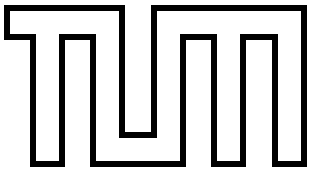


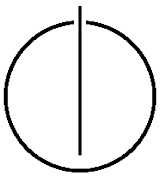
FAKULTÄT FÜR INFORMATIK

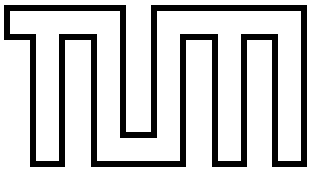
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Adapting Blackhat Approaches to Increase the Resilience of Whitehat Application Scenarios

Bartlomiej Polot





FAKULTÄT FÜR INFORMATIK

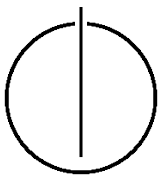
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Adapting Blackhat Approaches to Increase the Resilience of Whitehat Application Scenarios

Adaptieren von Blackhat-Ansätzen um die Ausfallsicherheit in Whitehat-Anwendungsszenarien zu erhöhen

Author:	Bartlomiej Polot
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	Dipl. Inf. Christian Schneider
Advisor:	Martin Brunner, BSc
Date:	June 15, 2010



Ich versichere, dass ich diese Masters Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.Juni 2010 Bartlomiej Polot

Dedication

I dedicate this thesis to my father Bogusław who celebrates his 50th birthday today, June 15th, 2010.

Acknowledgments

I would like to thank my advisor Martin Brunner, from Fraunhofer SIT in Munich, for his guidance in all aspects throughout the creation of this thesis. I would also like to thank Alexander Kiening, also from Fraunhofer SIT in Munich, for his help in the revision of the drafts.

Finally, I want to thank my parents Ewa and Bogusław for their support during my whole academic life.

Abstract

Botnets, large networks of computers controlled by a malicious master, are nowadays big threats to computer security. They have evolved from being simple centralized networks of hundreds of computers to complex multi-function systems with millions of computers. Botnets use different novel techniques to increase their resilience and be harder to analyze, track and control by numerous groups of people trying to counter-measure them.

In this thesis, the techniques botnets use to achieve their high resilience are gathered and analyzed to form a general idea about the state of the art in distributed malware. This knowledge is used to build a proof of concept application which takes advantage of the research and development done by botmasters, for its possible use in a number of whitehat scenarios.

The application combines well known Peer-to-Peer technologies with techniques used in famous botnets. This creates a network resilient to internal and external DoS attacks. The attacks that made former networks stop working, only cause an increase in response time and extra traffic in this network.

This thesis shows that botnet techniques can be applicable for general-purpose contexts and that combining all known protection measures it is possible to build a highly resilient network, resistant to most attacks that can bring normal networks down.

Zusammenfassung

Botnets, große Netzwerke von Computern die ein böse Master gesteuert wurde, sind heute große Bedrohungen für die Computersicherheit. Sie haben von einfache zentralisierte Netze von Hunderten Computern zu komplexen Multi-Funktions-Systeme mit Millionen Computern entwickelt. Botnets verwenden verschiedene neue Techniken, um ihre Widerstandsfähigkeit zu erhöhen und um schwieriger zu analysieren, zu verfolgen und zu steuern sind.

In dieser Thesis, die Techniken Botnets ihrer Widerstandsfähigkeit zu erhöhen verwendet, werden gesammelt und analysiert, um einen Überblick über den Stand der Technik in Verteilt Malware zu haben. Dieses Wissen wird verwendet, um ein Proof of Concept Anwendung, die Vorteilt von der Forschung und Entwicklung Botmasters gemacht hat und verwenden sie es in Whitehat Szenarien, zu bauen.

Die Anwendung kombiniert bekannte Peer-to-Peer Technologien mit Techniken, die in berühmten Botnets verwendet sind, um ein Netzwerk anfällig für interne und externe DoS-Attacken zu herstellen. Die Angriffe, die ehemalige Netzwerken kaputt gemacht haben, nur einem Anstieg der Reaktionszeit und zusätzlichen Netzwerkverkehr auslösen.

Diese Thesis zeigt, dass Botnet-Techniken können für allgemeine Zwecke und Kontexte anwendbar sein und dass mit der Kombination aller bekannten Schutzmaßnahmen ist es möglich, ein hoch belastbares Netzwerk, beständig gegen die meisten Angriffe, die normale Netze führen kann, zu bauen.

Contents

List of Figures

Outline of the Thesis

Part I: Introduction

CHAPTER 1: INTRODUCTION

This chapter presents an overview of the thesis and it purpose. The initial situation will be presented and the history of botnets will be described to have background information for next chapters.

Part II: Theoretical Background

CHAPTER 2: SOFTWARE REQUIREMENTS OF BOTS

The main software requirements of malware are presented in the second chapter. Along with each requirement, its applicability to whitehat scenarios is analyzed.

CHAPTER 3: TECHNIQUES USED IN BOTNETS

The infection and cloaking of a bot is carried on by very diverse methods, which will be explained further in the thesis in chapter three. Also, a description of the most relevant network technologies for botnets is presented here.

CHAPTER 4: FUTURE EVOLUTION OF BOTNETS

In the foreseeable future so the botmasters will keep on improving their malicious programs and the botnets will become better. Of course it is not possible to make a precise forecast of the techniques botmasters will start to use but it is possible to see some trends and evolution patterns. These will be explained in the fourth chapter.

Part III: Case study

CHAPTER 5: APPLICATION

The goal of this thesis is to develop a system that would benefit from all the innovations and field testing made by the blackhats and use it in a whitehat scenario. This application will be described and analyzed in chapter five.

CHAPTER 6: CONCLUSION

This chapter presents the final conclusions of the thesis.

Part I.

Introduction

1. Introduction

1.1. Initial situation

Cyber crime has become one of the most disruptive threats the Internet community is facing today. An underground economy has emerged for a long time exploiting weaknesses of today's Internet. More and more, they are using novel defense techniques to increase the resilience of their botnet-infrastructures. On the anti-botnet side there were a number of enhanced countermeasures proposed in order to gain control over the botnet, take it down and track the botmaster(s). But by now, past botnet-trends have shown that sophisticated botmasters are always able to evade botnet-tracking and other related countermeasures. Architectural innovations such as fast flux networks (sort of "round robin DNS - enhancement" that is traditionally used in high availability scenarios) make botnet-tracking techniques very difficult and it is nearly impossible to bring a fast flux enabled botnet down. This aggravates the permanent race between attack and defense and makes the attackers have always a lead over the defense.

1.2. Motivation

The motivation for this thesis is to take advantage of the techniques that computer criminals have introduced to make their network systems resilient. These computer criminals have managed to create networked systems of remote-controlled computers which they use for illegal activities. These computers are called bots and the networks of bots are called botnets [16, 66].

Botnets are created to keep working even under the close vigilance of computer experts worldwide [78, 61] and under the attack of rival botnets [55, 50]. These botnets execute the work they are rented for, continue spreading to new computers while their owners remain anonymous [17].

To achieve these characteristics, botnets are using a lot of already existing techniques, technologies and algorithms used in common network systems, in most homes and businesses [25, 61, 11]. Some techniques used in botnets are evolved versions of the original ones, other techniques are adapted and combined to form a platform used by the criminals.

In order to obtain a system stable and solid like a regular botnet, it is necessary to research the techniques used in other programs, develop the computer applications involved and deploy and test the whole system. Also, it is necessary to repeat this development process several times to improve and refine the final result to suit the original needs of the creator of the botnet. The result is a set of very complex computer applications [16]. It would cost a lot of time and money to develop something similar from scratch.

Botnets have characteristics that can be applied to non-criminal scenarios because some of the objectives are the same. These objectives include having a resilient and efficient

network and to provide security and anonymity.

The goal of this thesis is to take advantage of the work in researching and testing different techniques and algorithms done by computer criminals in the developing of their botnets and use it to develop a proof of concept application for non-criminal purposes.

There already are some signs that big industry players are starting to take innovative approaches similar to this, like for example twitter, that announced [2] that they will start to use P2P technology [28] to spread the information faster and more efficiently among their servers. As well, several projects already use some of the technologies, like the Coral Content Distribution Network [26], the YaCy search engine [3] or BitTorrent file sharing system [15].

1.3. A brief history of botnets

In the last years computer criminals have shifted their focus of action. Instead of using their advanced abilities to do harm for fun, computer criminals now use their abilities to make profit. Since the Internet has became a common business tool, there are a lot of money related activities taking place online. Some ill-intentioned computer experts realized that it is possible to interfere with these activities and make real money profit, creating a whole online crime industry [35].

One of the most often used means to make profit with computers are botnets [41]. Profitable activities like sending spam or doing DDoS attacks require sheer power that can only be achieved controlling a high number of computers. Other profitable activities like stealing credentials require to have local access to the computers of the users. All these requirements are met by botnets, and therefore are ideal to perform mentioned activities.

The term bot itself comes from the term robot. In computer science, bots are programs that perform autonomously some activities in online systems that are simple and repetitive. Bots can perform these activities faster and in a more convenient way than human beings. Common activities performed by bots include chat channel management or online games [85]. Bots can be benign (chat channel management) or malign (spam and attacks). This thesis is focused on the latter and the term bot will refer to the malicious kind of bot. The people that program and/or control bots are referred to as bot-masters or botmasters.

Botnets are big groups of computers connected to the Internet which have been infected with a computer malware, i.e. software unwanted by the owner of the machine it is installed and has harmful effects. The malware that form botnets has bot functionalities, hence the botnet name. Each computer in a botnet is a zombie host but often it is also called bot for brevity. The zombie host performs different tasks for the people that infected the computer. The most common tasks are sending unsolicited emails [68], often called spam, performing DDoS attacks [54], hosting phishing and scam sites [6] and gathering sensitive information and transmitting it back to the botmaster [41]. These activities are done either for the direct benefit of the botmaster or for third parties. Some criminals involved in running an illegal Internet business create botnets to help their original business. Other botmasters create their botnets only to rent the services the botnet has to offer to other computer criminals [10].

All bots follow a similar basic cycle [8]. Once a bot infects a computer, it starts a connection to a Command and Control (C&C) channel. This channel is used to communicate

the botmaster with the bots. On one end, the botmaster inputs new commands and on the other, botnets receive and execute those commands. Usually the commands a bot is able to execute are limited to a small set hardcoded in the bot. This limitation is overcome in most of the bots having a special command that allows to download and execute a program from anywhere on the Internet. This download and execute function is used to perform one-time actions that provide some profit for the botmaster like updating the bot itself with an improved version [62]. Most of the time, however, each bot is performing one of the native, built-in functions, for which the botmaster that controls the bot is making money.

The history of botnets starts in the last decade of the XX century. In August 14, 1996, the first botnet, SilverNet, was created and it was only discovered when its creator made it public. Since that moment numerous botnets appeared. The most usual network architecture for those botnets were Internet Relay Chat [60] (IRC) servers, which made neutralizing the botnets not very difficult. Taking down the IRC central server rendered the whole botnet useless, since the bots were unable to receive new commands [27]. Figure 1.1 shows a typical IRC based botnet. The botmaster and all the bots connect to the same rendez-vous server. The botmaster issues commands and the bots retrieve the commands and execute them.

