A Scalable Architecture for Persistent Botnet Tracking

by

Jay Zarfoss

A thesis submitted to Johns Hopkins University in conformity with the requirements for the degree of Master of Science in Engineering.

Baltimore, Maryland

January, 2007

Abstract

The botnet phenomenon has recently garnered attention throughout both academia and industry. Unfortunately, botnets are still a mystery. In fact, today, very little is known about even the most basic botnet properties, such as size, growth, or demographics. The primary reason for this lack of knowledge is the fact that the existing approaches for measuring such properties are simply inadequate; honeypots [30], even those with advanced virtualization [40], cannot scale to the task of botnet tracking, while silent drones do not offer the dynamism necessary to persistently track botnets. Furthermore, both of these techniques provide only one, internal, view of the botnet. As we will demonstrate, this single view will often fail to provide relevant information on botnet size or diversity. Indeed, the fog has yet to clear.

In order to gain a firm understanding of botnet dynamics, we have developed a lightweight infrastructure that overcomes many of the problems of prior approaches. Our infrastructure follows the entire life cycle of the botnet, beginning with the capture of botnet executables from various Internet vantage points. We apply novel techniques to automatically learn the network properties (or so-called "dialects") of the bot binary, and we subsequently transfer this information to a specialized robot. Similar, in spirit, to the aforementioned drone, our scalable robot joins live botnets and offers an insider's view of the malicious network. However, the similarities end here. Unlike a drone, the robot offers real-time responsiveness previously available only with high-interaction honeypots, thus avoiding botmaster scrutiny by intelligently acknowledging botmasters' commands. Moreover, unlike lightweight tracking systems of the past, our approach allows for long term tracking of the migratory botnet – a salient variety of botnet which, at present, remains conspicuously under-documented.

Our insider perspective is supplemented with an external source of scrutiny: DNS cache probing. These cache probes offer information on botnet demographics that, to date, has not been readily available. More importantly, our infrastructure relies on little or no human intervention from the initial point of contact with a spreading botnet to the deployment of these robots and DNS probes.

We have leveraged this automated and dually-faceted infrastructure against (presumably) the most prevalent class of botnet on the Internet today: the Internet Relay Chat (IRC) botnet. Resulting from the tracking of hundreds of IRC botnets, our observations have shed light on the global prevalence of the botnet phenomenon. However, these same observations have also raised many new questions, some of which raise doubts on earlier assumptions and statements regarding botnet dynamics. Arguably, we may never learn the answers to these questions, or win the battle against botnets, until this fog has cleared. To that end, we hope that the collaborative deployment of infrastructures such as ours better positions us to provide answers to the myriad of outstanding botnet questions.

Advisor: Fabian Monrose

Dedicated to the researchers, both academic and otherwise,

who understand the depths of these problems and thanklessly search for solutions.

Acknowledgments

I must thank my family, who supported me through this endeavor like they have through all others. Particular thanks to my mother, Jane, who repeatedly endured my absence from family functions due to my time in the research lab.

I also owe a tremendous debt to my three influences from the 4th floor of the Wyman Park Building at the Johns Hopkins Homewood campus: Fabian Monrose, Moheeb Rajab, and Andreas Terzis. Fabian, my advisor, could motivate a snail to outrun a cheetah. For instance, Fabian once advised a student so averse to writing, that this student judged his higher-education options on a single metric: the fewer English requirements, the better. Fabian then convinced the student to write this paper. Moheeb, the graduate student who neither eats nor sleeps, acted as the icebreaker for many facets of this paper. Time and time again, I was able to progress only after Moheeb laid the ground work. And finally, thanks to Andreas for so many times offering new ideas and directions in our many discussions.

Table of Contents

List of Tables

List of Figures

Chapter 1

Background and Introduction

"Welcome to this IRC Network........

If you have come here against your own will its because you have been oWnEd."

- *Botnet Welcome Message, October, 2005*

Although the existence of botnets has long been known among those in the Internet underground, the botnet phenomenon has only recently attracted attention from mainstream Internet communities. *Botnets* themselves are defined as heterogeneous networks of computers known as *bots*. These bots are usually unwitting participants in the network, and they are generally entered into service by their execution of some form of malware. To communicate with each other, these bots utilize some form of command and control (`C&C`) channel. Typically, a small set of human operators known as *botmasters* preside over this `C&C` infrastructure and issue commands to any and all bots connected to the network.

The `C&C` channel itself may be comprised of any standard or non-standard network protocol. However, the IRC protocol [21] seems to be popular choice among authors of botnet software. The IRC protocol itself has a long history with the Internet underground. For years, IRC served as a predominant medium for social human-to-human interaction on the Internet. This flexible social layer of IRC makes it close to ideal for controlling bots; bots and their botmasters can communicate in the same way that two human clients may carry on a conversation. Furthermore, the protocol benefits from free and relatively easy to use server implementations, and the protocol has many flexible options built-in to suit a variety of botmaster needs.

The type of commands issued to the bots is limited only by the imagination of the botmaster and physical abilities of the bots themselves. These commands may vary widely depending on the personality of a given botmaster, but the community generally accepts the idea that botnets are being used for mostly nefarious purposes. A botnet's actual utility may range from simple bragging rights among hackers, to more serious concerns like spamming, denial-of-service attacks, or identity theft.

Unfortunately, the global frequency at which botnets are used for these particular malicious activities is mostly a point of conjecture. The architecture described in this paper was developed for the sole purpose of measuring some of these activities by enabling a large number of botnets to be discovered and tracked. In the worse case, our infrastructure can record the commands a botmaster issues to his followers, while our best-cases can provide many more details as to a botnet's demographics.

1.1 The Bot Life Cycle

As a first step, the bot life cycle begins with an ordinary personal computer, and generally this computer has some version of the Windows operating system installed. This computer (the victim) will execute a binary which directs it to the botmaster's `C&C` infrastructure(s). There are several means by which the victim may initially receive and then execute this binary. Arguably the most common of these means is an already-infected bot scanning ranges of IP space for computers running exploitable services (typically common Windows services). These services generally run by default on unpatched Windows installations. (A more in-depth look at a particular exploitable service, RPC-DCOM, can be found on Symantec's website [11].) These exploitable services allow already infected hosts to inject a small amount of shellcode onto the victim machine.

The second phase of the bot life begins when the new victim executes the shellcode. This shellcode typically instructs the victim to download and run a much larger botnet binary. The binary may be hosted on the exploiting bot itself, or on some third-party website, and can be transferred through any number of common file transfer protocols (such as FTP, TFTP, or HTTP), and this binary typically contains the necessary instructions to join the botmaster's `C&C` network.

We take a moment to point out that unlike previous generations of worms, botnets tend not to be easily classifiable in their scanning techniques. In the past, worms such as CodeRed [43] exhibited a *uniform* scanning pattern: any IP address was equally as likely to be the target of a scan (assuming correctness of the random number generator used by the worm software). Later malware exhibited *non-uniform* scanning behavior: a worm may be much more likely to scan within its own /8 or /16 subnet than any random IP address. Botnets can exhibit these behaviors, but the addition of a human operator means that botnet scans can often be *targeted*. In other words, a botmaster can inform his victims to scan only a particular /8 or /16 (or even /24) subnet. The result is that a botnet will only find victims in this targeted subnet. An additional consequence of such scanning behavior is that botnets may spread much more slowly, but what they lose in speed, they gain in stealth; only those listening on the right subnet have any chance of noticing the botnet in question. However, even with the use of targeted scanning, we believe that a significant amount of unwanted Internet traffic can be directly attributed to these unique scanning patterns of victim-hungry botnets.

Additionally, other means for botnets to spread their binaries may exist. Naive human operators may receive and open botnet executables as email attachments. Alternatively, URL addresses for botnet executables may be sent through common instant messaging protocols. These infection cases are marginally less frightening, as they require some action on the part of a human operator to complete the infection cycle.

Once the binary has safely been transferred to a new victim, a third (optional) stage of botnet infection may begin. In this step, the newly executed binary may use DNS in order to locate the IP address of the `C&C` server. DNS offers botmasters great flexibility. The use of dynamic DNS means that the exact location of a botmaster's server can change with high frequency. Alternatively, DNS can allow for load balancing between multiple servers. However, DNS also has its disadvantages. Wise network administrators can easily poison these DNS entries by pointing their requests to either a local address or to a safe network black hole. Such poisoning would effectively sever the link between bot and botmaster.

Now with the IP address of the `C&C` server, the victim begins the fourth step, and connects directly to the `C&C` server itself. Once the victim completes a TCP connection handshake with the botmaster's server, the victim may be required to issue a password before it is allowed to complete its connection. In the case of IRC botnets, this password is sent with the use of the `PASS` command. Generally, this information is sent in the clear and can easily be observed in network traces. We presume botmasters choose to send commands in plaintext because common cryptographic implementations would potentially reduce the number of victims they could support with presumably limited hardware resources[1].

Once connected to the `C&C` server, the victim begins its fifth stage in the bot life cycle and completes its transformation to a fully functional member of the botmaster's army. Upon arrival, the bot typically receives a command immediately. In an IRC network, this command usually comes in the form of a channel topic notification, and this immediate command ensures that bots are always productive, regardless of the current activities of the human botmaster. Finally, when the botmaster wishes to manually instruct his followers, he generally must provide a password (different than the password bots used to gain access to the `C&C` server) to the bots, before they will begin accepting his orders. Once the botmaster has authenticated himself, he has the power to run any arbitrary software package on the victim's computer.

Unfortunately, a simple view of the botnet life does not reveal one of the more challenging aspects of botnet tracking: botnet migration. That is, at almost any point in the lifetime of a botnet, the botmaster may direct his minions to a different `C&C` server. Botmasters also commonly upgrade the malware itself by having the bots download a new executable from an easily accessible URL. Tracking these changes poses particular challenges for researches, and we discuss the possible solutions to this problem in the next section.

1.2 Botnet Detection and Tracking

Until now, the tools used to detect and track botnets have been categorized into one of two groups: high-interaction and low-interaction. High-interaction involves the emulation

[1]At least one botnet architecture, namely Agobot, seems to support cryptographic operations, presumably to evade intrusion detection systems.

of all aspects of an operating system, typically via the use of real Windows instances as honeypots. The exploitation of these honeypots can be directly observed through network logs, thereby discovering the existence of a previously unknown botnet. Furthermore, the Windows instance may then be left connected to the `C&C` server, thereby tracking all related botnet activity. Since the honeypot is a fully functional Windows instance, the honeypot should have no trouble tracking the botnet through every possible action or migration which may be commanded by the botmaster. However, long-term tracking with honeypots has some subtle challenges. First, administrators must be very careful to run their honeynet behind a very carefully implemented firewall, sometimes referred to as a *honeywall* [14]. This honeywall can ensure that the same machines used to track botnets do not contributed to nefarious activities, namely denial-of-service attacks or malware-spreading. In addition, administrators must take care that each individual honeypot has very few botware infections. Different botnet executables running on the same host may interfere with each other. Generally speaking, bot software may lock common mutexes or other system resources, thus blocking other software from being properly executed. We believe this behavior to be intentional, as it may help prevent bot software from being interfered with by a usurper. Furthermore, many bot-related operations (such as scanning attempts), may seize all network resources on a given machine. As a result, an administrator must essentially reserve one Windows instance for each botnet he wishes to track.

This high resource cost can be mitigated somewhat by using virtualization. In this case, multiple Windows instances may be run simultaneously on a single physical machine. However, even virtualization offers only a small improvement, as current hardware platforms can only support a few virtualized Windows instances with any reasonable efficiency. Furthermore, virtualized honeypots may also fail to run malware correctly if the malware in question uses techniques to check for virtualization [33].

Complex honeypot-management systems, such as Potemkin [40], while arguably mitigating scalability problems for botnet detection, do not address scalability issues for botnet tracking. The fundamental problem is simple: unlike high-interaction detection, high-interaction tracking requires persistent participation from the honeypot. As a result, the resources for a tracking honeypot cannot be reclaimed. Essentially, botnets and honeypots must maintain a bijective, one-to-one mapping throughout the tracking period. So although honeypots offer the best possible mechanism to discover and track botnets, they cannot be used cost-effectively to track a large number of botnets simultaneously.

Low-interaction solutions may be implemented with much more lightweight components which do not suffer from the scaling problems of their high-interaction counterparts. However, unlike the heavyweight honeypots which may be used for both detection and tracking of botnets, a separate lightweight solution exists for each of the detection and tracking problems. The malware collection tool (Mwcollect) and Nepenthis [1, 2] represent the most popular means to collect malware in a lightweight fashion. These tools, often referred to as *lightweight responders*, emulate commonly exploited services of the Windows operating system in order to collect the previously mentioned shellcodes. These shellcodes are then parsed for the associated malware location (generally a URL). If a location address is found via the parsing of the shellcode, the lightweight responder attempts to download the binaries for further analysis. Ironically, since lightweight responders do not actually execute the binaries in the collection process, they are not affected by anti-analysis techniques that may exist in the malware itself (such as virtualization detection). Currently, the primary flaw with these responders is their inability to parse obfuscated shellcodes for the location of the associated binary. This problem is the focus of ongoing research, and other researchers are currently examining means by which this problem can be overcome [2].

Low-interaction botnet tracking involves the use of a lightweight client to join the `C&C` server, using whatever communication protocol is necessary. In the case of an IRC network, this client can be nothing more than a simple IRC-supporting chat client. Previous work has used a simple drone client [14, 29] to do lightweight tracking of IRC-based botnets. The incredibly small footprint of these automated clients allow many botnets to be joined simultaneously without abundant physical resources. Unfortunately, unlike their heavyweight honeypot counterparts, these drones lack the responsiveness required to satisfy many botmasters. As a result, lightweight drones may quickly lose contact with an IRC-based botnet due to a failure to follow a migration or by direct banishment from the channel's botmaster (presumably due to the failure to acknowledge a command).

We believe our infrastructure improves upon the state of the art and mitigates many of these problems of earlier low-interaction botnet tracking systems. In particular, we supplement a front-end of many low-interaction botnet tracking robots with a much smaller support group of high-interaction Windows instances. We use these heavyweight Windows instances via a process of graybox testing to train our trackers. Thus, our low-interaction trackers gain two primary advantages over their earlier drone counterparts. First, our new trackers can better emulate standard communications for varying bot-dialects, thus appearing to the botmaster as a viable follower. Second, our many lightweight trackers can harness the power of only a few heavyweight Windows instances in order to learn how to follow certain botnet migration instructions. Finally, our technique of DNS cache probing further supplements our system in order to overcome any inherent inabilities of our insider agents. Taken as a whole, we believe this system provides a viable means to effectively discover and track botnets.

Chapter 2

General Infrastructure Overview

"You can't consider the problem of defense without first understanding the problem of attack."

- Doug Tygar, UC Berkeley, quoted from [24]

As a whole, our infrastructure can be divided into two logical segments: malware collection and wild botnet tracking. The former segment directly supports the latter. In other words, before we can track botnets in the wild, we first must become aware of their existence. In order to collect a malware binary, we use both lightweight and heavyweight collection tools. On our locally assigned IP space, we employ both lightweight responders and a small number of VMWare-based [36] virtualized Windows honeypots behind a distributed darknet. We allow the lightweight responders to cover roughly half of our IP space, but we disable the automatic download features of the responders. Instead, we merely collect the parsed addresses and download the files in an offline fashion. This mechanism allows us to download only previously unseen executables, while ensuring that we do not overload a particular server with simultaneous download requests.

A specially designed honeywall (in the form of a modified Snort-inline [25]), separates the other half of our IP space from a small collection of virtualized Windows honeypots. We use Network Address Translation (NAT) to map a relatively large IP space to only a handful of honeypots. The honeywall meticulously monitors traffic to the honeypots to ensure that misbehavior on the part of the honeypots (scanning or denial-of-service activity) is disallowed. This gateway also ensures that the honeypots do not cross-infect each other, and it does not allow more than one connection to a particular botnet IRC server. Ensuring that the gateway blocks all but the unique botnet traffic is accomplished by observing IRC-specific commands (e.g. `NICK` or `JOIN`).

Because honeypots represent such a precious resource, we do not use these machines for long-term botnet tracking. Instead, each honeypot is regularly re-imaged with a clean installation of Windows XP. The previous image is compared with the new clean image, and any suspicious files are extracted for analysis. During the height of our infrastructure's operation, virtual instances of Windows were reloaded daily. By using honeypots in this fashion, we ensure the collection of botnet binaries using obscure (or possibly "0day") exploits or obfuscated shellcodes that cannot be collected from the current versions of our lightweight responders. Furthermore, reloading our honeypots daily ensures that the honeypots are functional. Left alone for long periods of time, the honeypots are likely to suffer from severe instability due to malware infection. Additionally, malware may actually repair other vulnerabilities of a honeypot, thereby stifling further infections.

In order to collect as wide an assortment of binaries as possible, we supplement collection on our own IP space by harnessing the power of the distributed platform known as PlanetLab [28]. PlanetLab offers darknet IP space over a variety of IP ranges; we utilized ten different /8 prefixes for our experiments. Harnessing the power of PlanetLab ensures that our collection of botnet executables well represents the types of malware currently spreading on the Internet. Unfortunately, hosting off-the-shelf lightweight responders such as Nepenthes on PlanetLab is nontrivial and required substantial modifications. A more detailed discussion of these modifications can be found in earlier work by Rajab *et al.* [32].

Ideally, once we have collected a malicious binary, our goal is to track the behavior of the associated botnet (assuming the `C&C` infrastructure is still active). To this end, each binary is submitted for a detailed analysis in the graybox testing process. The output of the graybox testing process allows for the botnet associated with a given binary to be tracked through two lightweight tracking mechanisms: a DNS cache prober and a smart IRC tracking robot. Both the prober and the robot are highly scalable, and represent significant advancement over previous approaches [14, 29].

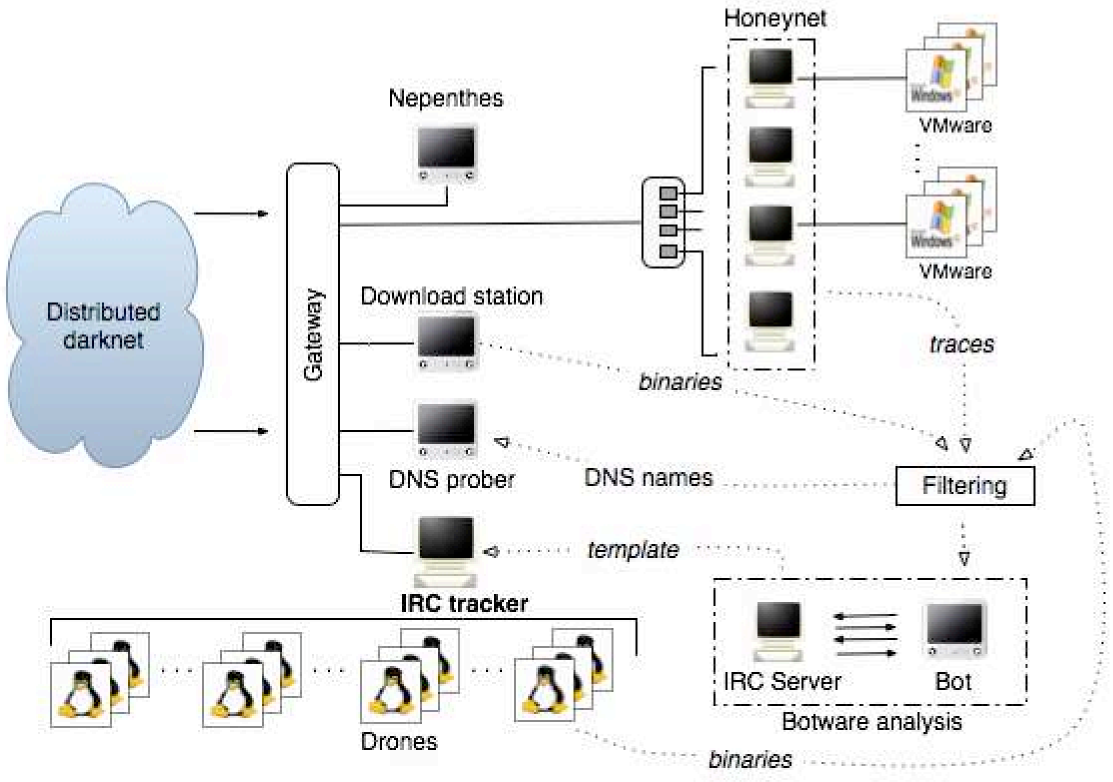


Figure 2.1: Overview of all components currently in use by our scalable detection and tracking infrastructure

The entire infrastructure can be seen in Figure 2.1. In general, the infrastructure uses only a limited number of heavyweight components, represented by the virtualized honeypots, and to a lesser extent, the graybox testing pipeline. Via this graybox testing pipeline, these few high-interaction components may support a much larger array of low-interaction trackers. The output of the graybox testing is basic information such as the IP address of the C&C server, as well as more advanced information to instruct a drone how to actively respond to botmaster commands. We further point out that the source of binaries need not be from the collection portions of our infrastructure. The graybox analysis and tracking components could easily operate via third-party-provided binaries. Generally speaking, the only tightly-coupled components of the infrastructure are the graybox testing outputs and the IRC tracking robot. All other components could easily be configured for use with an alternative infrastructure.

