch
VMProtect. (2008). VMProtect.
VMware. (2008). VMware.
Websense (2008). Analysis of Recent Storm Worm Packer. *Journal*. Retrieved from http://securitylabs.websense.com/content/Blogs/3083.aspx
Williamson, K. (2002). *Research methods for students, academics and professionals, Second Edition*. Wagga Wagga, NSW: Print Quick.

Winalysis.com. (2008). WinAlysis.

Wysopal, C. (2009). Good Obfuscation, Bad Code. Retrieved May 03 2009, from http://www.securityfocus.com/columnists/498?ref=oc

xC. (2007). Defeating Anubis File Analyzer. Retrieved Jul 21, 2007 from http://www.ryan1918.com/viewtopic.php?p=68714&sid=354448fa02136b766d94dfcea11b4e2d

Xuxian, J., Xinyuan, W., & Dongyan, X. (2007). *Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction*. Paper presented at the Proceedings of the 14th ACM conference on Computer and communications security.

Yan, W., Zhang, Z., & Ansari, N. (2008). Revealing Packed Malware. *IEEE Security and Privacy 6 (5)*, 65-69.

Yan, Z., & Inge, W. M. (2008). *Malware detection using adaptive data compression*. Paper presented at the Proceedings of the 1st ACM workshop on Workshop on AISec.

Yason, M. (2007). The Art of Unpacking. Retrieved Feb 12, 2008 from https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf

Yin, H., Song, D., Egele, M., Kruegel, C., & Kirda, E. (2007). *Panorama: capturing system-wide information flow for malware detection and analysis*. Paper presented at the Proceedings of the 14th ACM conference on Computer and communications security.

yoda. (2005a). LordPE.

yoda. (2005b). Yoda's Protector.

Yuschuk, O. (2008). OllyDbg.

Zeltser, L. (2007). *Reverse Engineering Malware: Tools and Techniques Hands-On*. Bethesda: SANS Institute.

Zhang, Q., Reeves, D., Ning, P., & Purushothaman Iyer, S. (2007). *Analyzing network traffic to detect self-decrypting exploit code*. Paper presented at the Proceedings of the 2nd ACM symposium on Information, computer and communications security.

Zhou, Y., & Meador Inge, W. (2008). *Malware detection using adaptive data compression*. Paper presented at the Proceedings of the 1st ACM workshop on Workshop on AISec.