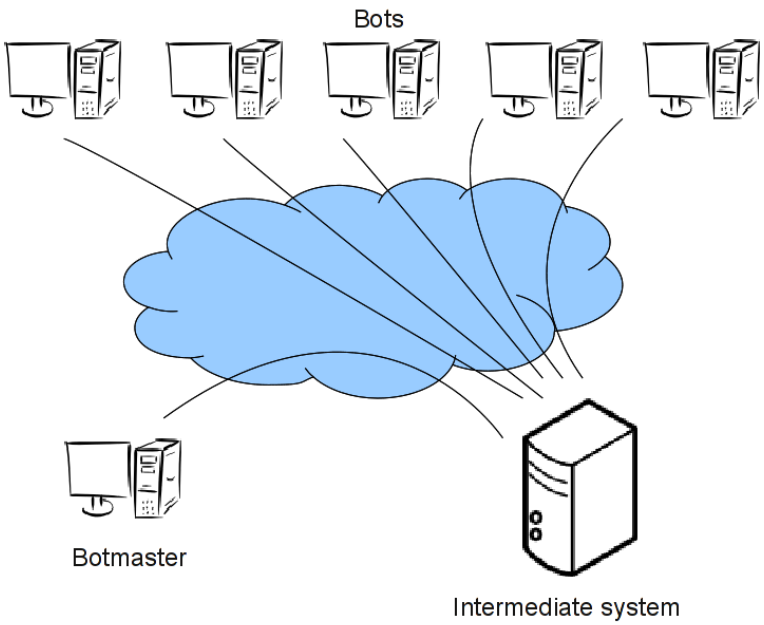


Figure 1.1.: Classic botnet structure.

The centralized design was very easy to understand and use, but since it was very vulnerable [27] botmasters started to think of new solutions to organize their botnets. To eliminate the vulnerability of the central server, botnets started to use Peer-to-Peer (P2P) technology [19]. In a P2P network all nodes communicate among themselves, thus there is no need for a central server [4], and botnets could benefit from that.

Peer-to-Peer has been known to the security community as a possible architecture for malware networks since 2005 [23]. But it until the Storm Worm appeared, it was not widely studied and analyzed. Figure 1.2 illustrates a decentralized, Peer-to-Peer network, where all nodes participate in the communication as equals and for an external observer the botmaster is just one more node in the network.

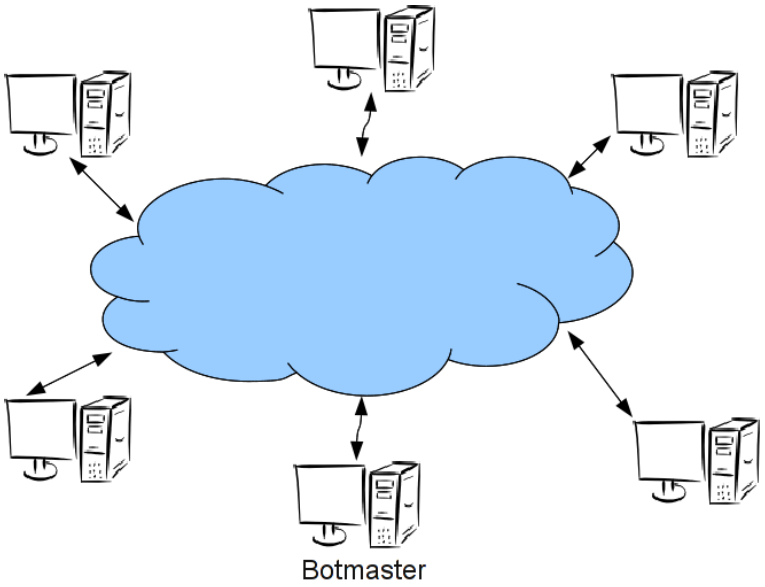


Figure 1.2.: Modern botnet structure.

In 2007 the Storm Worm (in fact, it was a Trojan) became very widespread, managing to infect more than a million computers [43]. This number made it reach the mainstream media. It was very hard to take down, since it used some new technologies, namely P2P connections to protect itself and to hide it's C&C servers. Once analyzed, it was possible to neutralize it and even take control over the bots, but it wasn't as easy as with previous botnets [61, 39].

Also in 2007, the Nugache botnet was the first to use a pure P2P architecture, staying alive for a very long period of time [23], demonstrating the advantages of P2P over IRC or other central server based botnets [79]. It was deactivated since the author of the malware made the mistake of including only 22 nodes to join the network. One those hosts were neutralized, the botnet could not expand anymore, since zombie hosts were unable to join the network.

Later, in 2009, the Conficker Worm took another approach and it started generating different domain names every day to contact its C&C servers [62]. This was a great challenge for the security industry. It took a lot of coordinated effort to take down these domains peemptively to thwart the functioning of the botnet. Conficker C, an evolution of the original malware, showed an improvement of this random domain name generation technique, as well as a clear evolution in response to the efforts taken by the security community to take it down [63]. Another remarkable feature of this botnet was the great awareness of its authors of the security and cryptography developments, not only using state of the

art cryptographic algorithms but also implementing new algorithms only few weeks after their publication, like the novel algorithm MD6 [63].

In 2010 alone there have been already cases with botnets more than 13 million strong like the Mariposa botnet [18], and the trend is for these infections to grow. The responsible people for the Mariposa botnet were found and arrested after several mistakes made by them led to their identification. These mistakes included making the botnet grow too much without dividing it into smaller botnets, thus attracting too much attention, and connecting to a command and control center without using the anonymity services in place, thus revealing the real identity of one of the botmasters. In the process, after a joint attempted takeover of the botnet by security researchers and law enforcement, the botmasters regained control of the bots and used the power of the whole botnet to retaliate against one of the security companies, based in Canada. This attack showed the potential to do harm by botnets, causing serious disruption of Internet connectivity to several ISPs and millions of clients in the country.

A clear trend in the botnet situation is that the botmasters are paying a lot of attention to the security industry and are improving their botnets accordingly [63]. After each update, the botnets are better built, harder to analyze and to take down and their botmasters harder to identify. The big black-market pressure assures that this trend will keep the same for a long time, also providing the security community with new techniques and technologies that can be applied to general purpose applications to improve these applications.

Another sign of the increasing importance and success achieved by botnets, is the mainstream media attention they are starting to get. Where previously newspapers and TV news only covered particular mass infections like the “I Love You“ virus, now they warn the public against the general threat that malware presents, making it even to the front pages [5].

Part II.

Theoretical Background

2. Software requirements of bots

2.1. Introduction

This chapter enumerates and describes the software requirements malware has and how they compare to whitehat software. The technical implementations to fulfill these requirements will be discussed in the next chapter.

Unlike last century virus programmers, contemporary malware authors focus their activities on one main goal, which is profit [56, 8]. This means that the malware itself has become of professional quality and constantly the techniques it is using become more sophisticated. It also means that the main characteristics of modern malware are very different from the old malicious programs.

As any complex piece of software, malware must target many software requirements [32] and here are presented the most important of them. Any botnet should target these requirements to optimize its functioning and most bots take care of them in one way or another [39, 62, 16]. Some of these requirements are also present in most common software systems, so this work is going to focus on those most valuable to be applied in a whitehat scenario.

2.2. Stealthiness

The usage of any computer system without explicit authorization of the owner of the computer is illegal according to the legislation of many countries in Europe, North America and other parts in the world. Even in the countries where it is not illegal, it is undesired. Since bots consume resources of the users' computers, like memory, processing power or bandwidth, users do not want bots on their computers. Thus, being stealthy is a very important requirement to be able to stay in the botnet and continue making profit.

In a whitehat scenario, stealthiness is not as critical since the presence of the software is known to the owner of the system and very frequently also to the final user. Nonetheless, some stealth techniques are useful in order to improve the final user experience with the computer and to allow him using the system without any negative impact caused by the software. This is the case of antivirus, defragmentation programs or distributed efforts like seti@home.

2.2.1. Hide presence

In order to be able to seize as many computers as possible, the malware must not trip any alarms and leave the rightful owners of the computer being transformed in zombies unaware of the presence of the bot. To accomplish this, both the infected computer itself and the network environment of the infected computer must be taken care of.

First, to avoid giving away its presence on the computer, the malware must have a mechanism to evade the antivirus and/or any other security software as anti-spyware, behavioral analysis software, etc. installed on the target system. It is necessary to avoid the detection by the antivirus to even execute the malicious code on the computer. Once accomplished at installation time, the malware make the cloaking permanent. The security software vendors update the virus signature databases quite often. If the malware was to be included in the signature database, the antivirus software could detect the infection and remove the malware or at least alert the user or system administrator of the presence of the bot. Of course, legitimate software would never alter the security software on a machine, but this does not mean that these features are useless. First, it is important for security vendors to learn what malware is doing to workaround their antivirus and enhance the products accordingly. Second, it would be very useful to use the same techniques used to disable security software and apply them to disable malware when it is detected on a system. Both these features are typical requirements of a security system so they will not be covered in this work.

It is also important for a bot to hide its own files and registry keys. Not only users can suspect if they see unexpected files on the system, but also antivirus systems will have a much harder time identifying that something is wrong if the files are well hidden. This techniques have already been exploited by other software, like DRM-enforcing software. Usually any technique that hides any information stored on the users computers from the users themselves is met with strong resistance from the community and a very bad public opinion. A very famous example for this is the Sony-DRM scandal [70].

Similar to hiding files and configuration options, the bot must avoid giving any suspicious error messages, since error messages could make the user find out about the bot and try to remove it. As well as hiding files, hiding error messages is not a desired feature in a common application.

To avoid being detected on the network the malware must camouflage its traffic. Listening to network communications and searching for usual botnet traffic signatures like IRC traffic is a very common technique used by firewalls or IDS (Intrusion Detection System) systems [51]. Bots must be careful not to use easily recognizable patterns in the traffic they generate and scramble the content of the traffic. These approaches are useful in some applications, specially when it comes to censorship or Internet usage limitations. Many governments try to restrict the access to the Internet to users[71], and any company could to so to its customers or employees. These techniques can help these users who are suffering limitations to use the network connection to any purpose they want, making it more difficult for the controlling entity to track and limit their traffic.

2.2.2. Hide size

Once security researchers have started analyzing the malware, it is important for botmasters not to attract too much of their attention. The longer the botnet is not under attack from security researchers and law enforcement agencies, the longer the botmaster can profit from it.

There are many different pieces of malware created and released every day, recent studies have shown that probably more than of legal software [81]. Much of this malware has bot functionality, and only some of them become big botnets. Therefore, if a botnet is

not known to be big, then it can perform its owner's deeds without attracting too much attention, since all effort will be directed towards bigger botnets.

If the size of the botnet becomes known and it's bigger than the average botnet in that moment, more researchers will start to analyze it and more resources (time, money) will be dedicated to eradicate or disrupt it. Security software companies will focus their efforts on eliminating this one botnet creating specialized tools that focus only on it.

The size concealment does not have a widespread application elsewhere, only in specific spots like number-crunching collaborative competitions or copyright protected file sharing.

2.3. Confidentiality

Botnets cannot have interact with physical good and are making profit only with information, as it is the only resource they can access, manipulate and process. Confidentiality does not help in the tasks of spam or DDoS attacks, but it is very important in identity or intellectual property theft or industrial espionage. Very often their profit is made because the information they access is unique and valuable and nobody else has it. Thus it is very important in order to make profit from it, not to disclose it to anyone except the botmaster. Stolen documents with intellectual property, industrial secrets or stolen online credentials become useless if someone intercepts and uses them to make their own profit first.

Also, the more information is encrypted, the harder it is to fingerprint the malware, either from the traffic on the network, files on the disk or processes in the core memory of the computer. This is due to the fact that no recognizable information is readable in clear text to be matched against a database. Only encrypted data and the encryption algorithms to use it reside in memory.