Chapter 3

Graybox Testing

"A botnet is comparable to compulsory military service for windows boxes."

- *Stromberg*

Given only a suspicious binary, we use a special brand of graybox testing to determine the botnet-specific information embedded in this binary. All suspicious binaries undergo the same rigorous testing process regardless of its origins. The purpose of this testing is several fold. First, this particular form of binary analysis allows us to first confirm what we already suspect: a given binary is in fact a botnet executable. Second, the testing allows for the creation of a malware database. This database catalogues the observed networking features of every binary and presents a wealth of information for the research community. The information may be of particular use to researchers attempting to reverse engineer obfuscated or otherwise "packed" binaries, since our infrastructure analyzes network features apart from the actual structure of the associated file. Finally, we harness the data collected from each binary for a much higher purpose, namely the configuration of a smart IRC tracking robot. The network fingerprints allow the robot to know not only *where* to connect (the server, port, and channel), but also teach the robot *how* to communicate with its quasi-botmaster. To accomplish this goal, the testing infrastructure generates both a configuration file and a bot-dialect template. These items are then passed to the tracking portion of the infrastructure.

Unfortunately, the process of generating robot configuration files is not very efficient.

In particular, the generation of a full template represents an especially slow process. More specifically, processing a single binary requires between 15 minutes and 35 minutes, variable on the behavior of the binary under scrutiny. As much as half of this processing delay is due to the time required to reload and restart the virtualized operating system (to ensure that it is running in a known state). Upcoming iterations of our infrastructure should greatly reduce this less efficient processing with the use of VMWare checkpoints for operating system reloading. Despite the amount of time required, no aspect of this process requires any form of human intervention; therefore, binaries may be processed immediately one after the other. We also point out that given only a few additional resources, many binaries could be processed simultaneously, thereby further mitigating the time delays. This level of automation is made possible because our binary analysis makes no use of debuggers, disassemblers, or other human-interactive tools traditionally used in binary analysis.

3.1 The Graybox Testing Pipeline

The actual graybox testing takes place on an isolated network segment of only two physical machines. One of these machines plays the role of the victim by running a virtualized copy of Windows XP. The other machine plays the role of both the `C&C` server and of the botmaster. We will often refer to the server as the testing server to distinguish it from a live `C&C` server in the wild. The testing server acts as a network black hole, and all network traffic from the victim machine is routed to this sink. As a result, the testing server can log any interesting network behavior of the victim, and it also prevents bot traffic from escaping the network segment. The entire testing process begins when a malware-collection entity copies a binary into the staging directory of the testing server. This process is many-staged, and an explanation of each of these stages proceeds below.

Stage 1: Initial Processing. The testing machine periodically checks to determine if any binary is currently being processed. If no binaries are currently undergoing processing, the testing machine then searches through four different staging areas for new binaries, with each staging area being marked by a distinguishing two-digit identifier. The two-digit codes represent the origins of the binary: extraction from virtual machines on the honeynet, local collection from lightweight responders on JHU IP space, collection from lightweight responders on PlanetLab space, or a download from an IRC tracking robot instance, respectively. The binaries from these four staging areas converge into a more central processing repository. Here each executable is renamed with a unique binary identifier, b_{id}. This identifier consists of the timestamp of the binary file (as opposed to the current time), a cryptographic hash[1] of the binary file, and the two-digit origin code. The timestamp and origin code illustrate where and when this binary first materialized, while the hash value establishes the uniqueness (or non-uniqueness) of a new binary. The b_{id} is used henceforth as the identifier for a binary in all aspects of the graybox testing process. If the b_{id} contains an already-processed hash sum, the testing machine logs this discovery and exempts the duplicate from further processing. Otherwise, the suspected malware is inserted in a queue to begin further testing.

Stage 2: Creation of a Clean Network Fingerprint. Analysis first begins on a renamed and uniqueness-verified binary when it leaves the aforementioned queue. First, the victim machine loads a clean virtual image of Windows XP. This clean image runs a simple program to download and execute the binary from the testing server. During the execution of this binary, the server logs any network traffic coming from the victim machine. Since the server acts as a sink in this network segment, any network activity initiated by the suspected malware will be detected on the appropriate network interface. This sink includes the server's use of a special-use DNS server to reply to any DNS lookup with the local IP address of the testing server. Furthermore, since the victim is running an otherwise clean version of Windows XP, malicious network activity can easily be distinguished from benign network activity. The server filters these traffic logs to create a clean network fingerprint $f_{net} = \langle \texttt{DNS}, \texttt{IPs}, \texttt{Ports}, \texttt{Scan\_Ports} \rangle$, respectively representing the target(s) of any DNS lookup requests from the victims, the source of any IP connection attempts, the destination ports of any connection attempts, and the ports on which the binary exhibited scanning behavior. f_{net} does not include IP addresses and ports of traffic that is generated

[1]We utilized the MD5 algorithm to compute cryptographic hash sums of 128 bits.

normally by Windows, such as a normal level of contact attempts to `windowsupdate.com`. Furthermore, scanning behavior in this network trace implies that the binary attempts to spread itself by known exploits without the explicit order from a botmaster. We define a scanning port to be any port for which there were attempts to contact more than n (=20) different IP addresses within the allotted network collection window of w (=3) minutes. We admit that because of the size of the window in which the clean network fingerprint is collected, some interesting network features may be overlooked. For instance, an attempt to contact a secondary server after $w+1$ minutes would go unnoticed; however, time constraints do now allow for the indefinite observation of a given executable. Furthermore, we have not noticed behavior of this type on any of our honeypots that have suffered a malware infection for more substantial periods of time[2].

In general, f_{net} contains enough information to discover the precise Internet location of a `C&C` server. Furthermore, an analyst may use this information to check on the current status of a `C&C` server (ie, to see if it is still accepting TCP connections). However, f_{net} does not contain any application-level protocol information, so it may not be possible to create a sustainable connection to a `C&C` server given only an f_{net} fingerprint. In particular, the `C&C` server may require an application-layer password before an agent is allowed to join the IRC server. Also, two botnet binaries could use widely different communications protocols (IRC vs HTTP) but have identical f_{net} fingerprints. This circumstance is possible only because botnets often run their distributed communication over non-standard ports.

Stage 3: Creation of an Application-Layer Fingerprint. In order to gain insights into the application-layer practices of the binary, we must first look back at what we learned from the network fingerprint. Rather than providing a complete picture, f_{net} offers enough information to continue the next stage of processing, namely the `Ports` and `Scan_Ports` fields. Presumably, the set difference of `(Ports - Scan_Ports)` includes any port which the botnet uses for communication. As such, the testing machine configures a specially modified version of UnrealIRCd [39] on precisely these ports. We quickly discovered the importance of not running an IRC server on the `Scan_Ports`, as listening on `Scan_Ports` causes the

[2]Admittedly, we cannot deny the possibility that these peculiar behaviors did in fact occur without drawing our attention.

network stacks to quickly become overrun with all of the connection attempts from the victim. Using this set difference of ports in our IRC server configuration ensures that we capture proper bot connection attempts when the associated botnet uses nonstandard ports.

We modify a base configuration file for UnrealIRC in order to ensure that the server listens only on ports relevant to the binary at hand. Once the IRC server is configured, the testing machine orders the destruction of the already running image of Windows XP. In its place, a new untainted image is loaded, and this image is commanded to run the same binary as before. This reloading is necessary to ensure that the binary again attempts to establish a connection to its hard-coded `C&C` server address. (In all likelihood, the binary instance run in Stage 2 would have since halted any connection attempts.) Once the new binary begins, it should establish a TCP connection to our special IRC server. In the case that the binary is not representative of an IRC botnet, the connection will fail after the initial TCP 3-way handshake. However, we nonetheless record this transaction to aid in possible future forensic analysis of the binary. Among the few binaries not using IRC, HTTP seems to be the protocol of choice, and the HTTP protocol is easily distinguishable by its `GET` or `POST` requests. Conversely, IRC-based executables are easily recognizable by the initial `NICK` or `PASS` commands of an IRC handshake. In the case that the binary is representative of an IRC botnet, the connection should complete normally, and the bot instance will join our server. After this successful IRC connection, the server automatically extracts an IRC fingerprint, $f_{irc} = \langle \texttt{PASS}, \texttt{NICK}, \texttt{USER}, \texttt{MODE}, \texttt{JOIN}, \texttt{CHANPASS}, \texttt{NOTICE} \rangle$, respectively representing the password used to complete the initial connection to the server, the nickname chosen, the username chosen, any modes (channel or user) set, channels automatically joined by the victim, channel passwords used, and any special notices sent to the IRC server. All of these fields are very easily observable; the binary transmits all of these fields in the clear upon initial connection, as observed in Figure 3.1. Our IRC server is modified to ignore any password, either server password or channel password, as well as to ignore any modes set or notices sent by the victim. Taken together, f_{net} and f_{irc} provide enough information to join a botnet in the wild.

However, for very subtle reasons, a single fingerprint pair of f_{net} and f_{irc} is not enough to *repeatedly* join the same botnet in the wild. The reason is due to subtle features of the

```
PASS 1181
NICK xyz-546101
USER zewuprly 0 0 :xyz-546101
NOTICE Version 2.1
JOIN #tow 1182
MODE xyz-546101 +xB
```

Figure 3.1: Example of application-layer data automatically sent to the IRC server from a typical bot-client. In this example, $f_{irc} = \langle \texttt{1181},\ \texttt{xyz}-\texttt{546101},\ \texttt{zewuprly},\ +\texttt{xB},\ \#\texttt{tow},\ \texttt{1182},\ \texttt{Version 2.1}\rangle$.

IRC protocol [21]. The IRC protocol generally does not allow for two users to have the same nickname on a given IRC server. To avoid naming collisions among their minions, botmasters generally program their bots to choose names with some amount of randomness. A simple naming scheme might be to choose a nickname of `USA|32321`, where the suffix numerals are chosen randomly. Unfortunately, determining which characters are deterministic and which characters are random is not always so obvious. Furthermore, assuming all of the characters are deterministic and repeatedly joining a wild botnet with precisely the same name is certainly not advisable. A botmaster might start to notice. To avoid this issue, the IRC fingerprint, f_{irc}, is collected twice. In practice, this double collection does not cause any slowdown in the pipeline, as two virtual instances of Windows XP are loaded in parallel. The only difference between the images is their network IP address. Thus, by comparing the two independent IRC fingerprints, $f_{1_{irc}}$ and $f_{2_{irc}}$, the testing machine can quickly compare the two and discover the deterministic and random portions of the IRC features. For example, a comparison of `USA|32321` and `USA|00554` quickly reveals the random suffix and deterministic prefix. IRC tracking robots will now be able to use different nicknames every time they reconnect to a botnet, while still ensuring those nicknames conform to the botnet's naming scheme. This ability is crucial in enabling our infrastructure to complete long-term and stealthy observation of botnets.

At this stage in the process, enough information has been gathered about the given binary to allow for some level of tracking of the associated botnet. As such, the testing machine will generate an appropriate configuration file for an IRC tracking robot. With this configuration file, the robot will have the ability to join a wild IRC botnet without immediately giving away its identity as a drone. This level of tracking has been accomplished by others, namely by Freiling *et al.* [14]. Unfortunately, this level of tracking is not sufficient for long-term analysis of botnets. By not acknowledging `C&C` commands, this earlier style drone will quickly stand out to an experienced botmaster. At worst, the drone could give itself away immediately after joining the IRC channel. This worst case scenario could easily happen because botmasters commonly issue commands within the topic messages. These messages are automatically sent to every actor that joins the `C&C` channel, and a botmaster who observes an agent joining his channel without responding to the topic command may immediately suspect foul play. This suspicion could lead to the botmaster banning our agent from his server, thus putting a stop to our data-collecting efforts. In order to avoid all of these problems of a silent drone, our robot must acknowledge commands coming from the botmaster. The learning of these acknowledgments begins in the next stage.

Stage 4: Discovery of the Command-Delimiter. To discover how to acknowledge botmaster commands, the testing server must attempt to discover how this binary distinguishes botmaster commands from ordinary channel traffic. From what we have observed on our honeypots, every botmaster chooses a delimiter to distinguish his commands from ordinary channel text. For example a command to begin scanning may appear as `.scan` or `$scan`, depending the botmaster's preference when he compiled his malware. To attempt to discover this special character, the testing machine employs a special delimiter agent. Upon execution, this delimiting agent joins the modified IRC server and enters a predefined channel. The IRC server then forces the malware agent into this same channel so that the delimiter agent may easily send commands to the victim. Once both players have entered the channel, the delimiter agent uses a variety of different commands (currently 12) with every possible command-delimiter. The commands are chosen to include the widest breadth of botnet dialects, as observed from our honeypots. Most of these commands are also mentioned as part of botnet source code analysis in work by Barford and Yagneswaran [3]. Figure 3.2 illustrates a list of the commands currently used by our system. This list is flexible, and additional commands could be added without significant impact on performance. If the victim agent responds, the specific ASCII delimiter character that evoked a response from the agent is recorded.

```
scanstop
bot.sysinfo
scanstats
harvest.cdkeys
version
ver
id
secure
status
makedir test
ddos.syn 10.0.0.1 80 1
udpflood 10.0.0.1 80 1
```

Figure 3.2: Complete listing of the 12 commands involved with the delimiter discovery process.

However, evoking a response cannot be achieved by simply sending commands over the IRC channel. Bots will generally not respond to their masters without the operator first authenticating himself to the bot [29]. This authentication practice is very common and prevents dueling botmasters from stealing each others hard-earned bots, and this simple authentication is a quintessential component of the fifth stage of the bot life cycle (as discussed previously in Section 1.1). Unfortunately, we often do not know the password that a particular binary requires. In cases where the binary was extracted from a honeypot, the botmaster password can sometimes be observed in the honeypot network logs. However, as resources are limited, we prefer to use lightweight responders to collect the binaries [1, 2]. In these cases, the botmaster password is unknown. Fortunately, we have developed a simple trick to circumvent this problem. Under normal circumstances, a bot will acknowledge a command that has been set as the topic of the channel. When a victim joins a channel, this topic is automatically sent to him in the form of an `RPL_TOPIC` message [21]. All IRC bots seem to execute these `RPL_TOPIC` messages without authentication. We modified our IRC server to allow for the delimiter agent to send these `RPL_TOPIC` commands directly to the victim. These modified commands allow us to evoke responses without knowledge of the aforementioned authentication password.

In practice, sending 12 different commands with 255 different ASCII characters each is not very efficient, for a couple of reasons. The first inefficiency arises because IRC servers typically enforce very strict rate-limiting in order to protect themselves from denial-of-service attacks. This problem is addressed by disabling the rate-limiting functionality in the source code of our customized UnrealIRC server. The second problem arises due to normal network delay between the victim and the IRC server. More specifically, after sending a command with a particular delimiter, the delimiter agent must wait for a short delay (on the order of a few seconds) for the victim to respond. The delay can be caused by several factors. The malicious software may naturally respond slowly due to poor threading implementations, these particular trials may lose efficiency due to the use of an virtualized Windows XP instance, other network scanning activities of the malware may be causing network delay, or the malware may be designed to respond slowly (so as to not trigger rate-limiting on public or unmodified IRC servers). Indeed, efficient malware can be hard to find. In practice, the delimiter agent waits for a reply for five seconds. Without this delay, determining the particular command which evoked the response becomes difficult. In order to reduce the number of delays, the delimiter agent utilizes a type of binary search algorithm to determine the appropriate marker as quickly as possible. More specifically, the delimiter agent first sends every possible combination of command and marker as a pre-filtering mechanism. The agent then waits for a response for several seconds. If a response is received, the agent quickly begins a binary search algorithm among the possible 255 delimiters by sending in rapid succession each of the 12 commands within the first half of the 255 possible ASCII delimiters. This process continues in a binary search fashion until the precise delimiter is determined. For example, if the delimiter character for a particular binary is a dollar sign, $, (ASCII character 36), the ranges of ASCII characters sent would be: 0-255, 0-128, 0-64, 0-32, 33-64, 33-48, 33-40, 33-36, 33-34, 35-36, 35, 36. The expected number of ranges sent is $2 * log(n)$, or in this case, 16. The alternative would be to try each delimiter linearly, with a naive expectation of 128 commands sent before a delimiter is discovered. Of course, this number could be lowered by first trying more "popular" delimiters, but we do not believe this approach offers much improvement over the binary search approach.

In some cases, the binary may fail to respond to all combinations of commands. Fortunately, failure is discovered almost immediately, thanks to the initial pre-filtering step. We believe these failure cases are due mainly to a simple security mechanism within the bot executable. The security mechanism is that the bot can be programmed to respond only to an IRC agent with a particular nickname. In cases where the particular botnet `C&C` server is up and running, a drone can join this rogue server in an attempt to discover the name of the operator. Unfortunately, even if the server is active, this technique is not always reliable, as botmasters are not always online, while others may change their name only immediately before issuing a plethora of commands. Another reason for failure could simply be that this particular bot uses a set of commands that we have yet to observe (either in the wild, or in the literature). We believe the former possibility is more common than the latter, as re-use of bot software seems to be very common throughout the Internet underground.

After the delimiter agent has finished, the graybox testing process forks and proceeds down one of two possible paths. If the binary's delimiter cannot be successfully determined, the testing machine catalogues all of the information about this particular binary by its b_{id} identifier. A drone configuration file is then sent to the tracking section of the network for use by a IRC tracking robot. Of course, in this case, a robot instance will be completely silent in the wild and not acknowledge botmaster commands. However, if the binary delimiter has been successfully collected, the testing machine makes note of the delimiter and continues onto the next and final stage of the testing pipeline.

Stage 5: Generation of an Executable-Specific Template. Delimiter character in hand, the testing machine next employs the assistance of a query agent. Like the delimiter agent, the query agent joins our IRC server and enters a similar channel to the victim. Using the `RPL_TOPIC` command-sending technique, the query agent methodically sends the victim a plethora of commands, recording the responses in a simple repository. We utilize a wide variety of botnet commands, as observed by our honeypots. Alternatively, one could use commands found in known botnet sources [3] or by examining strings of an unpacked botnet binary[3]. This query process is the most time consuming portion of the graybox pipeline, requiring as many as 30 minutes with our current implementation. Delays between commands account for most of the time expenditure. However, other specialty commands cause particular delays. For instance, a command to restart the victim's computer may

[3]We consciously choose not to address the complexities of binary unpacking.

force the virtual machine to restart, thus requiring the query agent to wait on the order of minutes before the victim rejoins the channel.

Furthermore, the complexity levels of botnet acknowledgments are very non-uniform. For instance, a few commands involve merely a static reply. An example command of this type would be a `.version` command. In response to this type of command, the bot sends nothing more than an embedded identifier string. However, most commands are not quite this simple. For example, many commands involve at least one argument. For example, downloading commands such as `.download http://address.com/` generally require the victim to mention the URL in the response. The query agent searches for these arguments in the responses from the victim and labels these argument tokens before saving the response to the template. In the cases where the command involves more than one argument, the query engine sends commands with very distinct arguments so that the argument indexes can easily be labeled in the responses. An example of argument replacement is demonstrated in Figure 3.3.

```
//Query agent sends command:
Query-man:Query-agent@10.0.1.100 PRIVMSG #default
:.download http://---.com/test.exe c:\test.exe

//Malware sends to channel in response:
USA|154276039!rsdjcvtgse@10.0.2.100 PRIVMSG #default
:(DoWnLoAd) Downloading URL: http://---.com/test.exe to: c:\\test.exe.
USA|154276039!rsdjcvtgse@10.0.2.100 PRIVMSG #default
:[DOWNLOAD]: Link o Dns Non Trovato, Riprova!: http://---.com/test.exe.

//Resulting Template Entry:
(DoWnLoAd) Downloading URL: **x1** to: **x2**.
[DOWNLOAD]: Link o Dns Non Trovato, Riprova!: **x1**.
```

Figure 3.3: In response to a `.download` command, a scrutinized bot replies with one of the arguments originally sent by the query agent. This argument is flagged by the query agent before the response is stored in the template.