All these reasons make it very important that the information in the botnet must be secured and encrypted so nobody can access it once it leaves the origin computer towards the Internet.

Confidentiality is very important in numerous everyday applications as for instance online banking, and at very desired in communication or data storage. This high demand is the reason why there are many different confidentiality mechanisms available in the software market like S-MIME or PGP and botnets usually do not introduce any innovations in this area, apart from using novel algorithms proposed by experts but not yet proven secure.

2.4. Quick response

Another very important income source for the botmaster is to rent the botnet to third parties for their own use [56]. This use is mainly either sending spam or performing Denial of Service attacks [41]. To perform their tasks in an efficient way, bots must be coordinated and respond to commands in seconds. In the case of sending spam, the earlier they start sending spam, the quicker they'll finish the amount paid for, so they will be available for the next paid task, although the sending rate is not high by itself.

In the case of DDoS attacks is even more important. If the bots are not coordinated and do not execute the DDoS command at the same time, the uncoordinated attack can fail.

The resource (bandwidth, processing power, memory) drainage done by the bots will be spread over time, allowing the target to handle the load. As a result, the target's resources will not be exhausted at any point and the DDoS attack will not succeed.

In general purpose applications, to have quick response is very important as well, especially when user interaction is involved like web browsing or content searching. Except in cases of bulk file transfer or email, responsiveness is a desired requirement in many applications.

Botnets in the wild and other proposed ideas for botnets offer some novel approaches to coordinate massive networks comprised of up to millions of hosts. Big scenarios as these present some difficulties to the development process of coordination solutions. Such big networks are rarely available to test any application and when they are, it is associated with great building and maintenance costs. The solutions developed around botnets are therefore extremely useful as these solutions have been tested and redesigned for several iterations and have proven their suitability for big scenarios.

2.4.1. Fast propagation of commands and data inside the network

Fast propagation of commands and data is related to the quick response aspect. To achieve a quick response there must be some mechanism to propagate the commands and the necessary data among the bots, so fast propagation of commands is a prerequisite to have quick response.

Quick response is not the only reason for the requirement of fast propagation of commands and data. It applies also for the distribution of common data among bots, as it occurs for instance in the case of a update to the bot software.

Botnets have shown that it is possible to implement an efficient data distribution system with millions of participants [39, 18], which is a very desirable capability for any network application that needs to share data among nodes.

2.4.2. Fast infection and propagation of the bot to other hosts

A very desirable feature would be that the botnet can expand itself quickly if needed. The faster botmasters have a big number of bots under their control, the faster they can start to make money with the botnet. And if a part of the botnet disappears (operating systems are reinstalled or disinfected, IPs are blocked, etc) the remaining bots should be able to re-propagate themselves to other hosts to keep the botnet alive at the desired size.

Apart from the property of fast command distribution, this requirement focuses on exploiting techniques and keeping infection vectors up to date, which as in any software requires manually creating updates. Updates are created to react to some need or weakness of the software and the solution must be manually programmed and released to the network.

In whitehat environments there is no need to attack or exploit other systems, since such an activity is illegal in many countries and morally questionable. Also, botnets do not offer a innovative update solution, since the bot functionality is the payload of some other infection vector. This area, although interesting from a security point of view, has little to contribute to general purpose applications and will not be examined in this thesis.

2.5. Availability

It is usual that any botnet will have unstable or unreliable bots. Either caused by attacks from other botnets, law enforcement taking down servers or just home users turning their machines off, bots will disappear from the botnet with no previous notice. This unpredictability is why it is important to have a good load balancing and distribution system. Since the platform is not stable, the workload should not rely on any single node. When a bot disappears from the network, the botnet must detect the missing bot and assign the task to another bot.

This way, bots that are under low load can start to perform the tasks assigned to the bots that leave the network so the whole task is carried on with as little time waste as possible, and with high availability as a whole system.

In a whitehat computer scenario the network is usually more stable than in a botnet, as the nodes are in the network by choice and are willing to cooperate with the system. In a mobile network scenario this is not true and hosts leave and join the network very dynamically. Either way, downtime is always a possible threat in any scenario. Therefore, a load balance and distribution system is a requirement in any application that aims to have a high availability. Botmasters have developed and tested techniques that use naturally unstable nodes to form a reliable platform, so it is interesting to adapt these techniques to take advantage of them in potentially more reliable base networks.

2.6. Anonymity

One requirement of paramount importance for the botmaster is to remain anonymous. The only weak point botmasters cannot workaround are themselves. If a botmaster gets caught, the botnets he owns might no longer operate or at least, he might not profit from the operation of the botnet.

But on the other hand, botmasters also need to provide commands to the botnet on a regular basis in order to execute orders, so somehow they must find a way that they can connect to the botnet, send the commands, retrieve data from the botnet and not get caught in the process.

In some countries, law enforcement can ask the ISPs for the records of any IP address at any time and most companies also maintain a log of accesses to their servers, so any logged botmastering activity with their IP can be traced back to their persons and identify them.

To avoid getting caught, botmasters must avoid giving away any personal information such as social security numbers and real names, which is easy, but also they must prevent law enforcement from finding out the IP address of the computer used to issue commands into the botnet.

Botmasters have developed and deployed mechanisms to hide the real IP address they use to connect to the Internet and also to hide the control infrastructure itself, like Virtual Private Networks (VPN), P2P structures or onion routing services like TOR [22].

These anonymity mechanisms are suitable for many other uses. Not only for avoiding censorship in countries with restricted Internet access, but also for general applications where users desire high levels of privacy. Having an anonymous channel to interact with

an application allows a user to share only the amount of data about himself that he explicitly permits.

2.7. Authentication

With no authentication, the bots would be forced to accept any information or command somehow injected into the botnet, as they would not have any mean to tell it apart from a botmaster's command. The botnet needs a strong authentication mechanism so the botmaster can avoid a rival or any researcher taking control of the bots. The bots must accept only information (data, commands) that are valid and authenticated.

With anonymity comes a greater difficulty in authenticating the messages. Anonymity complicates the use of traditional authentication mechanisms like IRC protection or fixed IP addresses, because this traditional authentication methods used in the first botnets do not offer big anonymity guarantees. Another problem is the low resilience of traditional authentication, since it relied on single channels easy to take down, so reliability mechanisms also interfere with authentication. Solutions described in the next chapter that are applied in botnet environments are taken in general from the industry standard solutions and therefore there is little to learn from botnets in the authentication realm.

2.8. Size assessment

This is not a critically important requirement, but a very useful information to have. Assessing and knowing the sheer size of the botnet is a powerful argument in favor of the botmaster when selling services. It is also very useful in order to know when to expand and when to segment the botnet, to avoid either attract attention or jeopardize the existence of the botnet.

With some architectures this is trivially easy, like with IRC controlled bots, where it is enough to just count the number of participants in the C&C chat channel. With some others like complex P2P systems it is much harder to know the real size of the botnet, even for the botmaster himself, since a great part of the network does not communicate with him directly as will be described in chapter three. The solutions for the size assessment problem are useful since assessing the size is a form of meta-information in the system and can be applied to other uses in various forms.

First, the original size assessment is useful for accounting and management purposes in any system. Size information can also be useful as an indicator of the state of the whole system, if the networks shirks too much it may be an indicator of something going wrong.

In addition to the size, the introspection methods may be adapted and used for other purposes, like knowing the physical composition of the network, information availability or any other meta-information suitable to the application the owner of the network.

2.9. Conclusion

In this chapter the main software requirements of bots and botnets have been listed and explained. The applicability of the requirements to white hat scenarios has been analyzed

and the implementations which will be explained in the next chapter have been introduced.

3. Techniques used in botnets

3.1. Introduction

In this chapter the techniques botmasters use will be analyzed, with special attention to methods and technologies that can be applied in whitehat applications. Specifically, the techniques and methods presented here are used to fulfill the software requirements presented in chapter two. Special attention will be given to those requirements that are resolved in an original way in botnets and are useful for their use in other applications.

3.2. Bots

For the sake of completeness, this section will describe techniques used specifically by the bot software locally, i.e. not interacting with other bots. Most of these techniques are typical for malware and its application to whitehat scenarios is limited.

3.2.1. Infection

In order to create a botnet, botmasters first have to manage to install their malicious software on victims' computers. The goal is to be able to run a malicious executable code to bootstrap the infection process. There are many different ways to accomplish this [83, 36], and here are presented the most usual ones. Most techniques use a similar multi-staged installation procedure [34].

This installation procedure just copies the bot executable files to the computer permanent storage and executes them. If the user lacks administrative privileges then the malware will use software or operating system flaws for privilege escalation to obtain said privileges. The malware will then configure the system to execute itself on every system boot and it will initialize all its functions, starting by hiding its presence in the system.

Although first botmasters used single infection methods, and even modern malware is sometimes single-vectored [39], nowadays it is common to see malware that exploits several of these methods to propagate faster and more effectively. Also it is becoming more common to have a deployment program that, once executed on a machine, scans the software present on the computer looking for vulnerable versions of the programs installed and then download the appropriate exploit code. This helps reducing the overall size of the bootstrapping stage of the malware which in turn helps avoid detection by anti-malware solutions.

Network service exploit

An exploit is a piece of code that takes advantage of a software bug in some other software. In the case of infection by remote exploit, the target computer must have some software

running that listens to Internet connections and has some flaw that allows remote code execution. These vulnerabilities allow an attacker (either the botmaster or another bot) to send some specific malformed information to the vulnerable application in order to make the application execute malicious bootstrapping code and infect the computer. The malicious code is included in the data sent to the application. The most common flaws that allow remote exploits are buffer overflows, format strings and SQL injections.

Since the start of the widespread use of firewalls on client computers and Network Address Translation (NAT) devices like home routers, the bulk of the computers of the Internet is protected against this vector [67]. This technique is still effective, and therefore very dangerous, against server computers that must have a public IP and running processes in order to offer services to remotes computers, like email gateways, web servers or DNS servers. These services must be kept up to date because any published bug will be actively used by blackhats in a very short period of time after being published.

Desktop software exploit

When a flaw is discovered in a software that usually is not open to other machines, it is necessary some interaction with the user to trigger the flaw and exploit it to install the malware. Frequently this interaction is the normal usage of the exploited program. The usual attack targets are web browsers, email clients and office document editors or viewers, because those are the most widespread applications in workstation computers [57].

To exploit a flaw in one of these desktop programs, blackhats must trick the user into doing some action that unknowingly will start the installation of the malware. Such actions vary depending on the flawed program but they always include some malformed content that makes the program fail and execute malicious code. These actions include opening emails, visiting web pages, showing images or opening documents. The infection mechanism is exactly the same as in the network service exploit case, with the bootstrapping code being embedded in the accessed data.

Trojans

The next level of user interaction needed is required by the programs called Trojan horses or just Trojans [80]. In this case the user is required to actively execute the malicious program to infect the target machine. In the case of Trojans no software flaws are directly involved in the bootstrapping part since the user run the malicious code directly as any other software in the system. This requires to make the users believe that the program they are executing is something else, like a funny joke, a data (image, text) file, a new useful program or a multimedia player codec or web browser plug-in update.

To make the user execute the program, malware creators use different techniques. The most common is a mild form of social engineering, sending generic messages with the malware and a false description of the content, usually describing it as something that will attract the attention of a wide sector of the users like humorous or adult content or spectacular news headlines. Sometimes the program is disguised as a data file by using an representative icon for data like images, music or videos. Finally, some malware authors offer the fake content on a web page but state that viewing requires an update to the client software. The proposed update is in fact the malware.