Other commands involve the release of more in-depth information. For instance, many commands will evoke reports of a computer's statistics. These statistics can range from network usage, to lists of installed software, to the speed of the processor. Fortunately, the query agent is acutely aware of every reasonable statistic about the virtual machines. Consequently, the query agent can quickly pattern-match these statistics to change them to more realistic values. For example, the internal IP address of the virtual machine is replaced with a special IP address flag, as shown in Figure 3.4. Additionally, in cases where a binary may respond with registration keys of software, the query agent scrambles these keys into random garbage before saving them to the template.

```
//Query agent sends command:
Query-man:Query-agent@10.0.1.100 PRIVMSG #default
:.netstat

//Malware sends to channel in response:
USA|154276039!rsdjcvtgse@10.0.2.100 PRIVMSG #default
:[NETINFO]: [Type]: LAN (LAN Connection).
[IP Address]: 10.0.2.100. [Hostname]: 10.0.2.100.

//Resulting Template Entry:
[NETINFO]: [Type]: LAN (LAN Connection).
[IP Address]: **EXT-IP**. [Hostname]: **EXT-IP**.
```

Figure 3.4: A response to a `.netstat` command includes the IP address of the local machine. This IP addressed is replaced with a special marker in the template entry.

Whenever possible, the query agent will take shortcuts. For instance, scanning commands typically involve a botmaster listing as an argument the exploit module he would like to use in the scan. Unfortunately, the names of exploit modules seem to be some of the most highly customized features of many of the bot executables, and we have observed many unusual names to represent common exploits. In cases like this, the query engine needs to know to ask the right questions. To get an idea of what scanning commands to issue, the query engine can use the fact that many bot binaries support a command to return scanning statistics. In other words, the bot will report on how many other machines it has infected with a given exploit module. If the query module observes a response to one of these `.scanstat` command variants, it will parse the list of returned exploits and use those specific modules to create scanning commands. These scanning commands are usually acknowledged with the port associated with the listed exploit module. As a result, getting a complete picture of a binary's scanning capability hinges on knowing the names of these exploit modules. Figure 3.5 demonstrates a sample response to a `.scanstat` command and a resulting follow-up command. If the victim does not respond to any `.scanstat` variants, the query agent sends commands with the most common exploit names observed in the wild; however, this process involves the use of many more commands, as the names of many exploit modules will likely be sent that are not supported by this particular binary.

```
//Malware sends to channel:
USA|154276039!rsdjcvtgse@10.0.2.100 PRIVMSG #default
:[SCAN]: Exploit Statistics: Dcom135: 0, lsass_445: 0, hearts1: 0,
Total: 0 in 0d 0h 6m.

//Follow up command from query agent:
Query-man:Query-agent@10.0.1.100 PRIVMSG #default
:.advscan hearts1 8 5 6 -r

//Malware acknowledgment reveals port of ‘‘hearts1’’:
USA|154276039!rsdjcvtgse@10.0.2.100 PRIVMSG #default
:[SCAN]: Random Port Scan started on 10.0.x.x:135 with a delay
of 5 seconds for 6 minutes using 8 threads.
```

Figure 3.5: In response to a `.scanstats` command, a bot replies with a list of exploit modules. The query agent uses these module names to learn port associations. In this case, the module "hearts1" is associated with port 135.

The query agent is also aware of many commands which involve threading, and threads pose an interesting challenge. More specifically, bots may respond differently to a given command depending on whether or not a particular thread is running. The quintessential example of such a command is any variant of the `.scanstop` command; the `.scanstop` command informs the bots to halt their search for new exploitable hosts. Furthermore, the response given by the victim varies widely depending on whether or not a scanning thread is already running. Figure 3.6 demonstrates different responses to the same `.scanstop` command due to thread states, along with the resulting template entry. The query agent attempts to issue commands in an order such that all possible responses are evoked from the victim. For these particular types of responses, the query agent leaves a flag in the template to denote the order in which a current command was issued. These flags are generally simple, denoting a single transition of a thread to an active or inactive state. The IRC tracking robot too must keep state so as to know which response to use. This is an unfortunate layer of complexity, but it is a necessity as many of the most common commands, namely scanning commands, require a small amount of state to be accounted for. In the particular case of scanning threads, the robot must keep track of the exact number of threads which are active; however, this is the exception rather than the rule.

Most processes are either running or not, and scanning-related threads seem to be the only processes that can run in any arbitrary quantity.

```
//Initial command sent from the query agent:
Query-man:Query-agent@10.0.1.100 PRIVMSG #default
:.scanstop

//Reply #1 from the malware:
USA|154276039!rsdjcvtgse@10.0.2.100 PRIVMSG #default
:[SCAN]: No Scan thread found.

//Scanning command ordered by the query agent to change state:
Query-man:Query-agent@10.0.1.100 PRIVMSG #default
:advscan hearts 8 5 6 -r

//Incidental reply to the scan:
USA|154276039!rsdjcvtgse@10.0.2.100 PRIVMSG #default
:[SCAN]: Random Port Scan started on 10.0.x.x:135 with a delay
of 5 seconds for 6 minutes using 8 threads.

//Repeated command from the query agent:
Query-man:Query-agent@10.0.1.100 PRIVMSG #default
:.scanstop

//Reply #2 from the malware:
USA|154276039!rsdjcvtgse@10.0.2.100 PRIVMSG #default
:[SCAN]: Scan stopped. (8 thread(s) stopped.)

//Reply #2 when processed for insertion into the template:
:[SCAN]: Scan stopped. (**THREADS** thread(s) stopped.)
```

Figure 3.6: Different responses from the `.scanstop` command. Each different response is given its own entry in the template directory, and the query agent replaces the thread count with a special marker representing the number of active scanning threads.

We have only encountered a few commands that seem to be completely beyond the scope of the query agent. The quintessential example is a shell command. With a shell command, the botmasters access a command prompt of the victim through the IRC channel, as seen in Figure 3.7. While a handful of the most common shell commands could potentially be added to the query agent's repertoire, recording vast swaths of different responses from a Windows command prompt is no substitute for a well-camouflaged virtual machine.

Of course, this entire process of generating a template for a binary's dialect is made possible only by the fact that almost all botmasters seem to use one of a handful of popular botnet architectures for developing their software. As a consequence of using one of a

```
[DeXTeR]!alexo@185-.-.-.net PRIVMSG [Del]29466 :.cmd services.bat
[DeXTeR]!alexo@185-.-.-.net PRIVMSG [Del]29466 :.cmd ftp.exe -i -s:a
[DeXTeR]!alexo@185-.-.-.net PRIVMSG [Del]29466 :.cmd tftp -i ftp.--.net ...
[DeXTeR]!alexo@185-.-.-.net PRIVMSG [Del]29466 :.cmd ipconfig
```

Figure 3.7: A botmaster issues shell commands directly over the IRC channel.

handful of architectures, many botnets seem to share similar commands. Researches at the Honeynet Project name a few prevalent bot software packages [29], and a more complete list of these architectures has been compiled by Barford and Yagneswaran [3]. We believe these previous works have at least mentioned most of the malware architectures we have successfully analyzed. Rather than changing fundamental pieces of these pre-made packages, the botmasters seem to provide additional modules and exploits, as well as certain formatting and configuration details (for example, which ASCII character to use as a delimiter). If all botmasters were using custom-built botnet software, our technique of querying would probably not be practical. In this sense, we are quite fortunate. Ironically, some of the most trivial differences between botnet dialects are arguably the most important. For example, many bots format their responses with particular color codes. These color codes cause the graphical IRC clients to display the messages in with different color formatting. Responding with an incorrect color code would immediately stand out on an IRC channel, but the query mechanism with resulting templates can ensure that the IRC tracker avoids such a faux pas.

Once the query agent has finished barraging the binary with commands, the testing machine saves the template with the appropriate b_{id}. Furthermore, all collected information relevant to this binary is archived. This archive includes the template (if applicable), an appropriate configuration file for a robot, PCAP [18] logs of all network transactions with this particular binary, and a short human-readable summary of the f_{net} and f_{irc}. As stated before, the process requires no human intervention. These archives may then be used to aid in an in-depth analysis of the malware. Furthermore, over a long period of time, these archives may be examined to look for interesting and emerging trends in the general behavior of these bot executables. After the template and configuration file are provided to the botnet tracking portions of the infrastructure, the next binary is processed from the queue.

3.2 The Indoctrination of the Non-believers

Of course, we must acknowledge a fundamental limitation in all of the previous five stages: every one of these stages assumes that any given malware binary will execute on a virtualized Windows instance in exactly the same fashion as it would on a standard Windows instance running directly on hardware. In truth, this may not be the case. Malware writers may include prebuilt software packages for virtual-machine-detection when compiling their malware. For our purposes, the exact techniques used to detect a virtualized environment are less important than the fact that some wild binaries simply will not execute in a virtualized environment. These particularly clever binaries may be distinguishable by a network fingerprint, f_{net}, in which all fields are null. (Since the binary never completed execution, it never began any distinguishable network activity.) However, such a bland fingerprint is probably not indicative of the binary's true behavior, and so any binaries with seemingly empty fingerprints are set aside for a different processing. And while this further processing does not fit directly into our infrastructure, it serves to answer a specific question: do malware binaries in the wild check to see if they are running under a virtualized operating system?

To answer this question, we first isolate all executables who proved to be uncooperative during the previous stages of graybox testing. We define an executable as uncooperative if it did not result in any interesting network behavior, as observable in f_{net} collected during graybox testing. These executables are then copied to an entirely separate (and isolated) network segment which consists of a pair of computers: a victim computer on which malware is run, along with a DNS-supporting server to act as a network sink. However, unlike systems used in the primary graybox testing pipeline, this victim machine has no operating system installed by default, and the server machine also runs a special disk-imaging server in order to provide full disk images for the victim machine. We utilize Norton Ghost [38] for this for rapid and automatic deployment of pre-made disk images.

To create network fingerprints of binaries based on their non-virtualization behavior, we set the victim machine's boot order to boot from the primary hard disk first, followed by an external media drive. A special Ghost boot disk is left in this external disk drive. During the initial boot sequence, the victim machine first attempts to boot from the hard disk, followed by the external Ghost media. The Ghost media then loads a special Ghost boot program that installs a pre-made full disk image onto the victim machine's hard drive. This pre-made image includes an unpatched version of Windows XP (with appropriate hardware drivers), and when the disk image is done loading, the victim machine automatically restarts. This pre-configured version of Windows XP also runs a special management program to collect and execute the next malware binary from the server. After execution, this next malware binary is left to run for several minutes, long enough for the server to collect a valid network fingerprint, f_{net}, of the binary. After this short time period has expired, the management program deletes the master boot record of the victim hard drive before restarting the victim machine. Thus, when the victim machine boots up again, the boot process defaults to the external media disk, thus installing (again, via Ghost) a new clean Windows instance for which to run the next binary. In this fashion, fingerprints are created for each previously uncooperative binary. We may then compare these fingerprints with those previously collected in the graybox testing. If the new fingerprints show interesting network behavior, we can safely assume that the binaries do in fact make use of virtualization detection.

Of course, the reader may be wondering why we did not choose to run all of the binaries in this fashion. There are several answers. First, loading a pre-made image onto a hard-disk is slower than simply copying a virtual machine disk-image file. With our particular hardware, we have observed that copying a virtual machine disk-image file is about four times faster, saving about ten minutes per binary (a significant amount of time, given a large number of binaries to process). In addition, using virtual machines provides much greater control over the testing environment. For example, malware may cause unpredicted instabilities on the victim machine, thus preventing the management script from restarting the machine. If the software is running in a virtualized environment, such a failure of the management script can easily be circumvented by stopping the virtualization software. However, if the software is directly running on hardware, any failure cannot be addressed automatically. Instead, these failures would require physical intervention from a test operator. Thus, we run this special stage of analysis on only the minority of binaries that seem to defy regular graybox analysis. We discuss the results of this special analysis in Section 5.1.

Chapter 4

Wild Botnet Tracking

"In 2006, the attackers want to pay the rent. They don't want to write a worm that destroys your hardware. They want to assimilate your computers and use them to make money."

- *Mike Danseglio, Microsoft Security*

4.1 IRC Tracking Robot

Ultimately, our goal is to observe botnets as they behave in the wild. In order to accomplish this feat, we need to gain an inside perspective in the form of our own agent on the inside of a botnet. Our IRC tracking robot serves this purpose. The robot itself is written in Perl and is extremely lightweight. We have run dozens of robot instances simultaneously on a single Pentium IV with no noticeable degradation in system performance. In this sense the robot is very similar in spirit to a lightweight malware collection tool like Mwcollect [1]. While Mwcollect emulates a personal computer for the purpose of collecting nefarious shellcodes and binaries, our robot emulates a personal computer in order to observe the actions of botmasters and their minions. Of course, our robot aims to emulate only the command acknowledgments within the IRC server. At no point does our robot take part in any scanning, denial of service, or exploitation activity. Regardless, the robot must overcome many challenges before it can ensure its place in a busy botnet.

Challenge 4.1: *Robots must be rapidly deployable in order to observe new botnet instances.*

In order to achieve rapid deployment, the robot uses a simple XML configuration file to determine how to connect to a particular `C&C` server. In all other facets, robot instances are identical. The XML file is simple enough to be hand-written with relative ease, but the primary source of configuration files in our infrastructure has been the output of the graybox testing pipeline. The pipeline generates these configuration files automatically, so robots can be prepared for deployment with an absolute minimum of human intervention. This configuration file can specify two primary modes of operation, silent or template-driven. In silent mode, the robot has a relatively easy job[1]. In this mode of operation, the robot merely connects to an IRC server and joins the specified channel. Once in the channel, the robot responds to `PING` requests from the server to ensure that the IRC-level connection is kept alive. Other than answering `PING` requests, the robot sends no acknowledgments to the server; however, the robot still uses the configuration file to choose an appropriate IRC nickname and IRC username. Therefore, an inattentive botmaster is unlikely to notice the robot among his other minions. The botmaster is particularly unlikely to notice the robot when the botnet is particularly large; however, in smaller botnets, a botmaster is much more likely to notice a silent member of his network. This issue is complicated by the fact that the IRC protocol allows botmasters to easily send commands to individual bots, rather than to everyone in a particular channel, and these targeted messages may or may not be visible to all of the other members in the channel. The case where all channel members may observe targeted messages is, of course, the preferred case for our robot, as it allows for more complete observation of botmasters' actions. Figure 4.1 illustrates the subtle differences between a publicly viewable channel message, a private targeted message, and a publicly viewable targeted message.

Challenge 4.2: *Targeted messages require robots to respond promptly and correctly in order to avoid detection.*

[1]Indeed, we are not the first to implement a silent IRC-tracking drone [14].

```
//Message to the channel will be sent to all bots within that channel
PRIVMSG #atta :.netinfo

//Message to a particular user will be sent only to that member
PRIVMSG USA|154276039:.netinfo

//Message sent to the channel with a bot target will be sent to all
//users, but only answered by the targeted bot
PRIVMSG #atta :USA|154276039 .netinfo
```

Figure 4.1: Three different command heuristics for sending commands to bots. The last two commands are targeted to the bot with nickname `USA|137048`, but the first and last are received by all members of the channel.

Unfortunately, a silent-running robot inherently fails to meet this challenge. The limitation of a silent running drone is that it immediately stands out as suspicious when it receives a specially-addressed message from the botmaster. The offending behavior is simply that the robot offers no response where a response is expected. To prevent this behavior from arousing suspicion, while in silent mode, if the robot receives a private message or is the focus of a targeted message from the botnet operator, the robot immediately imitates a connection failure. This feigned connection failure is implemented by simply failing to respond to the IRC server's `PING` commands. Generally, after two `PING` commands are not acknowledged, the server will remove the offending client by severing the underlying TCP connection and immediately thereafter reporting a timeout. `PING` timeouts are not uncommon among even well-behaved (real) botnet victims, presumably due to the amount of scanning being performed by the victims. As such, we believe our timeout strategy has proven relatively successful, but it still has several drawbacks. First and foremost, if the robot disconnects from the `C&C` server, it is not monitoring the behavior of the botnet when the botmaster is at his most active. Furthermore, impatient botmasters might configure their IRC servers to ban the IP address of the robot after a connection timeout occurs. Even with the many IP addresses at our disposal, a suspicious botmaster could ban our entire subnet, thus completely denying us the ability to continue data collection.

To avoid the drawbacks of the disconnect strategy, we believe the IRC tracking robot serves us best when operating in its second (and more complex) mode of operation: template-driven. In this mode of operation, the robot responds to commands as specified by the graybox-generated template. By acknowledging botmaster commands, we believe the the robot is more likely to go unnoticed when the botmaster is at his busiest. Thus, our robot may stay on the IRC channel when the botnet is exhibiting its most interesting behavior, and collect data accordingly. For example, denial-of-service attacks, click frauding, or bot-upgrading activities are all presumably clustered within these active periods. Furthermore, a template-driven robot stands a much better chance of going unnoticed in a smaller botnet. The robot then may stay in the channel as the botnet grows, thereby recording the botnet's entire life span.

The template itself is represented as a simple file-directory structure. When the robot receives a command (as defined by the previously discussed delimiter character), it searches the directory for a file of the same name as the first token of the command. For example, let us assume a simple (non-stateful) command `.visit http://website.com/` was received by the robot. This reception would prompt the robot to search within the template directory for a file with the prefix "visit". If no such file is found, the robot makes note of the template replacement failure before initiating a feigned timeout. Otherwise, the robot reads in the particular file and replaces any special terms included within the text. As discussed in the previous section, these special characters were inserted by the query engine and can include anything from arguments of the botmaster's command to the current IP address of the robot. However, most responses are more complex.

Challenge 4.3: *Robots must be aware of their surroundings and reply to botmasters with live network information.*

Templates merely provide a framework, and the robot must appropriately fill in applicable fields, such as the current server address or IRC channel name, when responding to specific commands. Most of this information is readily available to the robot from the configuration file; however, some information must be inferred from observed network traffic. For instance, the robot is unaware of its outside IP address due to the network address translation taking place at the gateway. As a result, the robot may not know how to properly acknowledge a botmaster command which references its IP address. To counter this difficulty, the robot will attempt to infer its IP address from the welcome message from the

C&C server itself, as illustrated in Figure 4.2. Of course, some (the minority) of the IRC servers do not offer such a helpful welcome message. If the outside address cannot easily be inferred via messages from the C&C server, the robot chooses an address uniformly at random from the known space used by the gateway[2].

```
//Welcome message announces outside IP address
botnet.net 001 USA|154276039
:Welcome to the botnet.net IRC Network USA|154276039!~rsdjcvtgse@---.---.35.67

//Template entry for .netinfo command
[NETINFO]: [Type]: LAN (LAN Connection).
[IP Address]: **EXT-IP**. [Hostname]: **EXT-IP**.

//Tracker response to the .netinfo command on live channel
PRIVMSG #channel
:[NETINFO]: [Type]: LAN (LAN Connection).
[IP Address]: ---.---.35.67. [Hostname]: ---.---.35.67.
```

Figure 4.2: After observing its IP address in the welcome message, the robot performs a replacement on the template entries before responding on the channel. The actual IP address is obfuscated from the figure.

Challenge 4.4: *The robot must remember past interactions with the botmaster in order to appropriately acknowledge the latest commands.*

In many cases the robot may respond to the botmasters' commands after only a simple replacement of a template file. Unfortunately, as mentioned in Chapter 3, botnet interactions are not always so simple. Bot responses are often *s*tateful and require deeper processing on the part of the robot. Finding the right balance between a robot that is complex enough to correctly mimic stateful acknowledgments and a robot that is simple enough to be maintainable is an ongoing challenge. For the time being, we address this challenge by ensuring that the most frequent stateful botmaster commands are handled correctly. When the robot receives one of these stateful commands, it should discover that the file listing the correct acknowledgment within the template directory will have a simple state-change identifier as a suffix. For example, the file "keylog.OFF.ON" contains the text used in responding to a `.keylog` command when the state transitions from "off" to "on".

[2]In the future, we plan to allow the gateway to communicate this outside IP address to the robot through a secondary channel, but we have not yet implemented this feature.

Upon receipt of this command, the robot modifies a small variable within its own program, so as to look for the correct template file the next time the `.keylog` command is issued. The only exception to this very basic "off" vs. "on" state system is a simple thread count. We have observed a very few commands (all relating to scanning or launching denial of service attacks), that spawn multiple threads. In these few cases, the robot keeps track of a thread count variable in order to correctly respond to follow-up commands. The logic for these exceptions is currently kept in the robot code; however, the discovery of more of these commands would warrant the development of a more generic solution.

Challenge 4.5: *The robot may encounter never-before-seen commands.*

Botmaster commands which have never before been seen by any of our robots present a particularly pernicious problem. Currently, the robot handles these commands in the same way in which a silent drone would – it simply feigns a connection failure. However, we are currently examining more sophisticated solutions to this problem. With only a few additional physical computers, the IRC robots could be supplemented with a small pool of "emergency" honeypots. When the robot encountered a command for which he had no known response, the robot could pass his entire transaction log to one of these emergency honeypots along with the associated botnet binary. An assisting IRC agent could then replay the entire transaction log to the honeypot in order to discover the appropriate command response. This response could then be forwarded to the robot for use on the live channel. This "catch up" type of transaction system is similar to the idea behind RolePlayer [12, 41], but here the goal is to stay unnoticed on a botnet rather than to efficiently collect a malicious exploit payload. We are currently exploring whether or not this type of mechanism can be implemented with a delay small enough to satisfy an impatient botmaster. Indeed, abundant delay could raise botmasters' suspicions. In the meantime, unknown commands are marked for addition to the query agent's graybox-testing repertoire, and future templates will thus include responses to previously non-handled commands. However, even with a complete template, the robot must overcome additional challenges in order to effectively respond.

Challenge 4.6: *The robot must not only respond with the correct acknowledgment, but it must respond with a realistic latency.*

In practice, some botmaster commands evoke instant replies. Others commands may require involved offline tasks, and thus these types of commands may require any arbitrary delay before a response can be returned. Of course, the botmaster is acutely aware of how long it takes the robot to respond. Thus, responding at an incorrect speed may endanger the robot's stealth. As an example, consider if a bot claimed to complete the download of a very large file after a delay of only a fraction of a second – any attentive botmaster should be surprised. Indeed, such behavior may prompt an unwanted investigation on the part of the botmaster. To address this challenge, the robot supplements the template with a few very simple timing rules when responding. These rules deal with simple issues such as the insertion of artificial delay when responding to any command which is presumed to involve downloading or other network-intensive activity. Due to the limited number of these rules, they are also hard-coded into the robot, and in general, keeping these rules up-to-date in the tracking robot is not difficult – all but one or two of these rules requires nothing more than a general categorization of a given command. This categorization, although originally hand-made, requires no additional effort, as it is already required by the query agent to perform graybox testing on the malware binaries.

Challenge 4.7: *The robot must interact with the botmaster's* `C&C` *(IRC) server in ways consistent with the botmasters' commands.*

For better or for worse, the botmaster is plainly aware of the state of the robot: the chat room the robot is a member of, the nickname the robot is using, and so on. Furthermore, the botmaster believes he should be in control of these factors, and particular commands may intend to manipulate this state of the robot on the `C&C` server. For example, a botmaster can order a victim to: disconnect from the server, restart the victim computer, join a different channel within the IRC server, or perhaps even connect to a different server altogether. (We generally refer to these commands as invoking "temporary" movement, as the functionality is supported by the bot software without changing the underlying botnet executable.) Properly obeying these command is critical. As an example, consider a botmaster who commands his minions to join a different channel. Any bot not obeying this command would immediately stand out as a lone member of the original channel, thus becoming the target of an unwanted investigation. To avoid such a scenario, the robot maintains rules for the handling of this small set of action-causing commands. Similar to the rules for handling response latency, these rules are hardcoded. Fortunately, we have thus far noticed very few of these commands (fewer than ten), and all of their associated actions are relatively simple, with the exception of the command to join a different server. In this latter case, the robot generates a new configuration file on-the-fly and spawns a new robot instance to begin tracking the migrated botnet. We point out that the robot performs no actions in response to cloning-style commands. Cloning involves connecting a new IRC instance to a new IRC server in addition to the original instance remaining connected to the original `C&C` network. In general, cloning commands may be used as a denial-of-service attack against another IRC server, and we do not wish to partake in such activities. We admit the possibility that our precautions here may cause a loss of interesting data. For example, botmasters could potentially use cloning as some form of bot leasing. In these cases, cloned victims may only support a subset of the original commands so as to ensure their ultimate allegiance to their original owner. Unfortunately, as of yet we have been unable to confirm the leasing hypothesis.

Challenge 4.8: *A wily botmaster could distinguish the robot from a legitimate victim by exploiting the robot's unwillingness to participate in malicious activity.*

Ultimately, this challenge cannot be overcome without abandoning our ethics – we cannot contribute to the mission of the botmaster. In truth, a botmaster's detection algorithm could be quite simple. For instance, the botmaster could send our robot a private command to scan a particular IP address – an IP address controlled by the botmaster himself. After receiving an acknowledgment from our robot over the `C&C` server, the botmaster would not observe the anticipated scan. The scan could even be performed on the same port as the `C&C` connection, thus ruling out the possibility that we are a willing victim behind a stubborn firewall. Fortunately, performing this kind of analysis on each and every victim within a botnet would prove to be extremely time-consuming, even on a very small botnet. Consequently, we believe responding only within the bounds of the IRC protocol is more than sufficient to hide among the bot masses.

Challenge 4.9: *Botnets may migrate to new* C&C *servers or suddenly change their dialects. The robot must continue to behave appropriately in the face of this dynamism.*

Botnets are infinitely flexible, thanks to one simple fact: botmasters can command their minions to download an entirely new package of bot software. Updated software could mean anything: they may add new features, change the dialect of the bot, or these new binaries may point the bots to an entirely new C&C server. If the updated binary includes an address for a new C&C server, we refer to such an update as a permanent move. In the case of one of these moves, a lightweight program such as our robot has no means to extrapolate the new IRC properties. As a result, the robot downloads the update and submits the executable to the graybox testing pipeline. In practice, any download is performed only after consulting a central download timer to ensure that no particular server bears an overburden of traffic due to our robot's downloads – we must take care not to unwittingly participate in a denial-of-service attack. The central downloader then transfers the binary into the staging area for the graybox testing pipeline. Once graybox testing on the binary has been completed, a new robot instance can be restarted via the automatically generated configuration file and template. At the moment, a human operator is required to approve the new configuration file and template before the new robot instance instance is started; however, this human confirmation takes only a few seconds and could easily be disabled if more throughput were needed. In this fashion, our infrastructure may track botnets even through movements driven by entirely new binaries.

As a quick aside, we also note that some botnet migration behavior cannot easily be followed by our robot using current techniques. Specifically, the robot cannot handle a rapid succession of certain bot movements due to the delay induced by the graybox testing pipeline. As an example, consider a set of botnet servers $S = \{S_1, S_2, S_3\}$ with binaries $B = \{B_1, B_2, B_3\}$ directing the victim to the respective C&C servers of S. Then, suppose an initial infection occurs with B_1, directing the victim to S_1. A botmaster may then issue a command to download a new binary, B_2, thus directing the victim to S_2. Once on S_2, the botmaster may *immediately* force a download of B_3 to force a redirect to S_3. The botmaster may then quickly shutdown S_2. In this case, a faithful victim will follow the chain of downloads to the final (and presumably more interesting) `C&C` server. However, our robot may not arrive at S_2 before S_2 has been shut down. Thus, the robot will never reach S_3, even though S_3 harbors the real botnet activity. We suspect more savvy botmasters use this technique of chaining to result in especially elusive botnets, and we believe one possible solution would be to send binaries directly to waiting honeypots to ensure that the new channels were visited immediately. Of course, such a solution would require additional hardware resources to support the honeypots.

Challenge 4.10: *Robots must be deployed constantly to ensure long-term persistent coverage of botnets, but should not be deployed redundantly.*

In order to ensure maximum botnet coverage, all robots are coordinated by a central management script to avoid two scenarios: 1) multiple robot instances are observing a similar botnet 2) no robot is currently observing a discovered botnet. The first scenario is possible because the collection infrastructure may capture a seemingly new binary for a botnet already under observation. To avoid this problem, all robots "check in" with a common persistent data structure. This data structure is simply a list of records, $f_{track} = \langle \texttt{SERVER}, \texttt{CHAN}, \texttt{TIME}, \texttt{CONFIG} \rangle$, representing the IP or DNS address of a botnet, the relevant IRC channel, the time of the last collection attempt, and the configuration used when the connection attempt was made. When a new robot instance is started, it will check the data structure for the time associated with botnet it intends to join. If the time listed with the given botnet is recent (less than one day old), the tracker simply cancels the connection attempts and cleanly exits. If the time is not recent, or no entry is found in the data structure, the robot marks the data structure with the relevant botnet information and the current timestamp before attempting to connect. The second scenario (a known botnet goes unobserved) may occur when a botnet goes offline, only to come back online a few days later. In these cases, the investigating robot may have long given up attempts to reconnect. To discover these reemerging botnets, we run a script daily to examine all f_{track} entries listed in our data structure. If any botnet is found with a stale time stamp, the script executes a new tracker instance to check the status of the given botnet. This new tracker will then update the data structure with the current time for this botnet, as well as resume tracking of the botnet if the associated `C&C` server accepts connection attempts. This level of coordination ensures that bot tracking is complete but not redundant.

Also, regardless of whether the robot is silent or is instead template-driven, the robot periodically disconnects from its target server and reconnects with a new nickname (and new IP address). This reconnection ensures that botmasters do not observe a consistent (and perhaps memorable) victim connected to their network. In addition, the reconnection allows the robot to observe any server welcome messages set by the botmaster. These welcome messages often contain interesting statistics about the server, such as how long the server has been running. Currently, the robot remains connected for a period of uniformly random time between 90 and 150 minutes. In order to minimize the loss of interesting traffic, the robot then reconnects after only a few minutes of sleep. This periodic behavior continues until the `C&C` server can no longer be contacted, or until a preset number of reconnects have occurred.

Challenge 4.11: *The information observable by the robot can be severely hampered by conscientious botmasters.*

Regardless of how stealthfully the robot tracks any given botnet, the amount of data available to a botnet insider may vary. More precisely, the data received by any given member of the botnet is subject to the configuration of the botmaster's `C&C` (IRC) server. Even if the botnet is being hosted by an unwitting public server, the botmaster may still manipulate the amount of information sent by setting the IRC modes of the channel. Ideally, the `C&C` channel will broadcast all kinds of interesting information to every member (including our robot). This information can include: a listing of the current bots in the channel (and their nicknames), notification of bots joining or leaving the channel, the IP addresses of the connected bots, and the responses given by all of the bots to any given botmaster command. However, at any point, the botmaster may disable the broadcast of any of this information. Support for this silencing is built directly into most out-of-the-box

IRC servers, and requires no custom modification from botmasters. In many cases, we can observe a botmaster changing these modes, thereby modifying the types of information that is sent to the robot mid-session. At worst, a botmaster will run a tight ship, and the `C&C` network will broadcast none of the afore listed information. In this case, the robot will receive only the commands from the botmaster, and for botnets of this type, even the most crucial botnet statistic – the number of currently online bots – is a complete mystery. Unfortunately, these worst-case botnets are not difficult to configure. To achieve the appropriate silence, a botmaster has several options; he may set the channel to an "auditorium" mode (set with the `MODE +u` command) to prevent the list of members being rebroadcast. Alternatively, a botmaster could moderate his channel in such a way that only those with "voice" may send messages to the channel (set via `MODE +m`).

In addition to demanding silence on their network, botmasters often mask important pieces of information, further limiting the breadth of view of a botnet insider such as our robot. A particularly common practice comes built-in to UnrealIRC and is referred to as "cloaking". Cloaking masks the hostname or IP address of channel participants such that their real identity cannot be determined through any normal IRC traffic. Bots generally are programmed to enable cloaking themselves by issuing a `MODE +x` command to the IRC server. Once this mode is set, the origin of the client machine is replaced by an MD5 sum of the original hostname concatenated with predefined (secret) cloaking keys, such that $cloakIP = \text{MD5}(key||IP)$. We know of no way to use this cloaked information in order to determine the original IP address of botnet members. Thus, due to the commonality of this cloaking practice, botnet tracking drones and robots typically fail to gather any information pertaining to botnet demographics. This failure is rather unfortunate, and has led researchers to heuristics such as timezone modeling to infer demographics of a given botnet [13]. We believe our, external, approach of using DNS cache probes to determine botnet demographics provides the best options for overcoming this problem of cloaking.

4.2 DNS Cache Probing

DNS cache probing (sometimes referred to as DNS cache "snooping") was first brought into the spotlight after the Sony rootkit debacle [22], although the first public discussion of DNS snooping was proposed somewhat earlier [16]. DNS cache probing allows us to generate a rough estimate as to the size, growth, and victim distribution of the *footprint* of a botnet with a DNS-addressable `C&C` server (or servers). We use the term footprint to refer to the number of end-hosts infected by a botnet executable, and this number can potentially be substantially larger than the number of victims currently connected to the IRC server. This discrepancy is caused by the fact that botmasters may not have the physical resources to host legions upon legions of bots. We also note that we are unclear as to which number previous researchers have used to describe the "size" of a botnet. Given that some researchers have claimed to encounter botnets of extremely large sizes, we suspect that the idea of a botnet size has often been associated with what we could consider to be the footprint. Regardless, the footprint may remain much larger than the current active bot population simply because botmasters' IRC servers often limit (in software) the number of clients they support. In fact, bots may be denied access to the server because of this anti-crowding technique. Clients may also have difficulty contacting their master due to network traffic, as repeated scanning activity can cause much network instability.

A detailed discussion of the exact methodology used by our DNS cache proper can be found in previous work by Rajab *et al.* [32]. In general, DNS cache probing and an insider robot complement each other well in terms of the data they can gather. In the minority of cases where a hardcoded IP address is used in lieu of a DNS name or the botmaster hosts his network on a public IRC server, our IRC robot can be used to monitor a network from the inside. In other cases where IRC server sends little or no information to channel members, DNS snooping may be used to infer general botnet properties. DNS cache snooping has proven particularly useful for determining victim demographics in the face of the aforementioned cloaking practice, and DNS snooping can also help continue the monitoring of a particular botnet in cases where the botmaster suddenly activates the silencing features so dreaded by the robot. Overall, the supplementation of DNS cache snooping to the IRC robot seems crucial to understanding the highest breadth of botnet variations.

Chapter 5

Results and Analysis

"/*
Doom Scanner starts here scans for backdoor left behind by MyDoom and tries to exploit it by sending \x85\x13\x3C\x9E\xA2+EXE File Contents to the remote host and hoping it happily executes it.
*/"

- *Source Code for Agobot3*

5.1 Binary Taxonomy

As a result of our automated graybox analysis, we generated a database of dynamic fingerprints from our collected malware. At the time of this writing, we have generated fingerprints for 1172 malicious binaries, distinct by their cryptographic hashes. Of these, 879 were confirmed to be IRC-related botnet binaries. In other words, 879 of the binaries actively participated in an IRC-style transaction with our modified server during the graybox testing process. Of course, we must be quick to point out that a cryptographic hash is not an ideal mechanism by which to determine malware distinctness. These hashes can differ for completely trivial reasons. For example, malware may use forms of *polymorphism* to change its underlying source code while maintaining identical functionality. Alternatively, two binaries may point to the same `C&C` server, but be slightly different versions of a similar code base. (One binary could be an upgrade from the other binary.) As consistent with the rest of this work, we do not examine the actual binary to determine why seemingly unique executables exhibit functionally equivalent network fingerprints. However, we make extensive use of the network and IRC fingerprints (f_{net} and f_{irc}) to examine the properties of the malware binaries, and the master log file marks the most direct and simplest means to examine properties of the malware collected by our infrastructure.

The Master Log. The master log is a simple flat file that is appended with a record whenever a binary is processed via graybox analysis. The record, $f_{rec} = \langle$`ID`, `PORTS`, `DNS/IP`, `NICK`, `CHAN`, `TALK`, `SCAN`$\rangle$, consists simply of the binary identifier (b_{id}), ports the binary attempts to contact, DNS address or outside IP addresses the binary attempts to contact, the bot-selected IRC nickname, any IRC channels the binary joins, whether or not we were successful in evoking IRC acknowledgments from the binary, and whether not the binary exhibited scanning behavior (see also Section 3.1). Table 5.1 shows a small logical snippet of the master log file. The snippet represents binaries gathered throughout a 48 hour collection period over our locally monitored Internet space.

b_{id}	**Ports**	**DNS/IP**	**NICK**	**CHAN**	**TALK**	**SCAN**
id 1	445:80	-.ru	na	na	no	true
id 2	12347	iso.-.net	klzxpls	#–#	yes	false
id 3	12347	iso.-.net	drbdyr	#–#	yes	false
id 4	139:6659:80:8888	utenti.-.it	DMPOP	#GhostBot	no	true
id 5	na	na	na	na	no	false
id 6	4000	DTC.-.NU	USA|19942	#UB3R	yes	false
id 7	1337	-.-.68.206	USA|07347	#bot#	yes	false
id 8	1023:6556	0x80.-.org	eyhet	#26#	no	false
id 9	7000	home.-.com	R-6684453	#n	yes	false
id 10	139:6659:80:8888	-.-.108.243	CJCGZ	#GhostBot	no	true
id 11	445:80	-.ru	na	na	no	true

Table 5.1: Portion of the master log, in table form. Full, b_{id} values are removed for brevity, and we have obfuscated the server addresses. "na" represents a null field. Regularly explained behavior, such as Windows networking probes or benign DNS lookup requests are not included in the DNS or ports fields.

Logically identical botnets are easily visible in Table 5.1. For example, f_{rec} pairs (1,11), (2,3), (4,10) all seem to represent logically identical binaries. The pair (4,10) is particularly interesting, as one version seems to use DNS to locate its server, while the other does not.

This may be indicative of a botnet evolving from a simple IP based `C&C` to one which uses DNS. We also point the reader to the fifth record in the table. This entry is indicative of a binary for which no interesting network behavior was observed when run on a virtualized Windows instance.

As a whole, this summary file gives us a very good indicator as to the true uniqueness of the different binaries we captured. For instance, if we assume that particular IRC channels bijectively map to logical botnets, we discover that our 879 distinct botnet binaries represent only 335 logical botnets. However, if we instead consider a botnet to be unique by its server address, then our collection represents an even fewer 259 logical botnets. Unfortunately, we do not believe any other literature has discussed these means by which to classify botnet uniqueness. At best, previous work has classified botnet binaries by the underlying framework used to build the software. For example, malware researchers have gone to great lengths to cluster binaries into logical families with labels such as `Win32/Rbot`, `Win32/Wootbot`, `Win32/Mywife.E`, and may not concern themselves with the logical `C&C` server used by the binaries. As a result, we believe that the `C&C` mechanism used by these binaries has been underrepresented in these classifications. However, for sake of simplicity, throughout the remainder of this paper, unless otherwise specified, we will refer to two binaries as "unique" if and only if the cryptographic hash sums of the two binaries are different.

The master log also gives an idea of the prevalence of IRC versus non-IRC botnets, as well as the prevalence of certain botnet scanning behaviors. IRC botnets are easily distinguishable once they successfully complete an IRC handshake to what they believe to be an outside server (in actuality our graybox server), while non-IRC botnets exhibit a slow-rate connection attempt to a particular outside DNS name or outside IP address without a successful IRC handshake. Most of our binaries represent the former: IRC botnets. To describe the automatic scanning characteristics of a botnet binary (as opposed to the scanning characteristics a botnet may undertake as a direct results of commands from the botmaster), we have partitioned botnet executables into two scanning variants: Type I and Type II. We define Type I botnets as those botnets whose binaries automatically begin some sort of scanning behavior *without* any direct contact to their botmaster. Generally, this scanning is not targeted toward any specific subnet, but rather done in either a uniform or local-subnet-favoring nonuniform fashion. Type I botnet executables are easily observed in the graybox testing process, and the master log shows that 275 out of 879 confirmed IRC botnet executables (considering uniqueness by cryptographic hash sum) were of this variety. These types of botnets presumably spread more like earlier types of worms, and can presumably be discovered more easily due to the more conspicuous scanning behavior. To date, we have not discovered a Type I botnet binary with a still-active `C&C` server. We presume that the control center was long ago discovered and removed by appropriate authorities, but the malware still continues to spread in a worm-like fashion. Of course, we must then ask the question: can this remnant behavior still be called a botnet, or do Type I binaries merely represent a new mutation of self-propagating worms? We leave these questions for future debate.