Autorun

Another possible infection vector is the traditional virus approach. The program installs itself in a removable media, usually USB Mass Storage Units like external hard drives or USB flash drives. The malware uses a function provided by the operating system that allows to automatically run a program contained in the storage in order to execute itself, thus running the bootstrap code and infecting any machine the storage media is plugged into. This other storage media turn into infected media that can potentially install the virus on other machines in the same way.

Pulling

In some cases, the malware is installed on a computer by another piece of malware already present on the computer. Usually it is an update to an existing bot executable in order to improve and extend the functionalities and fix possible errors in the bot. In some other cases the existing botnet infrastructure is used to deploy some other kind of malware [62]. This is only done when the new installed program doesn't conflict in functionality or purpose with the former one, since the base botnet is ruled by a profit basis. If the new malware would perform similar tasks, both program would have to compete for the resources on the infected system, thus reducing the profit for the original malware. This programs include scareware like false antivirus, adware or identity theft applications targeted to specific web pages or services.

3.2.2. Obfuscation and self protection

A very important goal in order to keep making profit as long as possible is to keep the infrastructure of the botnet intact. To make sure the bots remain in place, they must be as silent as possible to be undetected on as many systems as possible. The longer the malware operates without notice the more profit the botmaster can make per single bot, thus optimizing each infection and potentially spreading the botnet further. To achieve these goals the bot software must hide both its presence on the infected system as well as the traffic generated on the network.

Rootkit technology

Rootkit technology is based on the concept of integrating a software into the operating system code in order to intercept system calls. The interception is achieved by changing the pointer in the system call table to the rootkit code. After performing any operation the author of the rootkit designed, the software executes the real operating system routine to achieve the expected behavior of the call. The software that uses rootkit techniques is often also called rootkit.

Since system calls are executed via traps, a form of a processor exception, the rootkit code is executed at kernel level, giving it unlimited access to all the hardware of the computer and all the data structures of the operating system. This makes the rootkit potentially undetectable from the local machine, an offline analysis is needed to detect the malware.

Such analysis are usually executed booting the machine from an external bootable storage and running the appropriate security tools to find the rootkit on the disk.

Bots very often use rootkits to hide its presence to the local user and anti-malware programs. When any other software uses system calls intercepted by the rootkit, the malware code processes the call first and if the call was about to reveal information about the malware itself, it filters the results to return only the information that the operating system would return if no malware was present in the system. This way, all system call results appear to be clean and the malware can carry on its task without giving any clue to the legitimate user of the computer.

The calls the bot intercepts depend on the complexity of the rootkit. As a general rule, they involve file system, process listing and network information system calls, to cover all traces of the bot and the bot's activity [37, 70].

The most sophisticated methods are to load the malware before the operating system itself, installing it in the Master Boot Record (MBR) of the boot disk. This allows the malware to have complete control over the computer. A toolkit using this technique has been used to create bots like Torpig [78].

Regenerate executables

Most anti-malware software work based on heuristic pattern matching against certain data files called signatures. To avoid being detected by such software the malware code is regenerated and scrambled periodically so when it is downloaded to a new system it is not recognized by the anti-malware solution [42].

The encoding and scrambling techniques are highly successful and a known piece of malware recoded with these techniques is very often undetectable to any search engine on the market [14].

Since the anti-malware software updates its signatures periodically, the regeneration of the executable must occur more often than the usual update period for this software in order to avoid detection. The antivirus databases are updated one or twice a day and the malware is re-encoded as often as once per hour.

Disabling services

Related to the previous point, in order to be undetected by security systems, one of the first tasks by any malware is to disable the security software itself. The re-encoded executable file is not detected by the antivirus, but as soon as the malware starts performing the installation procedure, the security program will detect it and alert the user. The re-encoding only protects the bot from being detected in the very first stage of the infection. In order to complete the infection process, the malware must first disable all software hostile to it. usually, debuggers and other software analysis tools are also considered hostile to the bot, since the botmaster want to avoid the bot being studied by security researchers.

The malware goes through the list of active processes in the system and matches the names in the list with a internal list of well known security applications. In case of a match the bot either terminates or neutralizes the process. Only after making sure the computer has no hostile software, the bot continues activating and executing the main code.

Another similar feature of most bot programs is deactivating other programs or operating system services that could reveal its presence. Very frequently the update service is disabled to avoid installing updates that could eliminate the malware or alert the user. Also, the bot itself is a computer program and as any computer code it can contain errors and bugs. In addition, disabling other programs may also result in unexpected errors and crashes of those programs. Some malware also disables error reporting systems so the users remain unaware of the activities of the malware on their machines [13, 78].

Cryptography

While completely controlling the computer which it runs on, it is not possible for a bot to hide the traffic because it can be seen from other computers on the network or any system on the route of the traffic.

After an thorough analysis of a botnet it is possible for the researcher to reverse engineer the network protocol that it uses. This allows to quickly identify which hosts on a network are part of a botnet and react accordingly, either disconnecting those nodes, try to disinfect them or warn their users to do it themselves, depending on the clearance on the infected computers that the researching person has.

To avoid this automatic analysis many botnets use cryptographic algorithms to cipher the traffic and thus make it opaque to external observation. One such example is the Nugache botnet [46]. Using well known methods like SSL tunnels, the content of the traffic can only be known at the endpoints of the connection. One endpoint is the infected computer, which can be tricked by rootkits as seen before. The other endpoint is very often also an infected machine or a computer controlled by the botmaster, out of the analyzed segment of the network.

Blacklisting IPs

The cryptography used in the traffic avoids automatic detection of the botnet traffic but it is a desired requisite that the protocol itself is not known at all in the first place. A very common method of analyzing botnets is the usage of honeypots.

A honeypot is a system closely monitored and controlled which is used as a bait for different kinds of malware. It can be either exposed to the Internet to look for new malware or infected on purpose with known malware to analyze the behavior of said malware under controlled conditions. To avoid landing in a honeypot, and thus avoiding raise awareness of themselves, many botmasters equip their bots with a list of known security research IP addresses. The bots then avoid scanning or infecting those addresses, not to interact in any way with a honeypot and reveal their presence or even existence.

This technique also prevents some kinds of behavioral analysis. Many IDSs look for traffic with unusual destinations inside networks. Those can be multicast addresses, private subnets not in use by the organization, etc. Avoiding generating traffic with those destination addresses helps to avoid raising alarms and being detected.

Blacklisting domains

Blacklisting domains is a technique which combines the methodology of rootkits and the concept of IP blacklisting. The malware first installs a rootkit component that intercepts the domain name resolution service of the operating system. When any application executes the gethostbyname() call, the malware checks the domain name with a list of known computer security related domain names. If the requested domain name matches a domain on the list, the rootkit component does not call the original operating system call and returns a fake address instead. The returned address can be just an invalid address or an attempt to attack the user, trough credential theft, phishing or installing other malware [62].

This way, the user of the infected system or any tool executed on the system are unable to download any antivirus update, security scanner or malware removal tool.

3.3. Architecture

The architecture of a botnet is the way the bots are organized and communicate among themselves and receive commands from the botmaster. It defines its longevity, scalability and robustness. The architecture of the botnet should be chosen to maximize the performance of the bots and maintain all bots connected among them or at least with the botmaster, ready to receive and execute commands and capable of resisting attacks from the outside. There have been different techniques in use since the first botnets, and the successive techniques progressively improved the functioning of the botnets that used them.

3.3.1. Centralized networks

The traditional botnet architecture was a centralized network with a command and control server to which the bots connected every predefined amount of time to obtain new orders [8]. The most typical way of a C&C server was via an IRC channel, usually password protected. This architecture is very easy to implement but has a very weak single point of failure. If the central node fails, or is brought down by law enforcement or a rival, the botnet looses its command center and therefore is rendered useless, since bots cannot receive commands and stay idle.

3.3.2. Peer to peer networks

In order to avoid weaknesses of centralized structures, botmasters started to use a more complex architecture, but also more resilient against failure of a command and control node, constructing a peer to peer network among the infected computers to distribute some key information.

Peer to peer (P2P) networks are a type of network where the nodes that share resources (bandwidth, disk storage, or others) are organized without a traditional hierarchy among them. All nodes are servers and clients at the same time, and maintain connections with a number of other nodes to form the network [4], as shown in Figure 3.1. This way they form an overlay network over the underlying physical one. This approach has the advantages

that a failure in a node doesn't affect the connectivity of the whole network. Any failing node is only one more of the whole interconnected system, as in many P2P networks all the nodes are at the same level. This property allows nodes to constantly join and leave the network without any major impact in the overall connectivity and usually also on the content availability.

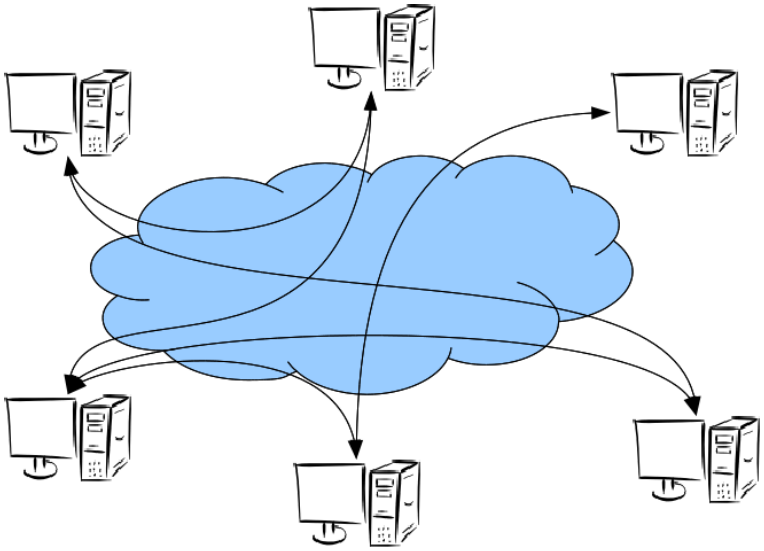


Figure 3.1.: In a P2P network hosts connect to each other, there isn't a single point of failure.

Another very important advantage is that the combined network bandwidth of the whole network is much greater than in a centralized network, since it is not limited by the resources of one unique node. All the nodes are servers as well as clients, so unlike in a traditional client-server differentiating network, the more people request a resource, the easier and faster is to obtain it for each one, since all the other clients are also servers from which to obtain the required resource. As an additional effect, once at least one copy of each part of the resource has been transferred to other peers, the original provider of the resource can disconnect. The content will be still available in the P2P network for the rest of nodes to retrieve a whole copy.

Additionally, since there is no node hierarchically higher than others, adding more peers to the network doesn't overload any node. Even nodes with low resources can participate in the big networks serving highly demanded content. The load will quickly spread to other nodes as soon as those nodes start having parts of the content.

This leads to a great scalability of the entire network. As each node is connected to only a subset of the other nodes, usually fixed size subset, adding nodes to the network doesn't negatively impact any node in particular or the network in general. Quite the opposite, adding nodes improves the overall connectivity, usable bandwidth and availability of resources.

An ideal P2P network can be considered offline only when all its nodes are offline, thus giving botmasters a huge potential to keep the botnet alive even under the attack of rival botnets or law enforcement disconnection efforts. There are many different P2P protocols

that are able to achieve these ideal objectives to different degrees of success. A well designed and implemented P2P network should be able to withstand the disconnection of many nodes without suffering an impact on the global inter-node connectivity, thus becoming close to the ideal.