Unlike their Type I counterparts, Type II botnets do not scan automatically; they instead merely attempt to contact their master. We are more interested in this style of botnet, primarily because unlike their more worm-ish counterparts, these botnets cannot be modeled using standard techniques [6, 31]. These types of botnets also have the potential to grow slowly and remain active for a longer period of time. Type II botnets enjoy these characteristics thanks to their very selective, botmaster-dependent, scanning behavior, thus allowing them to avoid the gaze of Internet telescopes. These bots may also scan for only a small portion of their life, thereby generating less traffic to fill up firewall and IDS logs. According to the master log, 604 out of 879 confirmed IRC botnet binaries were Type II. The remaining 293 non-IRC binaries of the 1172 appear to be either simple scanning worms, botnets using non-IRC protocols (such as HTTP), or are of unknown classification. We classify a binary as a worm if it exhibits scanning behavior on known exploitable ports (135, 139, 445) without any attempt to contact a non-local IP address on a potential `C&C` port. The unknown classification refers to binaries with a network fingerprint identical to that of a clean Windows instance. In these cases, the binary may have failed to run on the virtualized Windows instance due to some level of virtual machine detection mechanism. Table 5.2 presents a rough summary of all binaries listed in the master log file with associated counts for each binary type over the different uniqueness metrics.

	Uniqueness Metric		
	Hash (MD5)	**Server+Channel**	**Server**
• All binaries	1172	n/a	n/a
• IRC Botnet	879	335	259
IRC Botnet Type I	275	35	21
IRC Botnet Type II	604	300	238
• Not-IRC Botnet	174	23	23
Not-IRC Botnet Type I	153	12	12
Not-IRC Botnet Type II	21	11	11
• Worm	9	n/a	n/a
• Unknown	110	n/a	n/a

Table 5.2: Breakdown of all collected binaries over three different uniqueness metrics: Hash (MD5), `C&C` server with IRC channel, `C&C` server only. Hash sums seem to do a particularly poor job of uniquely classifying Type I (scanning) botnet binaries.

A more complete breakdown of all port activity observed from the malware can also be extracted from the master log, and Figure 5.1 illustrates the most frequently contacted ports for Type I botnets. The existence of HTTP-based Type I botnets is clearly visible from connection attempts on port 80, while activity over ports 139, 445, and 3127 presumably represents scanning exploitation attempts via DCOM vulnerabilities, LSASS vulnerabilities, and the MYDOOM backdoor, respectively. We also point out that among Type II IRC botnets, the standard IRC ports (6667 or 7000) were used only 46% of the time. In the majority of instances, one of dozens of other nonstandard ports were used for `C&C` traffic.

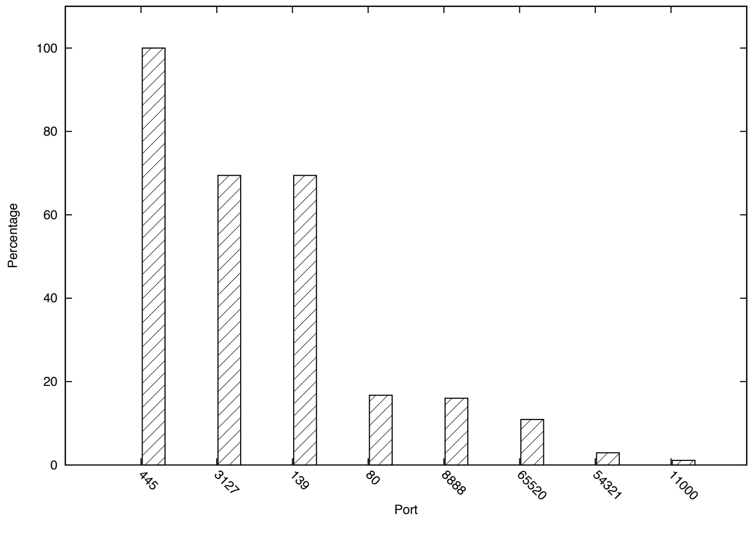


Figure 5.1: Histogram of the ports most commonly contacted by Type I botnets.

Finally, we can use simple information from the master log to infer some information about the prevalence of certain botnet architectures. For example, if we assume that different botnet architectures chose bot nicknames by distinct rules, we can map these architectures over time by observing the different IRC nickname patterns. For example, the master log shows us that common heuristics for nickname selection include: a fixed prefix followed by random numbers, random letters, or identifying characteristics of that bot (representing the computer's operating system or the country of origin). Figure 5.2 shows a chart of these different naming schemes over time. The chart also demonstrates that we generally observed few distinct naming heuristics. This lack of diversity in naming schemes makes graybox testing significantly easier, as for any given binary we must only determine which of these schemes is in use. Once this convention has been isolated, the robot may exploit the scheme to repeatedly connect to the `C&C` server with a distinct, but valid, nickname. The six naming conventions referenced in Figure 5.2, ⟨`TOKNUM`, `TOKLET`, `ALLLET`, `POONUM`, `USANUM`, `OTHER`⟩, representing a botnet-specific token followed by random numbers (`XXX|12334`), a botnet specific identifier followed by random letters (`YYY-GZZXRIJ`), random letters (`FXIQMJ`), a botnet specific token with random numbers wrapped with brackets (`[BOT|199433]`), a three-letter country code identifier[1] with random numbers (`USA|89234`), and any other naming scheme (`USA|123433|WINXP`), respectively. In general, the figure demonstrates the very consistent mix of each of the prevalent naming schemes throughout the collection period. In other words, since we did not observe any particular scheme with accelerated frequency, we did not collect any evidence to imply the adoption of a newer botnet architecture. Instead, botmasters may be upgrading their existing architectures with newer modules and features.

Feature Analysis via Template Examination. Unfortunately, the master log cannot shed light onto some of the deeper attributes of every botnet. For richer feature analysis, we must instead turn to the templates generated by the query agent. In addition to feeding a responsive robot, the templates can be used to distinguish differences in major botnet features. We were able to extract templates from 315 out of 604 confirmed Type II IRC

[1]We believe that the three letter country code is based on the international settings on the victims' Windows installations, rather than any kind of smart self-geolocating on the part of the botnet or bot software.

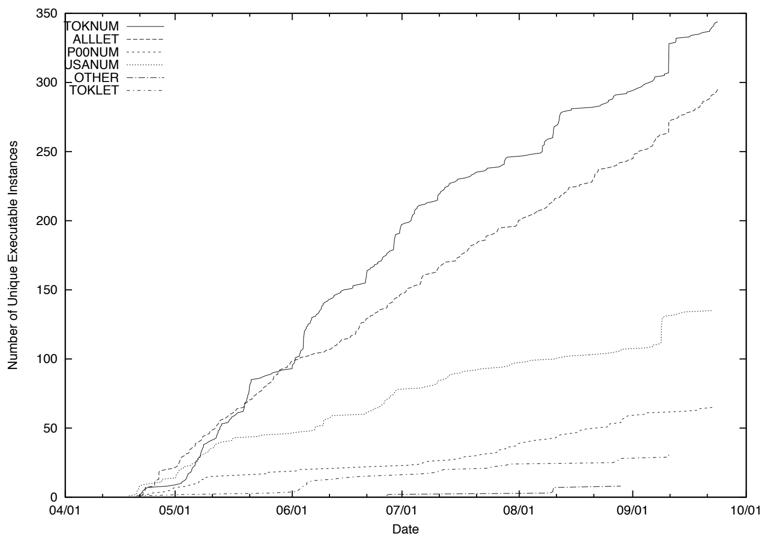


Figure 5.2: Measured cumulatively, the naming convention used by different IRC botnet executables collected by our infrastructure. Date is indicative of the time of collection of the binary from either PlanetLab our own our local space.

binaries. (The Type I IRC botnets we discovered were markedly less interesting, and less unique, as can be observed by Table 5.2.) These templates inform us of many of the capabilities of the binaries. For example, the templates generally include information on what kind of exploit modules are contained within a binary. Furthermore, the templates may also contain information regarding particular defense mechanisms built into a binary; these defense mechanisms vary from anti-virus-disrupting software to registry monitors to firewall disablers. In some cases, the malware actually patches vulnerabilities so as to prevent infection from other, foreign, malware variants.

Figure 5.3 contains a cumulative distribution function of the number of exploit modules contained in a given binary among those binaries which reply to a `.scanstat` command variant (273 binaries in total). We note that many binaries contain exploit modules with very similar names (eg. DCOM vs. DCOM135), but in these cases we still assume each name represents a unique exploit module. By far, the most popular exploits seem to be variants of ASN, DCOM, and LSASS vulnerabilities [9, 10, 11]. Sadly, patches preventing these exploits have been available for years.

Table 5.3 shows a list of common botnet features as inferred from responses to threading commands. As with the counts of exploit modules, we quickly note that percentages here

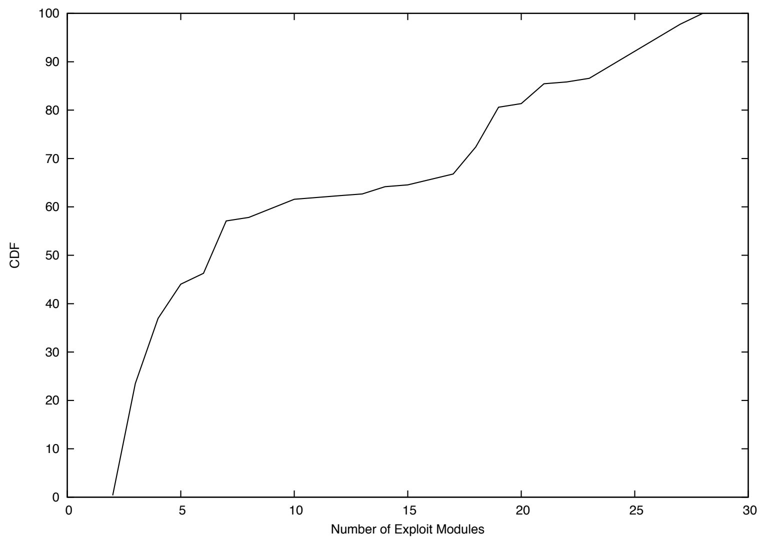


Figure 5.3: CDF of the number of (scanning) exploit vectors for malware binaries.

represent only those bot executables which responded to a known threading command (309 binaries in total). Including the feature-related statistics for all bot malware would require a more involved process of binary analysis which we do not employ. Regardless, our data suggests that some features are quite common among all bot software. We suspect that many botware packages come bundled with various software packages, only requiring a botmaster to enabled them in a header file before compilation of the malware. Indeed, some of the features are relatively impressive: the AV/FW Killer thread attempts to disable anti-virus and firewall software, while the system security monitor halts or patches vulnerable services (to prevent infection from another botnet). Other features include a FTP server, TFTP [35] server, or even an Identd [19] server (to allow IRC connections to certain public servers). We also observed a registry monitor to prevent the removal of bot software. Finally, a "ConnectBack" backdoor presumably opens a port to provide an alternative botmaster-to-victim communication channel. In some of our earlier work [32], we did not observe any instances of this "ConnectBack" feature, and we therefore believe it is a prime example of genuine evolution of botnet software.

AV analysis. We also subjected all of our malicious binaries to anti-virus software. We utilized both the open source solution, ClamAV [7], and the commercial solution, Norton's anti-virus software [37]. The results were relatively encouraging, although we point out that

Utility Software Thread	**Frequency (%)**
FTP Server	86
TFTP Server	85
AV/FW Killer	55
System Security Monitor	49
Registry Monitor	41
Identd Server	31
ConnectBack Backdoor	9

Table 5.3: The percentage of bots that launched (by default) the respective services.

both variants of our anti-virus software were updated at the end of the collection period. We conjecture that both software packages would suffer a lower classification rate without the benefit of after-the-fact virus definition updates.

Regardless, with these latest updates, ClamAV declared 960 out of 1172 (81%) binaries to be malicious, and Norton's AV was more successful with 1122 (96%) of the binaries classified as malware. As Norton's AV uses proprietary (and therefore tightly guarded) techniques to perform their classifications, we do not conjecture as to the causes of their higher success rates. Regardless, we display results for ClamAV, as its summaries were much more descriptive than Norton. (In general, Norton tended to classify most binaries simply as a generic form of "spybot".) In addition, we remind the reader that our results use a cryptographic hash sum (MD5) for uniqueness, and we believe anti-virus products use somewhat different metrics for determining uniqueness. Still we believe our results provide a general idea of the effectiveness of two popular anti-virus products against botnet malware.

Table 5.4 provides a listing of ClamAV results over the different malware types, as classified by our graybox analysis. The "Hit" column refers to the number of binaries ClamAV declared malicious, while the "Total" column refers to the total number of distinct (by cryptographic hash) binaries scanned. Finally, the "n-variants" column refers to the number of different malware families (as defined by the ClamAV output) for that test group. The reader should quickly notice that ClamAV achieves a near-perfect record for any malware that automatically scans: Type I botnet executables and worms. However, ClamAV does not perform nearly as well against binaries which do not automatically scan.

Figure 5.4 provides the actual ClamAV classification results against all the confirmed botnet binaries. Unfortunately, even these more descriptive results from ClamAV can be

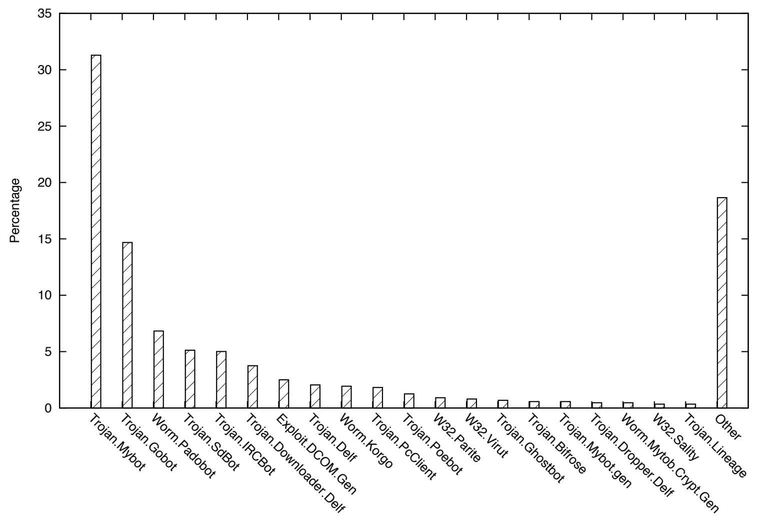


Figure 5.4: Histogram of ClamAV classifications of IRC botnet binaries.

maddeningly vague. For example, we are unclear as to the substantive difference between such labels as `Trojan.Mybot` and `Trojan.IRCBot`. We know from our own testing that many more executables utilize IRC for communication than those classified as `Trojan.IRCBot`. We are unsure why the contributors to ClamAV chose to refer to the communication mechanism in the naming scheme for one malware family, but not others. Also, we quickly note that no ClamAV-generated labels contained the phrase HTTP, despite the fact that many malware instance demonstrated the exclusive use of HTTP as their `C&C` protocol.

	Hit	**Total**	**Hit / Total**	**n-variants**
All binaries	960	1172	.81	41
• IRC Botnet	733	879	.83	33
IRC Botnet Type I	274	275	1.00	12
IRC Botnet Type II	459	604	.76	27
• Not-IRC Botnet	160	174	.92	10
Not-IRC Botnet Type I	153	148	.97	7
Not-IRC Botnet Type II	21	12	.57	5
• Worm	9	9	1.00	4
• Unknown	58	110	.53	15

Table 5.4: Breakdown of ClamAV success rates over all collected binary types.

Non-IRC Botnets Some speculation exists as to the future of botnet `C&C` mechanisms. Many authors believe that botnets are migrating to a pattern of peer to peer (P2P) communication, rather than the centralized approach of an IRC server [8, 23]. While we agree that this seems to be a logical step in botnet evolution, we have not observed any direct evidence of peer to peer communication in our graybox testing. Furthermore, earlier this year, researchers examining malware variants on public P2P systems discovered no evidence of any bot-related malware [20]. However, we must admit that our own lack of evidence cannot be considered conclusive proof of the nonexistence of private P2P botnets. For example, botnets with their own ad-hoc P2P system may currently be localized to regions of the Internet for which we do not have collection capability (either on our own JHU IP space or on PlanetLab). Another explanation is that P2P botnets await to be contacted from their infected peers so as to know where to direct their communiques. The search for these "passive" binaries remains the subject of future work, as our current graybox testing mechanism does not currently address the peculiarities of malware spreading in such a fashion.

The binaries we collected seem to suggest that HTTP is the choice `C&C` protocol after IRC. Of the 174 Non-IRC botnet binaries, all of them attempt to open an HTTP connection to an outside server using port 80, and we believe most of these HTTP connections are intended for command and control. Using HTTP, or at least using the common HTTP ports of 80 and 8080, would seem to be quite prudent, as traffic on these channels is presumably less likely to be blocked (or noticed) by large ISPs. For the most part, studying the dynamics of these HTTP-based botnets remains the subject of future work. However, we believe the study of HTTP botnets could be fruitful. For example, a cursory examination of the network traffic from HTTP bot executables seems to imply automatic reporting behavior not present in any IRC bot we have encountered. This automatic reporting seems to communicate the victim's network conditions to the botmaster, such as whether or not the victim can successfully send mail via SMTP (port 25) to known mail servers. Figure 5.5 illustrates the varying GET commands coming from HTTP-enabled binaries under varying network conditions.

Testing on Virtualized Windows vs. On-Metal Windows. Up to this point, all of the binary analysis has considered only the results from running the binary on a virtualized platform.

```
// Beginning of HTTP traffic from malware,
// with a SMTP server running on the network sink.
GET /reg?u=37BA2649&...&a=1&...&tv=5.1.2600.0 HTTP/1.1

// Traffic from same binary when no SMTP server is accessible.
// The difference is small, but undeniable.
GET /reg?u=37BA2649&...&a=0&...&tv=5.1.2600.0 HTTP/1.1
```

Figure 5.5: A binary using HTTP sends different information depending on network conditions at the victim. GET requests differ by their "a" parameter. Superfluous arguments are excluded in order to better distinguish this small difference.

However, as discussed in the previous section, the 110 binaries with an "Unknown" classification were later run directly on hardware. The goal of this analysis was to determine if the binaries exhibited network traffic on the hardware-based Windows instance that was not seen on the virtualized instances. The result: over 90% of the 110 binaries exhibited non-trivial network traffic when run on a non-virtualized Windows instance. In other words, some facet of the virtualization causes these binaries to behave differently. We cannot be sure if this difference in behavior is due to the binaries purposefully checking for virtualization, or if instead some subtle facet of VMWare simply causes the binaries to crash. We believe that future work with more traditional binary disassemblers may be necessary to determine the underlying causes for these behaviors.

5.2 Insider Observations

All of the data regarding botnets tracked "in the wild" in this work refers to a window of approximately three months, March to May, of 2006, and represents 93 servers joined. 69 of those 93 servers were accessed via a DNS name, while the other 24 made use of a static IP address. The vast majority of our data is collected from logs of the robot. However, some data comes from the network logs of honeypots. While the robot records any observed IRC-level activity in real-time, the IRC traffic from the honeypots is collected forensically. Post mortem, each network trace is then processed so that any transport-layer stream containing IRC-recognizable traffic can be separated. The resulting honeypot logs are then combined with the IRC robots' logs to create a single record of activity for any observed `C&C` server.