Peer to Peer networks can be categorized by two mayor aspects: whether the network uses a central server and by the way the nodes organize among themselves.

Centralization According to the usage of a central coordination server, P2P networks can be centralized or decentralized.

Centralized P2P
Some of the P2P networks need a central server to operate. The payload traffic is always among peers, but some protocols require the participating nodes to register, search for content or be coordinated by a central node. These networks still benefit from the higher throughput of the P2P technology, but don't share the resilience benefits since any disruption of the normal functioning of the central node affects the whole network.

Decentralized P2P
Other P2P networks are completely decentralized and need no central node to function normally. All nodes are completely equal and the resilience of the network is greatly improved since any single node can be disconnected without affecting the network as a whole.

Structure According to the internal node organization, P2P networks can be structured or unstructured.

Unstructured P2P Historically, there have been P2P networks with no structured organization. In these networks connections were established randomly among the nodes and the searches were performed flooding all peers with requests. This has two mayor disadvantages: time and traffic.

First, since all connections are performed randomly, the distance from a node performing a search to a node with the information required is unknown. The search may take a long time to reach the destination node, or, for rare content, not at all.

Second, each search generates a lot of traffic in the overlay network, since the search request jumps from nodes to nodes that have no relation with the information searched.

Structured P2P Decentralized networks can be optimized by adopting a structured approach, which assures a methodical way to obtain the search results. These P2P networks usually use a technique called Distributed Hash Table (DHT) to coordinate and perform searches in the network. These networks benefit from a very high resilience since there is no significant node to take down. On the other hand, other tasks are slower or more difficult, like joining the network or acquiring peers.

Finally, there are P2P networks that combine both approaches. They implement a decentralized management but also allow central servers or supernodes to help organize the network and speed up searches. When the central server is available the

nodes use it to register, find peers and perform content search. Once connected to the network they also exchange information with the peers they are directly connected, and continue to do so even if the central node is no longer reachable.

DHT

The acronym DHT stands for Distributed Hash Table. It is a very common way to have decentralized, structured P2P networks. The general idea, as its name implies, is to have a Hash Table distributed over all the systems that form the network. A Hash Table is a data structure that is used to store data under a given key. It is usually the internal implementation that programming languages choose for associative arrays.

Hash tables apply a hash function on the key and store the associated data at the position given by this hash of the key. If the hash function is of high quality, i.e. the resulting hash is a evenly distributed pseudo random result (see Figure 3.2), the hash table is a very efficient data structure. For a storage space that is sized correctly, the hash table usually allows O(1) data storage and access times. A correctly sized storage space is one that keeps some free space under the maximum planned amount of storaged data. Linear lookup hash tables, the most efficient implementation for low loads, perform with O(1) until the storage space is 80% full. If the hash table is used above the 80% threshold its performance starts to degrade to O(n). Other implementations like chaining hash tables degrade their performance in a linear way keeping a O(1) complexity, but the performance of chaining implementations is lower under the filling threshold and the implementation is more complex, requiring the use of dynamic memory.

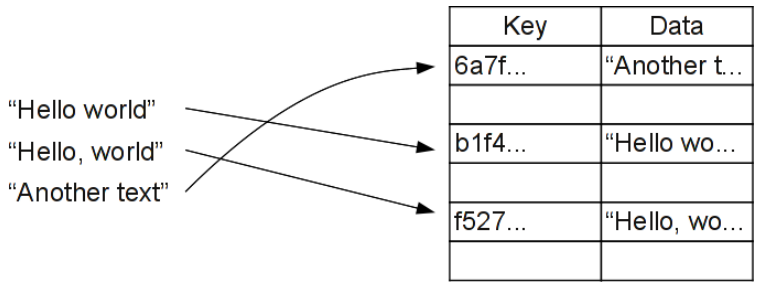


Figure 3.2.: To have uniform distribution, minimal data differences give totally different hashes.

Distributed hash tables apply the concept of a hash table to an overlay network. Each piece of data is to be found in one precise location, determined by applying the hash function to it. This location concept makes DHT a very efficient distributed data structure, since lookup are finished in a finite time, as opposed to flooding searches.

Each node joining a DHT generates a random ID with the same length as the hash length used in the system. This ID is very important to the DHT and is used in all the operations of the system. A distance between two nodes in the DHT is defined as the number of bits that are equal in their IDs, with the exact definition depending on the implementation. Usually it is defined as the number of most significant bits that match each other.

After joining the DHT, the node starts connections with other nodes. These connections are not set up randomly, but with a closeness criterion. Each node has a significant number of connections to nodes with small distance to the node itself. Also, the node sets up some connections to more distant nodes to speed up lookups reducing the number of jumps to reach a distant node.

The location of the data is also performed using the concept of distance. When a new piece of information is to be placed in the DHT, the owner of the new information first calculates the hash of the data. Then it performs a lookup among the active connected peers and chooses the one with the smallest distance to the data hash. Next, the node iterates asking the previously selected peer about the closest node to the hash among the peers of that preselected node. When it reaches the closest node to the hash it can find, it stores the information about the piece of data in that node. This results is the information being spread throughout the network, as shown in Figure 3.3, thus having the load spread and not on a single node.

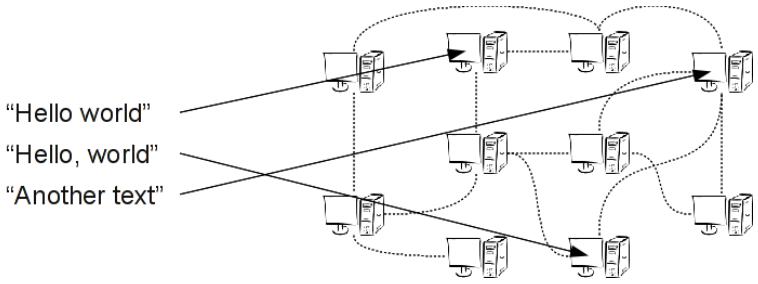


Figure 3.3.: The data for different keys is spread evenly throughout the network.

When another node wants to retrieve the information, it performs a similar process. The node iterates through different nodes via peers to reach the closest node to the hash of the desired information. Once a node is identified as the closest to the hash, it is asked by the searching node for the information about the data. If the found node has the information, the search is successful and the data is retrieved according to the information in the node. If the found node doesn't have the information, the search is considered as failed and the original node can inform the user that the data has not been found. The whole process executes in a finite time and gives back a conclusive result. The data is either retrieved or confirmed as missing [30, 48].

Kademlia Kademlia is a particular implementation of a DHT used in many P2P networks, like BitTorrent, Overnet or eDonkey. As any DHT, Kademlia allows a completely decentralized, structured operation of the network.

It starts by assigning to each node a random 160 bit ID. The ID can be generated per session but it is recommended to preserve the ID between sessions. The system relies upon the belief that the IDs will be uniformly distributed, so the IDs are generated with a random number generator.

In kademlia distance is calculated as the integer resulting of XORing two IDs or an ID and a hash. Each node maintains a routing table with peer IDs classified by distance. The

peer information (IP, Port, ID, etc) is stored in a different list depending on the distance between the two nodes. Each list is called a k-bucket, being k a variable parameter that defines how many contacts be put together. All peers that a node discovers or communicates with during the functioning of the session are placed in queues in the corresponding k-bucket. Then, all the peers in the k-buckets are checked via a PING message. If a peer is no longer online, another peer from the k-bucket queue is placed in the place of the disconnected node in the k-bucket and becomes an active contact.

Each piece of data in the network also has a 160 bits ID, obtained through a SHA1 hash function. The locator information (which nodes contain the actual data) is stored in several nodes with a close distance to the data ID. This is done for redundancy, to allow some nodes leave the network and still preserve the information. This multiple placement also improves the performance of the searches, since the more nodes contain the information, the shorter it is needed to look for it. To improve the resilience even further, the locator information is republished by every node that has the data every hour and the original publisher every 24 hours.

The search is performed by obtaining the ID of the data and querying recursively the nodes that are the closest to the data ID. The algorithm starts looking for the closest peer to the data ID among the local contacts in the k-buckets. This peer is queried for the closest node to the target ID among its peers, and the operation is repeated until no closer node can be found. Then the closest found node is queried for the locator information of the calculated data ID, and the response is final, either positive or negative. The algorithm assures a relatively fast search compared to flooding search and guarantee of conclusive results, whether the data is present in the network or not [49, 65].

The Kademlia protocol is vulnerable to the Sybil attack [65]. This attack is a type of denial of service, where a group of attackers can deny the access to a key in the DHT.

In the Sybil attack, the attacker nodes join the network with IDs close to the key they want to eliminate. When other nodes look for the information at the key under attack, it is highly probable that the search will arrive to an attacker node, since the distance to the key is very small. The attacker nodes will this way intercept all petitions for the information and it will become unavailable [24].

Examples of relevant P2P systems

Peer to peer networks have had a big impact in the recent usage of the Internet and many instances of malware have used them in one way or another. To achieve a better understanding of the current situation regarding the topic, here are presented some examples of P2P networks, P2P relevant concepts, and their relation with malware when applicable.

BitTorrent BitTorrent is the most popular P2P protocol nowadays. Different reports claim that up to 50% of the Internet traffic is caused by peers exchanging files using applications that implement the BitTorrent protocol [40].

The system works with a central server, called Tracker, to coordinate all the peers that have one specific file. The search is done off the network. A user wanting to download some content must first find by other means a .torrent file containing the description (metadata) of the content.

The .torrent file contains a list of trackers that coordinate peers that have the content and the partial checksums of content chunks. This file is provided to the software client which contacts one of the trackers from the list, registers with the tracker, receives a list of peers that participate in the sharing of the content and starts downloading different parts from different peers that already have portions of the required data.

Since the protocol uses a central server, it is not as resilient as other P2P protocols. A piece of data is usually tracked by several trackers in order to achieve some redundancy in case the original/main tracker stops working.

Also, a DHT-based trackerless approach has been proposed and implemented by many clients, but it is not the usual use case and many users ignore that this option even exists. With this option, not only the tracker maintains a list of peers participating in the exchange of a particular content, but each peer does this tracking [15].

FastTrack FastTrack is the P2P protocol used by the famous file sharing program Kazaa, among others. The network itself has not been frequently used to form a botnet, but it became famous for the amount of malware hosted as shared files [72]. Since around 2003 it was the most popular file sharing network and the hash used, UUHash, favored speed in spite of completeness, the network soon became full of false content that was in fact different type of malware.

The protocol has not been made public but reverse engineered from closed source programs. It is known to use supernodes to improve connectivity and scalability and only the supernode-client part of the protocol has been widely explored.

Overnet The Overnet P2P protocol is a decentralized protocol, i.e, it needs no coordination nodes. It became very famous in the computer security scenario when the Storm worm used it in 2007 as an extra layer to protect the Command and Control channels [39]. Thanks to its decentralized operation, when the copyright associations managed to take down all proper Overnet resources due to copyright infringement in September 2005, the network itself could still operate, although with limited functionality.

It was only a year later after the takedown that the Storm botnet used Overnet as its P2P layer, thus demonstrating its fitness to operate without any centralized nodes [61].

The Overnet network uses kademlia as the DHT system.

GNUnet GNUnet is a software framework developed under the GNU project, that allows using a anonymized, encrypted, decentralized P2P network [33]. It offers encryption services, peer discovery and resource location. It also allows to create a Friend to Friend overlay network. Some of the services were vulnerable to censorship and identification [44], but were addressed with new encoding techniques. Other than that, GNUnet offers all these services as an Application Programming Interface (API) for other programs to use. Practical applications based on the GNUnet include anonymous file exchange services and a chat proof of concept application.

GNUnet offers also a flexible anonymizing protocol, gap [9]. This protocol allows each node to determine the amount of anonymity desired independently of the rest of the network.

A proposed novel enhancement for GNUnet is the bootstrapping protocol, migrating from a system based on lists of bootstrapping nodes to a statistical, DNS based system [29].

Magnet URI scheme The magnet URI scheme is an open standard defined to create links to locate resources in P2P networks in an universal way. It is a Universal Resource Name rather than Locator, since it uses a hash of the contents of the file to identify it. It is up to each P2P network to provide the locator services to find and retrieve the file. This has significant advantages. Among them, it allows to publish a multi-network link, and it can be published and shared by any one, anywhere, in any way [1]. It is an example of how P2P networks, although separate and different in the inner functioning, can be combined and used together to achieve a specific purpose.

Other P2P applications The P2P networks presented here are only a small fraction of the P2P networks existing on the Internet. There are several other popular P2P networks which serve to many different purposes. Many of them are used for file exchange but there are other, with very different uses.

FreeNet is one of those, it used for anti-censorship anonymous communication with plausible deniability built-in. The Coral Content Distribution Network uses P2P technology to provide a replication service for sites under heavy traffic [26]. The YaCy system provides a search system with a P2P base, capable of indexing content unavailable to ordinary search engines [3]. Others, like SopCast or TVAnts, offer almost real-time multimedia signals via P2P mechanisms, suitable to retransmit live TV content.

3.4. Segmentation

In order to confuse security researchers and as a way to attract less attention, it is usual to segment the botnets into smaller botnets. This can be achieved by design or even on a live botnet, as it happened to the Storm botnet [39].

The segmentation is the process of dividing one big botnet into other smaller botnets. Those smaller botnets are comprised of the same bots that were part of the original botnet, but are organized in a different fashion. After the segmentation, bots from one botnet no longer have contact with bots from other botnets, but the same botmaster retains control over them, as seen in Figure 3.4.

The C&C channels of the different resulting botnets become different as well. Monitoring one channel doesn't reveal any information about the other botnets, not even their existence. Size assessment analysis done on the channel only reveal the size of the analyzed segment.

Segmentation has very little negative effects and numerous advantages. It allows to sell some botnets as a whole instead of renting them. Segmentation also prevents security breaches in one botnet to expand to all the former nodes.

Segmentation can also be done to specialize the botnets. It allows to have for example bots with high processing power and bots with high bandwidth in separate networks. This way, different botnets can be used for different tasks, maximizing the profit from them.

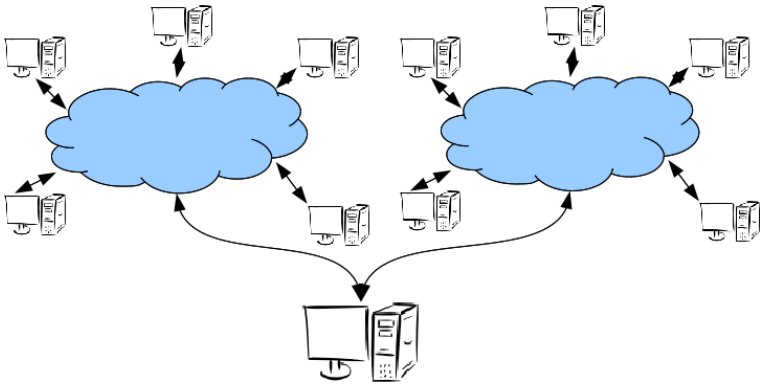


Figure 3.4.: After segmentation, the same botmaster still controlls all the bots.

Since it is still possible to devote several smaller botnets to one purpose, and they will behave as a larger botnet, the are virtually no downsides to this approach. The biggest inconvenient is the additional management overhead of having to issue multiple orders for one task.

3.4.1. Size assessment

In traditional central-server-based botnets, size assessment is as easy as counting the clients that connect to the central server.

In the case of P2P based networks the solution is not as easy, since the bot do not connect to any single point to be accounted for. In these cases, one solution is to crawl the whole overlay network address space to have an estimate on the number of nodes connected [39].

Another solution applicable to botnets that use an independent server protected by domain name randomization is to register a future domain and count the petitions received on the server and estimate the total number taking in account the domain name generation algorithm [62].

In DHT based solutions, it is possible to have a statistical estimate of the size of the network based on the analysis of the local routing table, as it will be described in the case study chapter.

3.5. Resilience

3.5.1. Traditional approaches

Traditional approaches to improve botnet resilience are low-technology solutions, like simple redundancy or hosting C&C servers in countries with relaxed computer-related laws. These solutions either reuse common practices or exploit legal reasons rather than structural strengths.

Centralized botnets used IRC servers as C&C channels to take advantage of the resilience built in the IRC protocol, more resistant to network connection problems than other protocols [60].

3.5.2. Fast Flux Service Networks

Since November 2006 it has been observed that botnets have started using a technique evolved from round-robin DNS cycling, called Fast Flux DNS [38]. It has two variants, simple and double fast flux. All this techniques are used to avoid the failure of connectivity in a single system affect the global reachability of the information shielded by the Fast Flux system. The most refined of these techniques virtually render useless blocking any IP. They relay on the DNS service to shuffle the IPs used, in order to make it impractical to track down the changes and react blocking them in a sensible time.

Round Robin DNS

To understand Fast Flux, it's important to know how Round Robin DNS works. Round Robin DNS is a technique developed as an attempt to balance load among different hosts offering a single service. As its name suggests, it is based on the Domain Name Service, and therefore it is independent of the service offered by the hosts, providing a generic solution for many different scenarios.

The basic idea behind Round Robin DNS is storing a list of several host addresses for a single domain name instead of using just one address. After each DNS query by a client, the list of servers is permuted. The most common permutation is moving the first result to the end of the list. This way, an effectively different IP address is given to clients each time they request a name resolution for the domain. With Round Robin DNS, the load to the final servers is distributed equally among all the hosts in the list [12].

However, the Round Robin DNS technique is not perfect and it has numerous shortcomings [86].

First, the list of domains is quite static since it is the domain name server that manages the list. There is no possibility to dynamically change the server pool as needed, as is the case with the very unstable hosts that make up a botnet.

Second, the list is relatively short as it contains at most a few dozen IP addresses. It would be impossible to contain the hundreds of thousands or millions of addresses of hosts that are in a botnet.

Another important problem is that Round Robin DNS only allows load distribution and not load balancing. The addresses are changed sequentially and not depending on the real time load level of the servers. DNS caching, different server resources and requests with different workload cause some servers to have a higher load than others. Since the list the Round Robin manages is static, new requests will be distributed among all servers, giving similar amounts of work to idle and loaded servers.

Simple Fast Flux

Simple fast flux is basically a Round Robin-DNS combined with a dynamic updates and a very short Time To Live (TTL). The TTL value is a property of the DNS record which describes how long the record will be valid [73]. This short TTL allows a very fast change of destination IPs, usually under 3 minutes [38].

Each node of the network registers itself as a valid IP for the domain for a certain amount of time. The DNS server includes the IP address for the short TTL preselected. Then at any given time, when a client request a domain name resolution, the pool of possible IPs is resolved by the DNS server in a Round Robin fashion. The whole process of delivering a request in a FF network is illustrated in Figure 3.5.

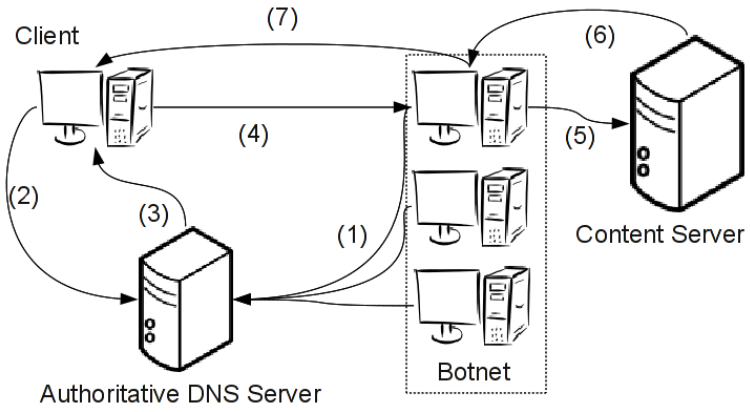


Figure 3.5.: (1) Bots register to the DNS server. (2) Client makes a DNS request about the domain. (3) DNS Server answers with the address of some bot. (4) The client makes the request to a bot. (5) The bot retransmits the request to the actual server. (6) The server sends the content to the bot. (7) The bot retransmits the content back to the client.

With this technique the IP for a domain is always different so it makes no sense to try to take down the destination servers, as they just will be replaced with other nodes registering themselves to the DNS server. This provides the system with a effective resilience property.

Also, a disconnection of any node that was registered to the DNS pool doesn't affect heavily the service, since due to the short TTL assigned to it, it would soon be replaced by some other node. In an environment with a lot of unstable nodes this is very important, since the system is self-healing by design.

Another advantage of having a dynamic update of the IP addresses is that nodes that are under heavy load can be excluded from the pool. If a node's address has been cached by some other DNS server and is getting requests from many clients or it is under attack, it is enough if the node doesn't register itself anymore with the Fast Flux DNS server, it will be deleted from the server as soon as the short TTL expires and another node will take its place [64].

Double Fast Flux

With the simple Fast Flux there is still a single point of failure, namely the DNS server. If this server is neutralized, the bots lose their ability to announce themselves and they become inaccessible, rendering them useless for certain tasks. The system is still vulnerable to having one critical IP that can block the whole network.

This vulnerability is prevented by using the same technique as Fast Flux but one level higher. Some nodes with DNS server capabilities register themselves as a NS record for the domain. Having DNS server capability means having a public IP, high bandwidth and processing power and not having the UDP and TCP port 53 firewalled. The proxy nodes, register to the main server controlled by the botmaster. The registration is illustrated in Figure 3.6. The actual DNS data usually come from the same server as the main content [64].

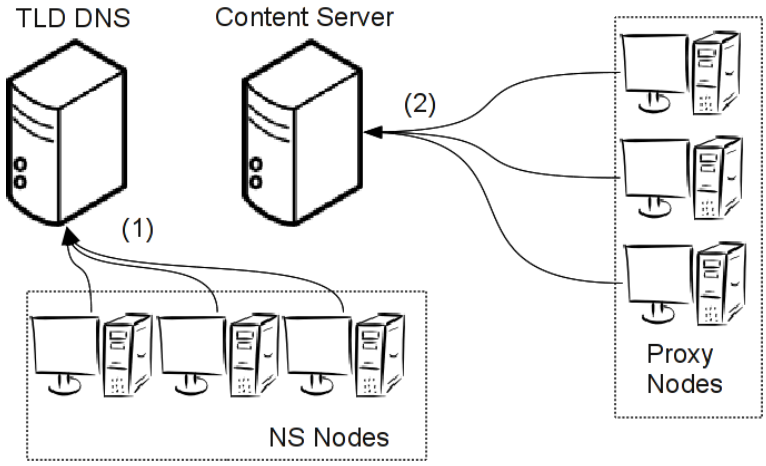


Figure 3.6.: (1) Bots with NS capabilities register as NS nodes. (2) The proxy bots register as valid A records with the main content server.

After registering as NS record for the domain, these nodes become the DNS servers for the zone. If one of these DNS nodes goes offline for any reason, another one will take its place seamlessly as the TTL is very low. The resilience properties of Fast Flux DNS are applied for the DNS server layer.