The IRC Welcome Message. Much information can also be gleaned from the welcome messages provided by the IRC server when our insider completes an IRC application layer handshake. In fact, as seen in Figure 5.6, the welcome messages of an IRC server can provide a cornucopia of information about the botnet server. These statistics can include such factoids as: the version of IRC server in use, the date and time the server was started, modes supported by the server, users and operators online, currently active channels, or the status of other linked servers. To avoid any confusion, we mention that the term “invisibles” on the line marked `251` refers to clients who have set modes to prevent their information being revealed on `WHOIS` lookups. (Bots become invisibles when they issue a `MODE +i` command, and this practice is quite common.) Furthermore, at the line labeled `255`, the server claims to have “0” servers, as there are no other IRC servers within the domain of this server (IRC servers are linked in tree structures, as parents or leaves). Finally, lines with labels `265` and `266` seem to refer to the current number of clients for this particular server as well as the collaborative tree of linked IRC servers. The “max” seems to refer to the highest user count ever observed by this server since it was last executed.

Of course, we must point out that a particularly wily botmaster could modify his IRC server to output false values. For the purpose of this analysis, we will assume that botmasters who are using off-the-shelf bot software have not gone this far. Instead, botmasters may instead simply change a few flags in the configuration files to suppress output of welcome message factoids. A less helpful botnet would not send any of the information included in Figure 5.6.

The welcome messages immediately demonstrate an interesting aspect of botnet `C&C` servers: botmasters seem to overwhelmingly use UnrealIRC as their server software of choice: 77% of `C&C` servers declared a version of UnrealIRC in their welcome messages. The actual percentage may be higher, with botmasters simply suppressing this output before it reaches the welcome message. Besides being an incredibly powerful piece of software (it supports most any IRC feature one can think of), UnrealIRC is also licensed under the GNU Public License and therefore free to use.

In addition, the welcome messages seem to imply a high-level of linking between servers, and UnrealIRC seems to offer linking through relatively simple modifications to the con-

```
:irc.---.com.sg 001 jnmhgu :Welcome to the Galaxynet IRC Network
     jnmhgu!~udsmwwdt@---.---.---.224
:irc.---.com.sg 002 jnmhgu :Your host is irc.-.com, running version Unreal3.2.4
:irc.---.com.sg 003 jnmhgu :This server was created Mon Feb 6 2006 at 19:26:30 SGT
:irc.---.com.sg 004 jnmhgu irc.-.com Unreal3.2.4
...
...
...
:irc.---.com.sg 251 jnmhgu :There are 1 users and 2123 invisible on 1 servers
:irc.---.com.sg 252 jnmhgu 4 :operator(s) online
:irc.---.com.sg 253 jnmhgu 23 :unknown connection(s)
:irc.---.com.sg 254 jnmhgu 26 :channels formed
:irc.---.com.sg 255 jnmhgu :I have 2124 clients and 0 servers
:irc.---.com.sg 265 jnmhgu :Current Local Users: 2124  Max: 2894
:irc.---.com.sg 266 jnmhgu :Current Global Users: 2124  Max: 2894
:irc.---.com.sg 422 jnmhgu :MOTD File is missing
```

Figure 5.6: A verbose welcome message sent by an IRC botnet server. "jnmhgu" is the name chosen by the robot as it joined this particular server. Some superfluous lines have been removed to better illustrate more relevant information, and the botnet DNS name and robot IP address have been obfuscated.

figurations file. Linked servers appear logically as one server to a client, and offer the advantage of enhanced load balancing. We were quite impressed at the number of botnet welcome messages that advertised some level of linking among the IRC servers. In fact 26 of the 71 IRC servers with verbose welcome messages advertised at least two functional servers. However, we must point out that this ratio is inflated by those botmasters who use public IRC servers for their C&C infrastructure. (By examining common IRC server listings, we noticed that at least six of the IRC servers were either public or semi-public.) Logically, using multiple servers may offer an added layer of protection for a botmaster. The botmaster himself may be connected to a different physical server than his minions, providing yet another hop to traverse before the bots can be traced back to their master. Similarly, multiple linked servers may aid in masking the true size of a botnet to network administrators or other whitehats. Another possibility implies cooperations between hackers who have allied themselves in underground organizations. And of course, botmasters may link multiple servers simply to control more bots under one logical botnet. All of these aspects of linking certainly warrant a more focused examination in the future.

We also noticed that IRC servers often report a very regular integer as the maximum user count. This count represents the largest number of simultaneous users the server has supported at any one time since being started. The fact that this number is often a very regular one (1000 as opposed to 1347), suggests that botnet servers often achieve their software-configured maximum user limit at some point in their lifetime. Our robot experienced this phenomenon on several occasions, thus temporarily preventing access to the C&C server. The error message observed by the tracker is seen in Figure 5.7, and while it occurred repeatedly on a few botnets, we observed it only on four of the servers. In practice, this disruption did little to hamper data collection, as the robot was always able to rejoin after only a few attempts. We note that we did not observe any botnet of a size larger than 3,000 online bots, although we believe the sizes of infected populations could be up to an order of magnitude greater. The limiting factor in these cases seems to be the server capacity rather than the number of available victims.

```
:irc.-.com NOTICE AUTH :*** Looking up your hostname...
:irc.-.com NOTICE AUTH :*** Couldn't resolve your hostname;
     using IP address instead
ERROR :Closing Link: DEA-25916[---.---.---.123] (This server is full.)
```

Figure 5.7: Capacity message sent from the botnet IRC server to the robot immediately upon completion of the TCP handshake. IP address of the robot and DNS name of the botnet have been obfuscated.

The user-counts listed on welcome messages also seem to confirm a phenomenon first reported by Dagon *et al.* [13], namely a diurnal pattern within the sizes of botnets. The figure in Figure 5.8 is directly reflective of the user-sizes as declared in botnet welcome messages. We point out that while the measurements of Dagon *et al.* [13] are collected with DNS black holes and SYN-packet counting (thus measuring an active infection *footprint*), our metric measures the population actually connected to the C&C server. We were impressed to see that this metric seems to suggest that the larger botnets all exhibit a *synchronized* diurnal pattern. The implication is that the relative time-zone distribution of all of these botnets is relatively similar. Of course, some synchronization is to be expected – botnets have, presumably, few victims geographically near the International Date Line. However, the relatively strong synchronization seems to imply a similarity among geographic diversities of many botnets. Furthermore, this phenomenon can be observed through the Google Maps^{TM} [15] visualizations of the upcoming section. In general, three main pockets of activity seem to be prevalent: North America, Europe, and Eastern Asia.

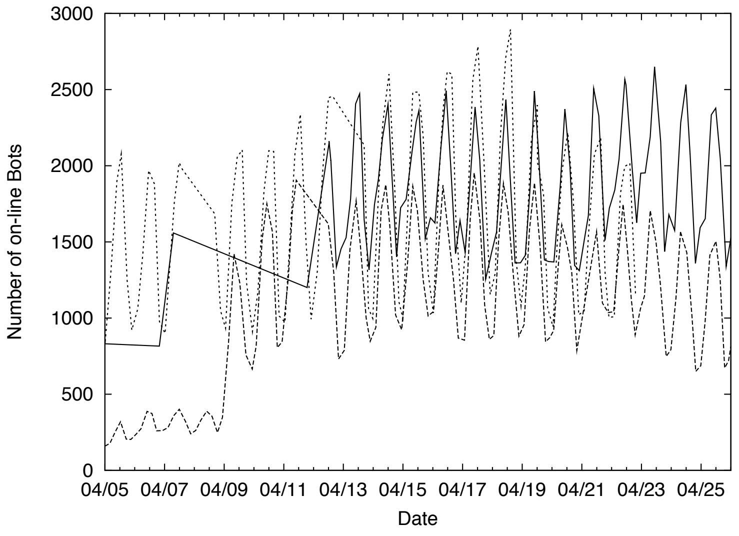


Figure 5.8: Evolution of the effective sizes of active bot populations for some of the larger botnets tracked.

Botnet Migrations. In addition to offering general botnet statistics, botnet logs also demonstrate the frequency of botnet upgrades being performed in the form of the bots downloading new binaries. Many of these upgrades also perform the previously discussed migration to a new `C&C` server; however, the honeypots are prohibited from participating in these relocations. The gateway prevents the honeypots from accessing IP addresses other than those of the original `C&C` server, to ensure that the honeypot in question does not send a malicious payload to an innocent bystander. Instead, the gateway logs can be parsed offline to determine the addresses of botnet upgrade binaries so that these binaries may be downloaded in a safe manner. The robot uses online logic to download binary updates, keeping a timer to ensure that his downloads do not constitute any form of denial-of-service attack. (The robot will not download from the same domain until ten minutes after the completion, or failure, of a download from the domain in question, and we know the robot's GET requests have no embedded exploit traffic.) As we can make no safety guarantees for our infected honeypots, the gateway blocks their immediate requests.

Unfortunately, following a migration of this type in real time is exceedingly difficult, for any botnet-tracking system. Honeypots are far too few, so we cannot devote a new honeypot to track a new server of the next binary. The problem is even greater for our robots. They simply lack the capability to execute newly-downloaded binaries in real-time. As we have discussed in Chapter 4, future infrastructure iterations could use a large pool of idle honeypots to execute binaries immediately after they are downloaded by the robot, only for the purpose of training a new robot instance. We considered this technique, but decided instead to devote our very few honeypots to collection of binaries that spread via exploits or shellcodes not supported by our lightweight responders (in addition to reserving two physical machines for the graybox testing pipeline).

However, even though we have been unable to directly observe an upgrade-prompted migration, our robot has observed instances of temporary moves on multiple botnets. Since temporary moves are ephemeral, they seem to represent more of a load-balancing technique than a pure migration. The first botnet for which we observed these types of moves was suffering from an obvious overcrowding problem. The `C&C` server was apparently programmed with a hard-limit of 1000 clients, and our robot often had to make repeated attempts to connect due to this limit being reached. Seemingly to make room for more minions, one particular botmaster would (almost daily) order the currently online bots to join a different server. Our tracker was lucky enough to observe this migration, and the associated graph of client counts on either server is seen in Figure 5.9. We managed to observe both the original and secondary servers only by programming the robot to remain on the original server in the face of this migration command while spawning an additional robot instance to investigate the secondary server. Furthermore, the capture of this data was possible only because the channel happened to be set without auditorium mode enabled (the channel broadcasted all relevant information to all members the channel). Because of both the channel configuration and the two robot instances, we were able to observe all of four facets of this temporary migration: the exit from the original server, the entrance to the secondary server, the exit from the secondary server, and the entrance back into the original server.

Figure 5.9 clearly shows all four of these phases, and the first two phases occur almost instantly after the command to migrate is issued at time 02:00. However, bots very quickly (on the order of minutes) begin to depart from the secondary channel and reenter the original channel. We believe that bots lost connectivity with the new server due to the fact that the

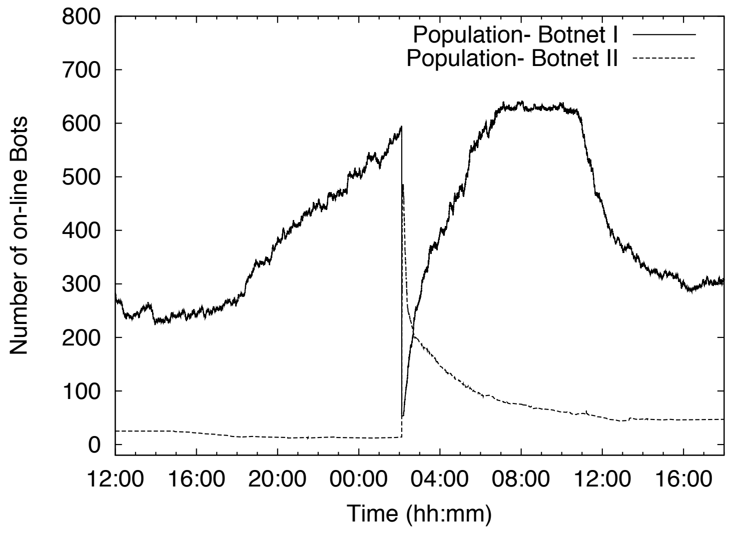


Figure 5.9: Inside view of a botnet migration, as captured by two robots.

topic command from the second server was of the scanning variety, and resulting network congestion caused connection failures with the secondary server. When these temporal scans were completed, bots returned to the original server. We regret that we have no other visualized examples of these migrations. Unfortunately, all other botnets for which we observed temporary moves did not broadcast channel membership information.

Ironically, despite the fact that botnet migrations are disturbingly difficult to track, we have no direct evidence to support the hypothesis that botnet migrations are used as a means to avoid detection or otherwise thwart investigators. While we still believe this hypothesis to be valid, we have only gathered conclusive evidence to support the use of migrations for organizing software upgrades or for botnet load-balancing. Without a doubt, the dynamics of botnet migrations deserve further study.

Bot staying times. In cases where the IRC server broadcasts quit and join notifications to all members of the channel, we can parse the robot's logs to ascertain the normal staying times for victims on the botnet. We refer to this cumulative pattern of bot staying times as the *churn rate* of a botnet. After parsing these logs, we discovered that the staying times are actually much shorter than one might anticipate, and we visualize these staying times for several botnets in Figure 5.10. We note that botmasters themselves are included in this graph, and botmasters tend to stay on the channel much longer than any of their victims.

We believe that the short staying times are best explained by network disruption due to scanning activity being performed by the bots themselves. (Since botmasters themselves do not perform any of the scanning, their connection with the C&C server does not suffer from this problem.) Furthermore, we actually observed one (fledgling) botmaster telling another to avoid setting high scanning rates for the bots, as using high scan rates can result in the bots pinging out, thus disconnecting them from the channel until scanning has ceased.

Of course, the churn rate of a botnet has several implications for its performance. A botmaster's ability to migrate his botnet, or to launch an attack, will be greatly affected by the amount of churn currently being experienced by the botnet. As such, one could conclude that a botmaster should choose his scanning periods wisely by ordering churn-inducing commands only after other activities have ceased. To our surprise, we noticed very few instance of this pattern. In general, churn-inducing scanning commands were intermixed with other churn-sensitive commands (such as a command to move to a different channel). Perhaps these botmasters believe their scanning rates to be low enough to prevent timeouts, or perhaps botmasters only care to interact with the bots on fast enough connections to handle the extra network traffic. We admit that we cannot completely explain the behavior; however, as we will discuss shortly, we do believe that we have observed botmasters favoring bots with faster network connections.

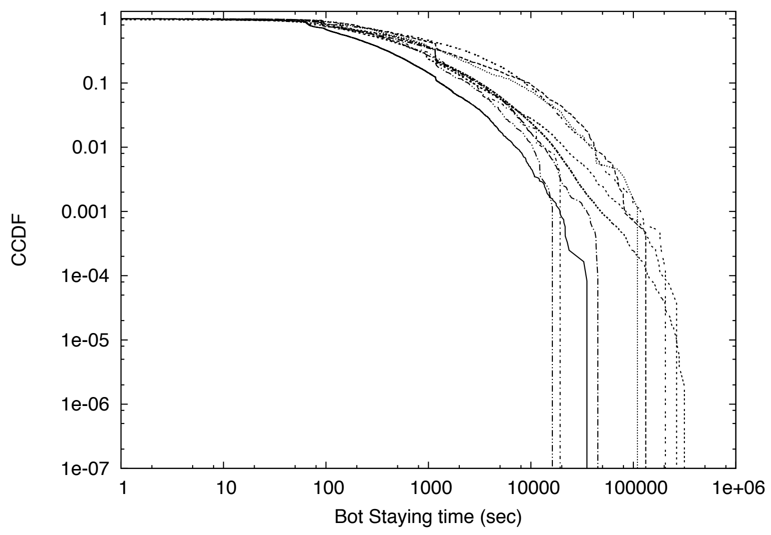


Figure 5.10: Staying times of channel members for non-auditorum IRC botnets as observed by the robot

The Practice of Cherrypicking. While on various botnet IRC channels, our robot observed many different botnet behaviors, many of which we had not been anticipating. The first such behavior that immediately presented itself was a phenomenon we began referring to as *cherrypicking*. Normally, one uses the term cherrypicking to describe the practice of call center employees choosing their next call merely by selecting a flashing red button on their phone. In the case of a botnet, we use the term cherrypicking to mean the practice of hand-selecting particular victims for more specialized tasks. We often observed botmasters probing the different capabilities of their victims through information-gathering commands such as `.netinfo` or `.sysinfo`. In other instances, we believe bots were ordered to download large (but completely benign) data files merely to determine their specific downloading speeds. Once the pick of the litter had been determined, the botmaster would either move these victims to different channels, or order them to download a different binary. Alternatively, the botmaster may simple kick an unproductive bot off of his network.

Much to our delight, cherrypicking often allowed the robot to collect additional information about the botnet's members. After a botmaster issued `.sysinfo` command variants to examine his flock, our robot was able to observe information about the active victims of those channels from the resulting acknowledgments. Table 5.5 illustrates some of the findings and indicates that botnets are comprised of a wide diversity of operating systems. We also point out that even the most recent version of Windows XP (Service Pack 2) was not immune to attack. The participation of a presumably patched Windows system may indicate infection via a non-scanning mechanism, such as by opening an email attachment.

		Service Pack			
OS version	% inf.	None	SP1	SP2	SP3+
Win XP	82.7	.49	.50	.01	n/a
Win 2000	16.1	.12	.04	.06	.78
Win Server	1.2	.57	.43	n/a	n/a

Table 5.5: Distribution of exploited hosts, as extracted from robot logs.

The IRC-savvy reader should be skeptical about how we were able to observe the botmaster communicating to individual victims over his channel. Typically, when chatting in an IRC channel, sending a private message results in the server sending the message only to the intended recipient. However, a botmaster may also opt for a targeted, but public technique. These different techniques were first mentioned in Chapter 4, and we refer the reader back to Figure 4.1 for the discussion regarding these different command-issuing techniques. Figure 5.11 shows an example of cherrypicking as observed by the robot. Note that this example observation is made possible only by the fact that the botmaster chose to use the more visible communication mechanism when singling out a particular bot.

```
// Botmaster authenticates himself and commands
// all bots to report how long they've been connected to the server
:]DjPoia[!DjPoia@Staff.---.Org PRIVMSG ##DoS## :.l crir0x
:]DjPoia[!DjPoia@Staff.---.Org PRIVMSG ##DoS## :.up

// Now knowing how long all bot have been active,
// botmaster commands three bots
// to download a different binary
:]DjPoia[!DjPoia@Staff.---.Org PRIVMSG ##DoS##
      :USA|936461 .scarica http://f4b10.---.org/Dos.exe dos.exe 1
:]DjPoia[!DjPoia@Staff.---.Org PRIVMSG ##DoS##
       :USA|844592 .scarica http://f4b10.---.org/Dos.exe dos.exe 1
:]DjPoia[!DjPoia@Staff.---.Org PRIVMSG ##DoS##
      :USA|820974 .scarica http://f4b10.---.org/Dos.exe dos.exe 1
```

Figure 5.11: Typical botmaster cherrypicking. Only selected bots download the special binary. Botnet DNS addresses are obfuscated in the figure.

Using IRC to Support Multiline Parsing. The tracking robot also observed some rather novel uses of the IRC protocol, such as instructing the victims to perform multiple actions at once. For example, on several occasions we observed a topic command that instructed the victim to join several other channels simultaneously. Each of these new channels then specified a different action (in its respective topic message). This practice can be observed in Figure 5.12, and seems to be a workaround for the failure of many botnet software packages to support multiline parsing. By using this technique, botmasters can set arbitrarily many simultaneous default commands through an equal number of topic-ready channels. However, this multiple-join approach seems rather wasteful when a botmaster could presumably modify the botnet source code to support multiple commands per line. Thus, we can assume that either 1) botmasters don't always have access to the source code for the binaries they are spreading or 2) botmasters don't have the technical know-how or motivation to change the source. Given how easily we were able to download botnet source codes off of the Internet in the Spring of 2006, we suspect the latter possibility to be more likely.

```
 // Tracker first joins the initial channel of the botnet
JOIN #S7 99771122

// Botnet server responds with the topic of the #S7 channel
:h00k3r.---.org.it 332 S7‘6570514928 #S7 :.j #1,#2,#3
....
// Tracker parses the topic and actively responds by sending a join command
// with exactly the argument provided by the botmaster
JOIN #1,#2,#3
...
// The IRC server itself parses the ‘,’ delimiter, moving the tracker
// into the three different channels and providing the appropriate channel topics
:h00k3r.---.org.it 332 S7‘65705 #1 :.acv workssseng 154 3 0 -r -s
:h00k3r.---.org.it 332 S7‘65705 #2 :.sa -s
:h00k3r.---.org.it 332 S7‘65705 #3 :.dl http://www.-.net/h2.exe c:\\h2.exe 1 -s
```

Figure 5.12: Interaction between a robot and a botnet server using a multiple-join technique for bot multitasking. `.acv` and `.sa` are different scanning commands, while `.dl` is a command to download and run a new binary.