When a client tries to access the Double Fast Flux enabled domain, first it queries the TLD server for the NS record for the domain zone. Usually this step is done by the client's DNS server, since workstations rarely perform the recursive name resolution. The root server returns one of the NS bots as the authoritative DNS server for the domain. When the client or his recursive DNS server queries the NS bot for the A record for the requested domain name, the bot relays the query to the main content server, to which all proxy bots have registered previously. The main server answers the query with some proxy bot's IP address and the NS bot relays the answer to the client. The process can be seen in Figure 3.7.

The rest of the transaction continues as in the case of a single Fast Flux.

Using this technique is very effective and the only way to disrupt a network that uses Double Fast Flux is to go even higher in the DNS and neutralize the whole domain at the DNS registrar. There is no particular IP address that can be disconnected to disrupt the network. The origin of the queries is a Top Level Domain server, which is a public server and cannot be disconnected [73, 53].

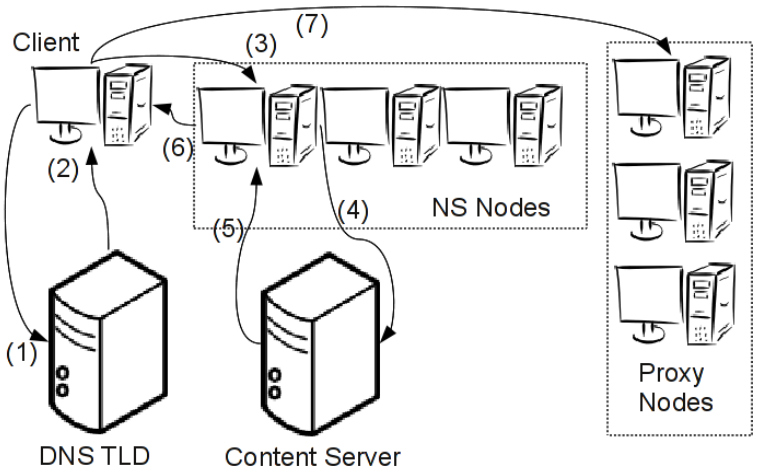


Figure 3.7.: (1) The client (or its DNS server) requests the NS record for the domain. (2) The Top Level Domain server returns one of the registered NS nodes. (3) The client (or its DNS server) requests the A record for the domain. (4) The NS bot relays the request to the content server. (5) The content server returns the address of a proxy bot to the NS node. (6) The NS bot returns the reply to the client. (7) The client contacts the proxy bot as in the case of Simple Fast Flux.

3.5.3. Domain name randomization

While fast flux is very effective against host failure or host blocking it still has a single point of failure. Fast Flux and Double Fast Flux have in common that they use only one domain name. This weakness is attacked by law enforcement asking domain registrars to block those domain names that are being used to host illegal content or for illegal activities.

Some of these DNS registrars take the domain names down only after a certain time. This can be due to high workload of the responsible people or deliberately to buy some time for their clients, from whom they make profit. Even considering these obstacles, the domain names are taken down at the end. If the botmasters don't take any action before the domain name stops working, the channel is blocked.

To prevent this, botmasters have started using domain name randomization. Bots that use this technique include, among others, Conficker [25], Torpig [78] or Srizbi [84]. This technique is a natural evolution of the trend. Fast Flux changed constantly the IP of the end points, while having a static DNS server and domain name. Double Fast Flux improved it by also moving the DNS server around, but using a static domain name. The next step is to make the last static element, the domain name, also dynamic.

Domain name randomization consists of making the bots generate different domain names depending on the time. Each time slice, which usually is a day, the target domain name varies, thus making the blocking much harder.

Since the former techniques relayed on static elements to coordinate, using dynamic domain names, the bots and the botmaster must agree somehow to use the same domain names. The most usual way to coordinate this is to use a pseudo-random generator based

on time. Based on the date, both botmaster and all the bots from the botnet generate a new but equal domain name. The botmaster registers the name with a DNS registrar and the bots start updating the records using Double Fast Flux or contacting the domain to receive orders, depending on the purpose of the domain name in the particular bot design.

Unique generation

Some bots use domain name generation to come up with one random domain name or at most a few of them [62]. The domain name generated by the bot depends only on the time. This technique isn't very robust because it allows security researchers predict future domain names and block them in advance.

The prediction can be obtained either reverse engineering the generation algorithm and feeding it with future time data or simply changing the time on a infected host and analyzing the bot's behavior.

Multiple generation

To avoid early domain blocking, or at least make it more difficult, botmasters have tired to use generation of multiple names. Every time slice, they generate a very big pool of possible domain names. They exploit the fact that it is very difficult and impractical to block all of them.

Botmasters then register only a few names from the pool to deploy their payload. Individual bots can try to obtain the payload contacting all of them in sequence until they succeed or only a small subset, to avoid signature based detection [45, 63].

3.6. Anonymization

Botmasters have to control their botnets but they know that if they are caught doing so they will be identified could face legal consequences, so they have to remain anonymous.

They use several methods combined to control the botnet without revealing their identity, and usually they are only traced if the make a mistake, since these methods are very reliable to perform their task.

3.6.1. Onion routing

The principle of onion routing is very simple. There must be a network of computing systems in place, acting as routers, to carry the messages from a sender to a recipient. These routers can be any computer system with network connectivity that wants to be part of the network, including personal computers. Usually people donate their machines to an onion network to benefit themselves from the anonymity it provides. A very popular system that implements onion routing is TOR [22].

When a sender wants to send a message to somebody, it packs the message in a special container and sends it to some random peer. This peer also packs the message in a container and sends it to another peer as well. After repeating the process several times, one peer finally delivers the message to the intended destination. Since the message bounced from several undetermined peers, it is almost impossible to know the real origin of the

message. The only method of getting the original sender would be to trace back each hop of the route, a very difficult task since the nodes usually won't collaborate, or won't even keep a track of the packets they route in the network, due to the high traffic load [31].

3.6.2. Message authentication

The direct and desired consequence of the anonymization is that the source of any anonymized message, command or update is unknown. If no measure was taken, this would lead to chaos inside the botnet since any communications must be accepted to keep the interconnection up. To avoid anyone taking over the botnet, the bots must have a method to discerning which incoming data has been issued by the botmaster and disregard all the rest.

This goal can be achieved used well known asymmetric key cryptography technology proposed in [74] and done for instance by the Sinit bot [75] or Phatbot, which is based on a P2P network that uses public key, WASTE [76]. In asymmetric key cryptography there are actually two keys to make the system work. Application of one key to any piece of information can only be reversed by applying the other key. One of the keys is made public and the other one is kept secret. This is why it is also known as public-key cryptography. The public key is used to encrypt data, since it can only be unencrypted by the private key, only in possession by the intended recipient of the information. The private key is used to digitally sign information. Since the issuer of the information is the only one who posses the private key, once he applies it to some information anyone can verify that it has been issued by him, checking that the public key correctly recovers the information [69].

Since the processes involved in asymmetric key cryptography are very CPU-costly, usually the implementation uses an intermediate information which is smaller than the whole message but provides the same functionalities. In case of encryption, the message is encrypted with a symmetric key. The symmetric key is encrypted with the asymmetric public key and attached to the symmetrically encrypted message. In case of signature, a hash of the message is calculated and the it is signed with the asymmetric private key.

In an anonymized botnet, all data that the botmaster wants to transmit to the bots is digitally signed, and the public keys corresponding to the private keys used for signing must be distributed with the bot, even within code.

3.7. Conclusion

This chapter contains the detailed description of the most important techniques used by bots and botnets, their origins, their applications and examples of their uses in other scenarios. These techniques will be extended to other potential applications in chapter four and some of them will be used in the case study in chapter five.

4. Future evolution

4.1. Introduction

This chapter presents the current trends in botnet design and outlines a possible future evolution of the botnets and bot behaviors.

Bots and bot masters are not done yet coming up with new ideas. They are going to change and improve, not only to avoid analysis by researchers but also to compete within the black market with other bots.

It is very difficult to make predictions in technology related topics as it is a very quickly changing environment. And of course, it is specially difficult in a obscure and secretive illegal market.

Nevertheless, there are some visible tendencies in recent bots towards certain behaviors and technologies. Also, it is possible to guess the bot masters actions based on previous reactions to Whitehats' activities and analysis of their botnets, since they are aware of those activities and are trying to avoid them in the next generation of bots.

4.2. Diversification of infection vectors

This technique nothing but a combination of the infection vectors present in many examples of current malware. Future malware will probably propagate like email links to download multi-program exploits, on removable drives or on social networks, depending on the method available at any particular time. Whenever a new infection vector is discovered, it will be just added to the pool instead of being used alone.

This way the infection is much more efficient, since it can use different vectors to avoid different protections. Many computers are protected against these vectors individually, but a mix of them can infect those computers. For example, using social networks to avoid a company perimeter firewall and remote service exploitation from inside to spread among the company workstations would be effective. Social networking alone would only infect the hosts whose users click on the malware links. Remote exploitation alone would be stopped by the external firewall. The two vectors combined manage to infect all hosts.

4.3. No clear central C&C

Since the C&C servers are a very obvious point of failure, this will probably be improved. Botmasters have already distributed the end points with P2P and Fast Flux technologies so it is just a matter of time to see that happening with the C&C servers. Usually P2P networks are used as a protective layers to distribute the addresses of C&C servers or to distribute updates, but the C&C servers are still unique and prone to failure.

4.3.1. Random nodes as C&C

One possible option is to have small groups of bots organizing with an ad-hoc randomly chosen node as a local C&C node. In case this node is taken down or just becomes unavailable for any reason, the rest of the bots will choose another node as the C&C, having a flexible group based hierarchical structure. This can be expanded as a hierarchical structure forming a dynamic tree structure for the whole botnet. [87]

4.3.2. No C&C nodes at all

The C&C nodes are not important by themselves, but for their function. What is important is to make the commands get to the bots, so if there were a way to make that happen without a unique server, there would be no need to have C&C nodes at all.

As it has been described before, P2P technology and DHT are very good and efficient methods to distribute information. It has been used to distribute updates and C&C addresses. There is no technical reason not use these known methods to distribute directly the commands instead of pointers to commands. If the control can be done in a P2P fashion there would be no need to C&C servers, and therefore no single point of failure, making the network much more resilient and probably able to stay alive as long as each individual node remains infected.

This way the C&C servers will be replaced by just C&C channels inside the network. Due to the nature of the P2P environment, it would be possible to anyone to access these channels, so in order to prevent a hostile take over, the commands must be authenticated, the easiest way being public key signatures.

GNUnet is also a very good candidate as it already integrates anonymization in the core of the protocol, with the accompanying plausible deniability. The botmaster would be able to inject new commands into the botnet and it would be looking to any external observer as just a retransmission of some incoming packet from the in-botnet traffic. It would be sufficient for the botmaster to own an infected node and of course the private keys in order to have complete control over the botnet.

4.3.3. Hybrid

As usual, all methods have their shortcomings, and this case is not different.

Local hierarchical group based botnets expose partial vulnerabilities to disruptions in case the local C&C node is taken control of. It doesn't eliminate the problem, just reduces it.

Pure P2P based botnets lack the responsiveness and accountability (let's remember that botnets are a business) of a more centralized solution. Also as usual, it would be the desired solution to have the best of both worlds and that is possible with a hybrid solution. The botnet would usually behave as a pure P2P network with no nodes more important than others, but will adopt a hierarchical structure for short periods of time based on the wishes of the botmaster, with some nodes acting as superpeers. This way, it is easy to control the bots in a coordinated fashion for tasks that response time has a critical importance (for instance, DDoS attacks) and to function in a distributed way for tasks that don't

require it, thus improving resilience overall. A full description of a Hybrid solution can be found in [82].