General Bot Management via IRC Constructs. In some instances, we observed botmasters using particular IRC channels to manage their botnets. For example, botmasters may reserve one channel for scanning activity and one channel for attack (denial of service activity). In some cases, these channels would partition the connected bot population, while in other cases bots may occupy multiple channels concurrently. Furthermore, different channels on the same server often represented bot populations using different bot software. In the common case, a botmaster issues a command to download a new binary, with this new binary pointing to a different channel. In these cases, a botmaster can observe the progress of a forced upgrade merely by checking the number of bots in a given channel. Keep in mind, the entire infected population cannot be updated with a single command, since some bots may not be directly connected to the `C&C` server (recall the phenomenon of high bot churn rates). As a result, we often see botmasters repeatedly issuing update commands to ensure the entire infected population is re-indoctrinated.

Heterogeneous channels (that is to say, channels with bots using different software packages) did not seem to be especially common. In only one instance did we observed bots running distinctly different software occupying the same IRC channel, observable by differently formatted acknowledgments and the response to different commands. However, on several occasions we observed channels where different iterations of the same original software seemed to be in use in the same channel. By examining the robot's logs, we were able to observe these subtle version differences by noticing small inconsistencies in responses among channel members. For example, a botmaster command to collect scanning statistics may reveal identically formatted responses, but some bots may display more exploit packages than others.

5.3 Botnet Growth

Scanning Commands. Most of the botnets we tracked seemed to expand their footprint by scanning for vulnerable victims, and the robot's observations of scanning commands tell us a story about botmaster scanning practices. In general, scanning commands were very common, with 14% of all direct botmaster commands observed by the tracker seeming to be scanning-related. Scanning commands were even more common as topic commands, with over 40% of topic commands ordering the victims to scan. Botmasters may be wise to use scanning commands as the channel topic, because this practice ensures that a new victim will become productive immediately after completing their connection to the `C&C` server. However, live scanning commands tend to be much more targeted than their topic counterparts; scanning commands set as topics seemed to focus more on localized scanning of the victim's current class A or class B subnet. Table 5.6 demonstrates these observations more precisely. We believe these results are directly indicative of a botmaster using a hands-on approach when attempting to break new ground into an unknown subnet. Also, the botmaster may be searching for a particularly vulnerable population before devoting much scanning time to a particular region.

Patterns of Botnet Growth. Unfortunately, modeling growth poses many challenges previously not addressed by the literature, thanks to the wide variety of exploit scanning patterns illustrated by Table 5.6. And of course, scanning may be only one spreading medium for

	Default Topic	Botmaster Command
Localized scanning	66%	15%
- `Class A`	11%	18%
- `Class B`	89%	82%
Targeted scanning	32%	87.4%
- `Class A`	80%	88%
- `Class B`	20%	12%
Uniform scanning	2%	0.3%

Table 5.6: Relative frequencies of scan-related commands on tracked botnets.

the botnet. For example, a botnet binary could spread by sending links through victims' instant messaging clients. Finally, all of these variables are exponentiated in complexity thanks to their dependence on the most complex variable of all: the human botmaster. At a whim, the botmaster can change when the bots scan, how fast they scan, what exploit they use, or if the bots use any other spreading medium.

Because of the wide variety of variables contributing to a botnet's growth, we see a variety of growth patterns. These growth trends can be observed by simply parsing the logs of our IRC robot. For the 52% of botnets that broadcast joins and leaves to the channel, we can monitor the ever-changing size of the botnet channel (an initial baseline size can be obtained by parsing the welcome statements, as discussed previously). We assume that cloaked IP addresses maintain a bijective mapping with IP addresses, and that each IP address represents a unique victim. As a whole, we believe the resulting numbers are a direct reflection of the number of active participants on the channel. Three of the more prevalent trends pulled from data collected in this manner can be seen in Figure 5.13, and the exact growth pattern observed seems to directly reflect the commands issued by the botmaster to his minions. For example, botnets with a semi-exponential growth pattern as in Figure 5.13.a show very little variation in their commands, generally leaving a standing order as the topic of the channel. The first botnet in this figure had an unwavering and standing order for botnets to scan their local subnets for over a one-month period. The resulting growth appears very similar to that of non-uniform scanning worms.

Of course, botnets may also exhibit much more intermittent behavior. In many cases, a

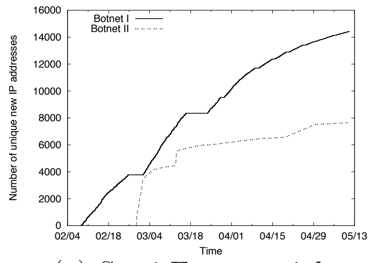
C&C server may fall out of service briefly, only to reactivate a day later. In other instances, a botmaster may turn his attention away from scanning activity in order to use his victims to aid in denial of service attacks or other nefarious activities. Regardless of the reason, we have observed downtimes of varying lengths. In one of the cases represented in Figure 5.13.b, the botnet C&C server fell out of service for a period of 18 days before coming back online (only to be re-discovered by our honeypots). This inconsistent behavior appears as a staircase pattern on our growth charts, and server downtimes are reflected by periods of little no growth on the charts. We point out that, quite conversely, a Type I botnet would probably experience (footprint) growth even during server downtime; however, we were not able to track any Type I botnets[2].

Figure 5.13.c illustrates a more linear growth pattern. This growth pattern was prevalent for many botnets, and seems to be linked to the behavior of time-scoped scanning. For example, often a botmaster will order his victims to scan for a fixed amount of time before ceasing any spreading activity. Scanning only continues due to an active botmaster reissuing scanning orders on a regular (often several times daily) basis. Alternatively, a botmaster may simply leave a standing (but still time-scoped) scanning command as the channel topic. Thus, scanning occurs only when victims first become infected, or otherwise leave (and then rejoin) the channel. Victims could rejoin the channel for a plethora of reasons. Perhaps the network connection is simply overloaded, or perhaps the human operator of the victim machine performs a restart in an attempt to restore some level of system stability.

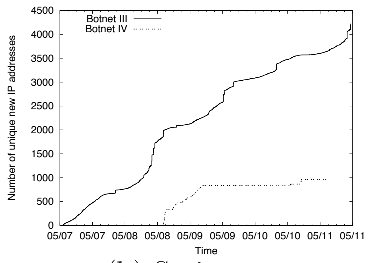
We verified the growth numbers observed by our robot by comparing its logs against hit counts from our DNS probes. Thus, we gain a second perspective of botnet growth, considering the infection footprint rather than the actively online victim population. The resulting data compliments the robot's observations very nicely. Figure 5.14 shows growing sizes of unique DNS cache hits for the focused botnets. The same semi-exponential, staircase, and linear patterns are visible in the charts. The charts seem to further confirm the heterogeneity of botnets, with many different victims residing in different DNS domains.

As a whole, the scanning strategies of botnets do not seem to be conducive to the flash-

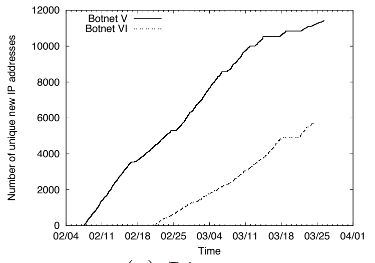
[2]The robot did join one Type I botnet being hosted on a public server, but the server administrators were aware of the problem and had already begun banning victims.



(a) Semi-Exponential

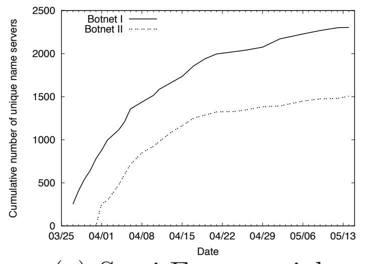


(b) Staircase

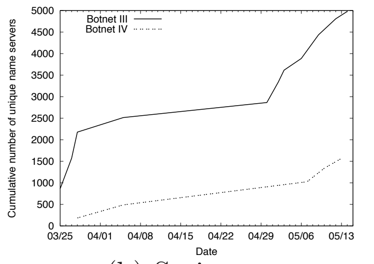


(c) Linear

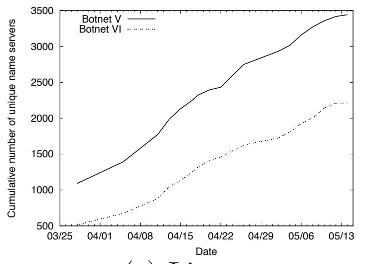
Figure 5.13: Views from the robot showing multiple botnets with the three most predominant growth patterns.



(a) Semi-Exponential



(b) Staircase



(c) Linear

Figure 5.14: Views from the DNS cache probe showing footprints for multiple botnets with the three most predominant growth patterns.

epidemic effects observed with such phenomenon as the Witty Worm [34]. First of all, botmasters generally seem to use older and well known exploits that only have an effect on unpatched user-end systems. Furthermore, the scanning methodologies used (extremely targeted scanning, localized scanning, or time-scoped scanning) seem to sacrifice spreading efficiency for a level of stealth. As an example. consider the fact that every one of the Type I botnets we discovered no longer had an effective C&C server. Presumably, the automatic scanning activity had gained the attention of the right (or wrong, depending on your perspective) people, thus resulting in the eventual shutdown of the IRC server. On the other hand, many of the Type II binaries we discovered still had active servers (and botmasters). These botmasters were probably wise to be choosy about what ranges they scanned, even if their victim count suffered as a result.

5.4 Botmaster Commands

Of course, botnets support all kinds of behavior other than simply scanning for new victims. In order to better understand overall botnet behavior, we must examine the non-scanning commands doled out by the average botmaster. To this end, we loosely classify all botmaster commands as seen by the robot (and honeypots) into one of eleven categories, described as follows:

- Attack - Bots launch a distributed denial of service attack.
- Click Fraud - Bots visit a web-based ad.
- Cloning - Bots create a copy of themselves on another IRC server or channel.
- Control - Bots perform a specific IRC-level action, like changing channels.
- Download - Bots download a new file from a provided URL address.
- Hosting - Bots start their own application server such as a FTP or HTTP server.
- Identity Theft - Bots provide potentially personal information, such as clipboard contents or specific file contents.
- Mining - Bots provide system information (such as network statistics).
- Piracy - Bots provide CD keys or activation codes that reside on the host system.
- Scanning - Bots scan for new victims.
- Unknown - Resulting bot behavior was not determined.

The categorization of any given command (in other words, determining that the `.scan` command fits under the "Scanning" category) involves a two step process. First, we must determine the intended action of the command. This can generally be accomplished by observing the behavior of the malware when sent the command during graybox testing[3]. Second, we must assign this logical action into one of the aforementioned eleven categories.

[3] In a few cases, determining the actual action resulting from a command was not possible, simply because the binary did not respond when placed under scrutiny.

Admittedly, one could argue that this assignment is somewhat subjective, but we doubt this subjectivity creates any conflict, since the categories were created to fit the data, not visa versa. The "Unknown" category consists of any command for which we were unable to determine the action it was intended to trigger. We also classified many commands that appear to be botmaster typographical errors (`.scantop` in lieu of `.scanstop`), in this category rather than try to guess the intended function.

Furthermore, if we examine botnets by their overall size (as determined via the robot logs or via DNS probing), we notice that botmasters over smaller domains tend to practice many more "hands-on" techniques. These botmasters are more likely to manage their bots with manual commands, moving them from place to place. In addition, these smaller botnets also tended to favor more data-mining activity. With fewer bots, the botmaster has time to examine each of his victims in detail. (As mentioned before, with these small botnets, the active responses of the robot allowed our agent to stay online even during these periods of hand-inspection.) Botmasters of larger fiefdoms seemed much less focused on extracting the statistics of individual bots, and tended to favor the downloading of new binaries for their management. On the other hand, cloning activity seemed to be more prevalent in larger botnets. This behavior presumably comes about because of the use of cloning as a denial of service activity, and cloning will only be an effective attack with larger botnets. A full breakdown of command distribution among small, medium, and large botnets is shown in Figure 5.15.

The Behavior Behind the Commands. Unfortunately, a simple distribution does not tell the whole story. For example, cloning activity comprised of only 0.2% of all activity for the botnets tracked, but accounted for 41% of the activity on one botnet in particular (23 cloning commands). We believe cloning used in this abundance is likely to represent a denial of service attack against another IRC server. In general, many botnets seem to be specialized, focusing on one particular activity. In many cases, this activity is obvious from the DNS name: `xxx.fl00d.com` may refer to a botnet performing an abundance of denial of service activity (flooding). In other cases, the IRC channel name may provide a clue to the botmaster's intended use for his minions.

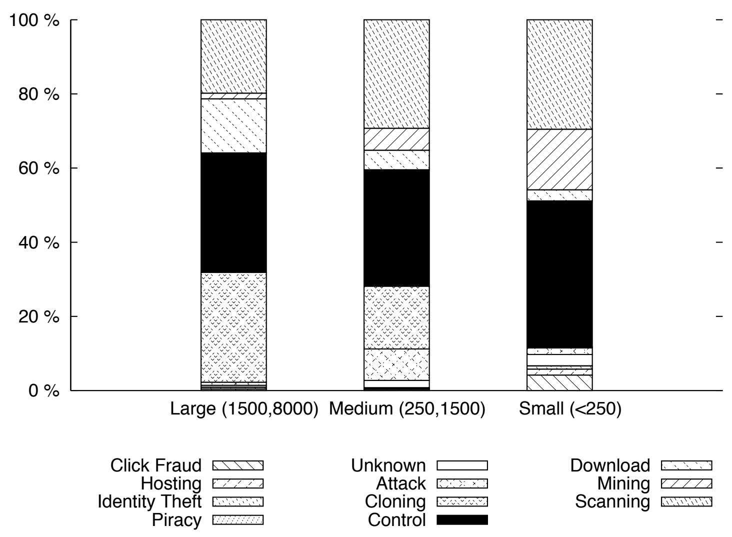


Figure 5.15: Percentage of command types as a function of observed botnet size.

Indeed, inferring the significance behind the command statistics is wrought with difficulty. Perhaps the first deficiency of our system lies with the inability to appropriately classify the class of “Download” commands. A command to download a binary may do any number of things, for the simple reason that the transferred file in question could perform any task, not merely those tasks directly involved with communicating with the C&C server. For example, a downloaded file may open a pre-packaged web-server, complete with “phishing” website. Unfortunately, this type of (passive) behavior is not directly observable in with our infrastructure.

Furthermore, many other challenges hinder an insightful analysis of a particular botmaster’s behavior. He may actually be several individuals sharing single nickname, he may be controlling several botnets simultaneously with different nicknames, he may be supplementing his management with the use of his own management scripts, or he may simply be testing some code he wrote last weekend for a homework assignment. We are still missing the answers. Plus, the human dynamism of the botmaster essentially squashes any dreams we may have of generating a simple model of botnet behavior like those models previously created for scanning worms. To our knowledge, no one has been able to fully understand the Wizard of Oz without peeking behind the curtain, and in many ways, we have yet to draw the botmaster curtain.

Chapter 6

Botnet Visualization

“My computer’s sick. I think my modem is a carrier.”

- *Unknown*

In order to visualize the global spread of a given botnet, we harnessed the power of the Google Maps^{TM} [15] JavaScript API. Using Google Maps^{TM} means that our visualization has an interactive interface as well as being cartographically accurate. Of course, regardless of the APIs used, constructing a relevant botnet visualization first requires knowledge of the Internet address of both botnet server and botnet victim(s). Discovering the address of the `C&C` server is generally trivial, since discovery of a botnet usually inherently provides this address. Determining the addresses of victims is a different story altogether. In many cases, our insider may not have any view of other victims on the channel. Furthermore, even if the tracker does see other channel members, the Internet addresses of these members is usually obfuscated by the previously discussed practice of cloaking. These problems greatly inhibit the robot’s ability to paint a clear picture as to a particular botnet’s current membership. As a result, we have preferred to rely on DNS cache probing to provide the Internet addresses of victims. True, the DNS cache probes do not determine the exact addresses of victims, but they do discover the addresses of the DNS servers who support infected populations. We assume that these DNS servers are geographically close to their clients. The reader should keep in mind a simple result of basing our visualization on DNS cache probes: the resulting maps represent a botnet’s footprint, or its overall infection count, rather than a currently connected army of Windows foot-soldiers.

Of course, in order to effectively visualize IP addresses on a map, these addresses must be mapped to geographical coordinates. To this end, we utilized the IP2Location [17] dataset. The producers of this dataset release an update database every month, and we made sure to use the database released during the month the DNS probes were made. While we cannot comment on the precise accuracy of the IP2Location datasets, we have no reason to doubt their reliability.

The creation of an eye-pleasing botnet map first begins with output from our DNS cache probes. This output consists of a simple list of all IP addresses this botnet server used over the sample period, as well as a list of the IP addresses of any DNS servers with infected clients. Many botnets were found to have servers with multiple IP addresses. In some cases, multiple server addresses were the result of load balancing via DNS. In other cases, the IP address of the server changed, while the DNS name did not.

This output is quickly processed with the IP2Location data by a simple script to produce an XML output file. Figure 6.1 illustrates the format of the file. The resulting latitude and longitude points fit easily into Google's API. The XML file is then easily parsed by a JavaScript website, and the associated points are drawn. Figure 6.4 illustrates the final output. Server addresses are represented by stars, while transparent (orange) circles are used to represent the geographic locations of cache-positive DNS servers. We refer to these images as server graphics and victim graphics, respectively. Because the victim graphics are transparent, clusters of geographically close hits appear brighter on the map, due to the graphics overlapping. More precisely, each victim graphic has an alpha value (opacity) of 0.10. Thus, the geography underlying the intersection of ten or more victim graphics is completely masked.

Additionally, we quickly discovered that the IP2Location dataset seems to map many IP addresses (or ranges of IP addresses) to a single intersection of latitude and longitude. To better illustrate "heavy hitter" coordinates, the diameters of the transparent orange circles are multiplied by a factor, $f_{dia} = \min(hits, constant)$, before being displayed. *hits* refers to the number of DNS servers mapping to this geographic location, while *constant* ensures that the orange graphics do not occupy an obtrusive amount of the screen. In our examples

```
<bots>
     <master dns="irc.botnet.net" ip="XXX.XXX.240.180" lat="52.517"
          long="13.4" date="4/1/2006" index="0"/>
     <victim ip="XXX.XXX.154.74" lat="42.4402"
          long="-76.4922" index="0" count="1"/>
     <victim ip="XXX.XXX.128.188" lat="53.133"
          long="23.15" index="0" count="1"/>
     <victim ip="XXX.XXX.32.2-XXX.XXX.4.33-XXX.XXX.37.196-XXX.XXX.37.211"
          lat="35.859" long="14.489" index="0" count="4"/>
     <victim ip="XXX.XXX.1.9" lat="-31.933"
          long="115.833" index="0" count="1"/>
</bots>
```

Figure 6.1: XML format used as input to the Google Maps^{TM} visualization. IP prefixes are obfuscated, and `index` values are not necessary for visualization of a single botnet.

wet set $constant = 100$.

Also, in order to keep a victim graphic's size consistent with the underlying geography, the diameter of victim graphics are resized whenever an end-user changes the zoom level of the map. In order to handle a zoom level change, victim graphics are modified by a factor of $f_{zoom} = \sqrt{|(zoom_{old} - zoom_{new})|}$. Google Maps^{TM} currently supports a set of zoom levels $Z = \{0, 1, \cdots, 15\}$. The dimensions of server graphics remain static regardless of zoom level. The relative resizing can be viewed in Figure 6.2 overlooking a botnet based in South Korea.

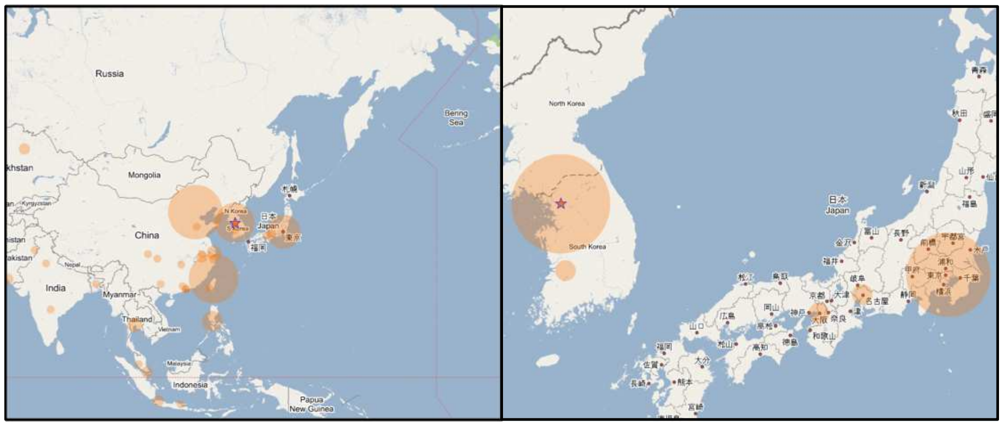


Figure 6.2: Two different views of a South Korean botnet. Zooming prompts automatic resizing of influence circles. Server icons remain unchanged.

The graphics of botnet servers were further emphasized in the cases where botnet servers were linked together for load-balacing purposes. In these cases, we draw a blue line between servers to denote the connection. To increase visual appeal, the line is drawn elliptically. (Alas, we cannot claim credit for this idea, as elliptical lines are often used by airline companies in maps of flight routes.) Unfortunately, the underling API does not provide direct support for ellipses, so the curves are created by combining many smaller straight lines. The JavaScript source code for determining the points of these lines is provided in Figure 6.3, with an example of this output displayed in the right side of Figure 6.4. Lines are displayed with similar thickness and precision at every zoom level. Also, the graph of connected servers is currently generated as a star graph, with the most geographically central server chosen as the vertex of highest degree.

```
var height_factor = 7;

for (var m = 0; m <= 100; m++) {
   var e_point = new GPoint(distance / 2 * Math.cos((m / 100 * Math.PI)),
          factor * distance * Math.cos(angle) / height_factor
          * Math.sin((m/100 * Math.PI)));

   //now fix the angle of this ellipse to match the given points...
   e_point.x = o_x * Math.cos(-1*angle) + o_y * Math.sin(-1*angle);
   e_point.y = (-1) * o_x * Math.sin(-1*angle) + o_y * Math.cos(-1*angle);

   e_point.x += line_bounds.minX + line_width / 2;
   e_point.y += line_bounds.minY + line_height/ 2;

   line_ra.push(e_point);
}

var child_line = new GPolyline(line_ra, "#3333FF", 2,.3);
```

Figure 6.3: JavaScript algorithm for point-placement of the elliptical lines. Vertical height of the ellipse is determined by the `height_factor` variable. 100 points are placed for each line.

The visualizations show that botnets are very generally very global. However, some botnets seem to be much more regionally focused. Figure 6.4 compares the visualization of several very distinct botnets. Several interesting features immediately appear from these visualizations. First, while some botnets seem to be very focused in China, other botnets seem to be diverse among the major population centers of North America, Europe, and Asia.

Similarly, India and South America seem to enjoy complete immunity from some botnets. Admittedly, we are not entirely sure why some botnets seem to focus on particular regions while others do not. A likely explanation is simply because a botnet may spend most of its lifetime spreading within a particular /8 space, with all of these addresses being localized to a particular area of the world. Alternatively, a botnet may spread via vulnerabilities found only in certain variants of Windows. For example, the German version of Windows XP may contain a vulnerability not found in the English (US) version.

We believe these visualizations provide an efficient means to quickly ascertain a botnet's demographics. An obvious extension to this work would be to update the botnet statistics with real-time tracking, rather than using pre-generated XML reports. Indeed, these real-time updates could be quite an addition to an otherwise drab network security command center. Still, we believe our visualizations have inherent usefulness even as generated reports.

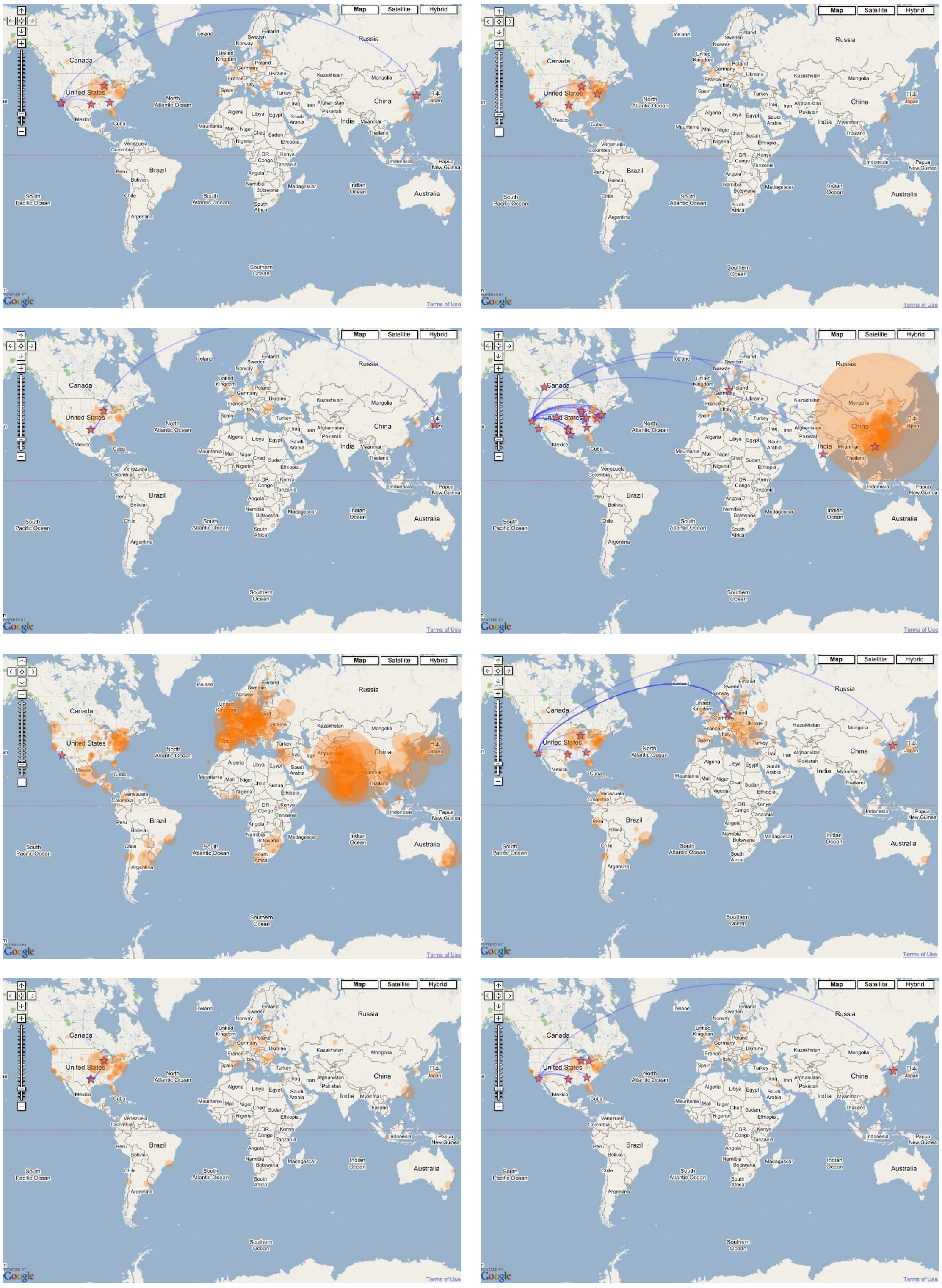


Figure 6.4: Several very geographically distinct botnets viewed at similar zoom levels.

Chapter 7

Related Work

"tell me a good range to scan... i want a fast range to scan"

- *Unknown Botmaster, March, 2006*

Despite botnet prevalence over the Internet, research on the botnet phenomenon has been slow to reach the mainstream academic community. Our tracking infrastructure was highly motivated by the earlier work of the Honeynet group with both their informal study [29] and related academic paper by Freiling *et al.* [14]. In these works, the authors describe their use of lightweight responders to capture malicious binaries, honeypots to determine the location of `C&C` server locations, and silent IRC agents to observe traffic on botnet channels. Our infrastructure improves on their approach in several ways. First, we use a much broader collection network of lightweight responders and honeypots to collection malicious binaries. Next, we employ a fully automated graybox testing mechanism to train our IRC drones. This training enables our drones to appear as active participants on the botnet channel. Furthermore, unlike any system of the past, our graybox back-end enables our robots to monitor botnets during their most active periods. Finally, we compliment our IRC tracker with DNS cache probing. As we have shown, the external viewpoint provided by cache probing is undeniably crucial to the study of botnet demographics.

Others have attempted to measure botnet prevalence by instead measuring the amount of time before an un-patched Windows instance unwittingly joins a botnet. Cooke *et al.* explored botnets in this fashion as well as discussed the possibilities for alternate `C&C` channels such as peer-to-peer (P2P) protocols [8]. While we agree that P2P protocols are viable mechanisms by which botnets may communicate, we have not found any evidence of P2P communication mechanisms either through our honeypots or in any of the network fingers prints, f_{net}, generated by our graybox analysis. Instead, we believe that protocols such as HTTP will provide the next communication mechanism for botnets, as HTTP traffic may more easily slip through hardened firewalls.

Microsoft has taken their own approach to measuring the prevalence of malware and other botnet phenomenon. In their most recent Security Intelligence Report [4], Microsoft leverages data collected by the usage of the Windows Malicious Software Removal Tool (MSRT) [26] and the Windows Defender [27] products. Without any observation of live botnets, statistics collected by these tools have provided an incredible amount of raw data regarding the number and frequency of present-day malware variations. Microsoft researchers also provide malware prevalence information by region, as well as by different operating system. We point out that our own data regarding operating systems of infected hosts on botnets (see Table 5.5) closely matches Microsoft's own observations of infection.

While Microsoft has collected their statistics from victim computers, other researches, namely Barford and Yagneswaran, have examined botnet source code in an attempt to understand the problem [3]. We believe we have seen several live botnet variants using variations of the sources studied in this work. In fact, the recurring observation of these few botnet architectures is quite striking. This relative homogeneity of botnet software highlights the importance of understanding the capabilities being developed in current botnet source code.

In the realm of malware collection, Vrable *et al.* have produced an impressive system with their creation of Potemkin [40], a scalable virtual honeynet system. Potemkin attempts to overcome the scalability issues of high-interaction honeypots by deploying paravirtualized Windows instances directly in response to incoming traffic. While this effort may prove invaluable for malware collection, it does not overcome the problem of long-term botnet tracking, as each paravirtualized instance must be reclaimed shortly after a malicious binary is collected.

However, Potemkin may soon realize a vast increase in capability, thanks to an incorporation with the lightweight mechanism known as RolePlayer [41]. RolePlayer can mimic arbitrary protocols in an extremely lightweight fashion, only defaulting to the resources of a high-interaction honeypot when encountering never-before-seen traffic. This idea of a RolePlayer proxy has been partially implemented with a system known as GQ [12]; however, the final implementation is currently incomplete. Still, a completed GQ system stands to enable orders of magnitude improvements in the ability to capture the payloads of scanning worms and botnets. With this system, the vast quantity of (unremarkable) scans could easily be filtered by RolePlayer, and fully-functional high-interaction honeypots will be utilized for never-before-seen traffic. Indeed, RolePlayer and Potemkin should prove to be excellent companions. Furthermore, we believe the same hybrid approach could be applied to botnet tracking; RolePlayer's techniques could be applied to the IRC tracking robot, with Potemkin-like virtual machines ready to handle never-before seen botmaster commands.

With regard to automatic malware analysis, researchers at the University of Mannheim (associated with Honeynet group) have developed a technology with similar goals to our graybox testing. Their platform, known as CWSandbox [42], uses API hooking to observe Windows system calls made by the malicious binary, and is similar in technique to the earlier Norman Sandbox [5]. In some ways, CWSandbox is more powerful than our graybox testing technique: it can observe all API calls issued by malware, whereas our system only records the network activity resulting from those API calls. However, our techniques of graybox testing allow for much of the same information to be inferred without delving into the complexities of emulating the Windows API.

Finally, other researchers have attempted to model the spreading behavior of botnets. Dagon *et al.* recently attempted to model botnet propagation under the assumptions that botnets scan without contact from their botmaster (Type I botnets only), and do so uniformly [13]. In fact, our results have shown, overwhelmingly, that botnets usually scan in a non-uniform or targeted fashion, and we are uncertain how these changes in assumptions would affect the propagation models of Dagon *et al.* Furthermore, issues such as footprint size, online member size, churning rate, propagation mechanism, and human behavior factors of the botmaster, will also have to be considered before an exhaustive botnet propagation model could be developed.

Chapter 8

Conclusion

"Imagine a school with children that can read or write, but with teachers who cannot, and you have a metaphor of the Information Age in which we live."

- *Peter Cochrane*

Left unchecked, the botnet problem poses a great risk to Internet health. Unfortunately, learning about botnet behavior has proven to be extremely difficult. Using previous techniques, tracking more than a trivial number of botnets has been inefficient both in terms of physical resources and manpower. It is our hope that a combination of both high-interaction and low-interaction tools can mitigate problems of scalability while enabling long-term and accurate observation of botnets.

Our DNS prober and IRC robot have taught us much about botnet behavior in a relatively short period of time. We are now better able to answer questions about botnet growth, membership, migration, and general behavior. Furthermore, our discoveries prompted many more botnet questions. For instance, do reports of a botnet's size refer to a footprint, or their current membership? Is a botnet with an offline `C&C` server still a botnet? How does the churn rate of a botnet's online population influence its effectiveness for malfeasance? Prior to the construction our infrastructure, we knew not even to ask these questions. Taken as whole, our tools and techniques have taught us many important facts about botnets, and only by understanding this complex phenomenon can we hope to develop effective botnet defenses.

Bibliography

[1] Paul Baecher, Thorsten Holz, Markus Kötter, and Georg Wicherski. The Malware Collection Tool (mwcollect). Available at `http://www.mwcollect.org/`.

[2] Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix Freiling. The Nepenthes Platform: An Efficient Approach to Collect Malware. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2006.

[3] Paul Barford and Vinod Yagneswaran. An Inside Look at Botnets. In *Series: Advances in Information Security*. Springer, 2006.

[4] Matthew Braverman, Jeff Williams, and Ziv Mador. Microsoft Security Intelligence Report, January – June 2006: A in-depth perspective of trends in the malicious and potentially unwanted software landscape in the first half of 2006. Available at: `http://www.microsoft.com/technet/security/default.mspx`, 2006.

[5] Norman Sandbox Information Center. Sandbox Whitepaper. Available at: `http://sandbox.norman.no/pdf/03_sandbox\%20whitepaper.pdf`, 2003.

[6] Z. Chen, L. Gao, and K. Kwiat. Modeling the spread of active worms. 2003.

[7] Clam AntiVirus. Available at `http://www.clamav.net/`.

[8] Evan Cooke, Farnam Jahanian, and Danny McPherson. The Zombie Roundup: Understanding, Detecting, and Disturbing Botnets. In *Proceedings of the first Workshop on Steps to Reducing Unwanted Traffic on the Internet (STRUTI)* , pages 39–44, July 2005.

[9] Microsoft Corporation. MS04-007: An ASN.1 vulnerability could allow code execution. Available at: `http://support.microsoft.com/?kbid=828028/`, February 2004.

[10] Microsoft Corporation. MS04-011: Security Update for Microsoft Windows. Available at: `http://support.microsoft.com/kb/835732/`, April 2004.

[11] Symantec Corporation. Multiple Microsoft RPC DCOM Subsystem Vulnerabilities. Available at: `http://securityresponse.symantec.com/avcenter/security/Content/2003.09.10.html`, September 2003.

[12] Weidong Cui, Vern Paxson, and Nicholas Weaver. GQ: Realizing a System to Catch Worms in a Quarter Million Places. In *ICSI Tech Report TR-06-004*, 2006.

[13] David Dagon, Cliff Zou, and Wenke Lee. Modeling Botnet Propagation Using Time Zones. In *Proceedings of the* 13th *Network and Distributed System Security Symposium NDSS*, February 2006.

[14] Felix Freiling, Thorsten Holz, and Georg Wicherski. Botnet Tracking: Exploring a root-cause methodology to prevent denial-of-service attaks. In *Proceedings of* 10th *European Symposium on Research in Computer Security, ESORICS*, pages 319–335, September 2005.

[15] Google Maps API, `http://www.google.com/apis/maps/`.

[16] Luis Grangeia. DNS Cache Snooping or Snooping the Cache for Fun and Profit, Available at `http://www.sysvalue.com/papers/DNS-Cache-Snooping/files/DNS_Cache_Snooping_1.1.pdf`, 2004.

[17] IP2LOCATION, Bringing Geography to the Internet. Available at `http://www.ip2location.com/`.

[18] V. Jacobson, C. Leres, and S. McCanne. Packet Capture library. See `http://www.tcpdump.org/`.

[19] M. St. Johns. RFC 1413: Identification protocol, January 1993.

[20] Andrew Kalafut, Abhinav Acharya, and Minaxi Gupta. A Study of Malware in Peer-to-Peer Networks. In *Proceedings of the* 6th *ACM Internet Measurements Conference (IMC)*, October 2006.

[21] C. Kalt. Internet Relay Chat: Client Protocol. RFC 2812 (Informational), April 2000.

[22] Dan Kaminsky. Welcome to Planet Sony, `http://www.doxpara.com/`.

[23] Jayanthkumar Kannan and Karthik Lakshminarayanan. Implications of Peer-to-Peer Networks on Worm Attacks and Defenses. In *un-published manuscript*, 2003.

[24] Key clicks betray password, typed text, `http://www.securityfocus.com/news/11318`.

[25] Willaim Metcalf. Snort In-line. Available at `http://snort-inline.sourceforge.net/`.

[26] Microsoft. Malicious Software Removal Tool. Available at: `http://www.microsoft.com/security/malwareremove/default.mspx`, 2006.

[27] Microsoft. Windows Defender. Available at: `http://www.microsoft.com/athome/security/spyware/software/default.mspx`, 2006.

[28] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. *SIGCOMM Computer Communication Reviews*, 33(1):59–64, 2003.

[29] Honeynet Project and Research Alliance. Know your enemy: Tracking Botnets, March 2005. See `http://www.honeynet.org/papers/bots/`.

[30] Niels Provos. A virtual honeypot framework. In *Proceedings of the USENIX Security Symposium*, pages 1–14, August 2004.

[31] Moheeb Abu Rajab, Fabian Monrose, and Andreas Terzis. On the Effectiveness of Distributed Worm Monitoring. In *Proceedings of the* 14th *USENIX Security Symposium*, pages 225–237, August 2005.

[32] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Proceedings of the* 6^{th} *ACM Internet Measurements Conference (IMC)*, October 2006.

[33] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. 2000.

[34] Colleen Shannon and David Moore. The Spread of the Witty Worm. *IEEE Security and Privacy Magazine*, 2(4):46–50, July 2004.

[35] K. Sollins. The TFTP Protocol (Revision 2). *RFC1350*, July 1992.

[36] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *USENIX Annual Technical Conference*, 2001. Available at `http://www.vmware.com/`.

[37] Symantec. Norton anti-virus. See `http://securityresponse.symantec.com/`.

[38] Symantec. Norton ghost. See `http://www.Symantecstore.com/NortonGhost`.

[39] The UnrealIRC Team. Unrealircd. See `http://www.unrealircd.com`.

[40] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfarm. *Proceedings of ACM SIGOPS Operating System Review*, 39(5):148–162, 2005.

[41] Nick Weaver Weidong Cui, Vern Paxson and Randy H. Katz. Protocol-Independent Adaptive Replay of Application Dialog. In *Proceedings of the* 13^{th} *Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, Feb 2006.

[42] Carsten Willems and Thorsten Holz. CWSandbox: Automatic Behaviour Analysis of Malware. Available at: `http://www.cwsandbox.org/`, 2006.

[43] Cliff Zou, Weibo Gong, and Don Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of ACM Conference on Computer and Communication Security (CCS)*, pages 138–147, 2002.

Curriculum Vita

Jay Zarfoss was born on February 8, 1982, in York, Pennsylvania. He received his Bachelor of Science degree in 2004, and he is current pursuing his Master of Science in Engineering – both from the Computer Science Department of the Johns Hopkins University in Baltimore, Maryland. His research interests focus primarily on advancements in network security.