4.4. Size obfuscation

When using a P2P based architecture, the need of dedicated C&C servers will probably disappear completely. This doesn't mean that the Fast Flux techniques will become outdated. To be part of a P2P network, each bot will need to find some peers. It can be done providing a list of known bootstrapping nodes in the code. However, modern bots usually delay their activation several hours, in which the nodes designated for bootstrapping can go offline or just be blocked.

One solution for this is to perform a random network scan of Internet hosts hoping to find some other bot and get a peer list from him. However, random scans are not a very stealthy method and can probably be easily identified by IDS solutions. This is where the random domain name generation and fast flux can play an important role. Having the bootstrap nodes constantly changing in DNS and IP in a coherent way, discoverable by any bot, makes it very hard to disrupt.

Another advantage would be that these nodes would only be contacted on activation or after a long period of inactivity by the bot, where all the previous peers would be not reachable. This way, registering a domain before the botmaster and monitoring the traffic wouldn't give researchers an estimate on the size of the botnet, but only an estimate of the growth of the botnet in a particular slice of time. This is because each connection received wouldn't be an active node but a new, bootstrapping bot. Given that the botnet won't grow too much to avoid attracting too much attention, after a small period of time very few new nodes will be activated, only to maintain the bot count compensating uninstalled bots from former nodes. It is still a interesting measurement, but much less informative than the total bot count.

4.5. Encrypted traffic

One of the most important tasks for a security researcher is to know what a bot is doing and how it is doing it. A very useful method to discover this is by snooping on a bot's network traffic streams. An obvious evolution already seen in botnets [46] is to make this analysis as hard as possible by encrypting all the traffic for the bot control.

As the bot itself has to understand the traffic, it will have the key in one way or another, so in the end it will be possible for anyone determined enough to get the information, but it will be much more complicated at very little cost for the botmaster. There are readily available Crypto libraries so encryption represents a very little effort. Also it thwarts most attempts to fingerprint the traffic by any network intrusion detection system, as properly encrypted data can only be read by the endpoints of a connection.

4.6. Signed content

One technique security researchers use is to flood the botnet P2P system will bogus information [61, 39, 11]. With this attack, nodes become saturated with data and the original botmaster's commands or data become inaccessible.

A remediation to this attack can be allowing to store only information that has been digitally signed by the botmaster. If there is some information that doesn't validate against the public key that any bot has hardcoded, the information is not accepted and the DHT records are not set.

With this solution, nobody will be able to publish content in the botnet C&C channel, and therefore the commands issued by the owner of the private key will spread without any problem to all nodes participating in the P2P network.

Other data that is generated by the bots themselves cannot be digitally signed since the private key cannot be embedded in the bots, to avoid reverse engineering of the key and a takeover of the botnet. Data that the bots generate and share with other bots can be placed in a different DHT, parallel to the C&C DHT. Since individual bots can be taken over, there is no easy way to protect the bot-generated data to be polluted with garbage, since it would require a complex trust-based rating system, which will not be contemplated in this work.

Another solution is to have bots "push" trusted and signed information to each other in a P2P fashion, independently from the DHT. This way there would be no need to have a separate DHT and each node could locally check the information signature before taking any action. After checking that the data is legitimate and taking any required actions, the bot would inform its peers and send them the information. In case of an invalid information being injected in the network, the first bot would check the signature and discard the data since the validation would fail.

4.7. Disguised traffic

Encrypted P2P is not uncommon and will probably become more popular as file sharing applications are more widely used. However, in some environments it is uncommon and can rise suspicions, like in a company's internal network, when the company doesn't use any similar software. In these cases the traffic must be disguised as some other class of traffic. The ideal candidate is some traffic that is encrypted on the wire, so nobody can check its contents and relatively high bandwidth so the sudden increase is not noticed. HTTPS traffic is the ideal candidate for it. It is almost never blocked in any network, and even when all traffic is blocked, HTTPS may be proxied to the outside. Even when proxied, it is usually not unencrypted since it would be an employee privacy violation, which many legislations forbid.

4.8. Dynamic domain name generation

Bots like Conficker used an algorithm to generate a pool of possible domain names and randomly select one to contact the C&C channels. This made impractical for law enforcement to register or block these domain names peemptively. But it is possible to go one step further and make it impossible.

Conficker already used online services to get the precise time to generate the names [62]. But time is something predictable, there is no problem in generating future names just using a future date as input.

It is possible and desirable to botmasters goals to use other information dependent on the time but impossible to predict. Something simple to parse and stable enough across the world to be reliable to seed a pseudo random number generator and obtain the same results for all or at least many bots and the botmaster. The domain names would be generated with that information and therefore it will be impossible to know what domain names will be used the next time slice. Meteorological reports of some major city or information about some stock values would be good candidates for this kind of information. These are even available in public fixed formats, like airport METAR strings about weather conditions, which are not likely to change in format or be hidden from the public.

4.9. Using third party services

There have already been some botmasters that used social networking sites as infection vectors. Sometimes it consists of parsing twitter's trending topics and automatically publishing a tweet with those and some social engineering message to make people click a link and execute some exploit code [20]. Other bots use it as a C&C channel [58].

Another variant is using social network sites as a modern version of email worms. Each time the malware infects a system, instead of sending emails to the whole contacts list, it publishes a story on a social networking site so the contacts will read it and run the exploit code [5].

Third party services can also be used to provide anonymity. In order to have complete anonymity even from other bots in the network, nodes must have no direct communication among them. The only way to share information to stay connected is using third parties for it. Some bots will publish information and others will subscribe to it retrieving it from the third party. Those third parties can be any Internet service with enough legitimate users to hide the botnet coordination traffic. Services like twitter, google, facebook, pastebin and wikipedia can be very good examples of these high traffic sites.

4.10. Friend to friend

Another possible architectural improvement to thwart investigation by other people is to have a Web of Trust (WoT) inside the botnet.

Friend to friend (F2F) is a special topology where the P2P approach is combined with the concept of Web of Trust. In a F2F network, each node only connects directly to other nodes that belong to trusted peers.

This connection pattern has two main advantages: it allows out-of-band key exchange and preserves the node anonymity. In case of needing a resource hosted in any peer not in the friend list, data exchange might be achieved by connecting to it via a path through common friends or friends-of-friends, thus avoiding any direct connection and revealing no information to each other. This approach has also the possibility of identity disclosure, if a peer starts misbehaving, for instance sending invalid information or has been compromised.

In this case, a possible metric to evaluate the trust level of a peer would be the amount of spam or DoS traffic sent by the bot. The bots then would organize in a F2F kind of network, so any bot would only have connections to peers of similar value. This would make analyzing a bit more difficult, reducing the connections maintained by each peer and having the most valuable nodes hidden in the core of the botnet. The total size of the botnet would also be concealed as each node would have only limited knowledge of the bots and would maintain connections only with similar bots.

Finally, the performance could also benefit from this scheme, since high bandwidth bots wouldn't have to communicate, and wait for, slower bots, thus not wasting time and improving the efficiency.

4.11. Improved Fast Flux structures

As seen in chapter 3, both single Fast Flux and Double Fast Flux use nodes of the network as a frontend to the content. They act as a reverse proxy to protect the main server where the content really resides. This way the main server is protected against revealing its IP address trying to access the content, since all communication will be done through a proxy bot.

One vulnerability of this system is that executing a proxy bot under controlled conditions and then monitoring and analyzing the network connections the bot establishes, can in fact reveal the real address of the content server. This gives security researchers a target to take down, a single point of failure. If the main content server is taken down, the content bots are serving will no longer be available. Even the DNS service will be disrupted, but at that point the DNS service alone will be useless.

To avoid this single point of failure, the content the botnet is used to serve, must be distributed. Fortunately, the infrastructure for this already exists, as the bots are already involved in the DNS resolution and content serving. The same bots are also part of a big P2P network, with a very high bandwidth, which is an ideal environment to data distribution in an efficient way.

To make the system work without the backend main servers, the content must be distributed to the former proxy nodes using the P2P network. These former proxy servers will work as content servers. If the service hosted by the system needs to serve dynamic content from a backend database, the DHT can serve as the distributed data storage. If some information is of special importance and must remain private, it can be stored encrypted with the public key of the botmaster embedded in all the bots, so only the owner of the system or an authorized part can access the data.

The NS nodes that relayed DNS queries to the main backed server, will have to handle the resolutions themselves. In order to do the name resolutions they need the IP addresses of the final content servers. The NS nodes will still register as authoritative name servers for the domain and wait for registration of A host addresses.

The content server will query the top level domain server for the NS record for the domain. The TLD server will answer with one of the NS nodes that will be registered and the content node will register as a valid A record with the NS node, as seen in Figure 4.1. It is also possible to request the NS register several times and register with more than one NS

bot, depending on the amount of the A bots and NS bots.

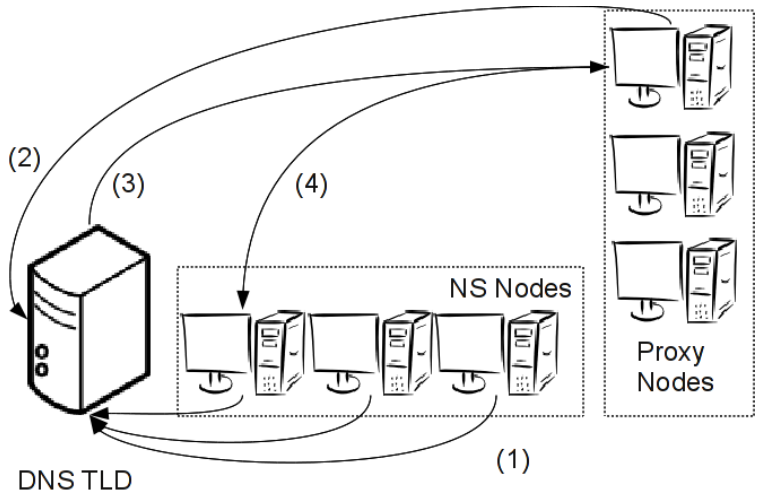


Figure 4.1.: (1) NS Bots register to the DNS TLD server. (2) A ready content node requests the NS register for the domain. (3) DNS Server answers with the address any of the NS bots. (4) The content server registers as a valid A record with the NS bot.

With this schema, the client (or security researcher) perspective of the system won't change. What will change is that the nodes are no longer proxies to a server but servers on their own. If a researcher intercepts and analyzes a bot, there will be no backend connection, so there will be no information leak about the system and no special importance servers to take down.

4.12. Smartphones

Smartphones and PDAs are a common thing nowadays, and they are becoming more and more connected to the Internet. The bandwidth they have is not as big as with computers yet, but they compensate it by usually storing more personal information, as the victims carry them always everywhere. This makes smartphones prone to be the storage of passwords, account information and other sensitive data. It would not be unexpected to see malware attacking those devices, not for sending spam or DoS attacks, but for data theft.

4.13. Conclusion

In this chapter, the trends observed in botnets have been discussed and their possible evolutions have been proposed. Also, some of the techniques described in the previous chapter have been analyzed to look for weak points and to remediate them.

Part III.

Case Study

5. Application

5.1. Introduction

This section describes the proof of concept application developed using the knowledge acquired from the analysis of botnets and presents the behavior of the application under various hostile scenarios. The goal is to apply techniques used in botnets to a improve the resilience of a whitehat scenario. After the introduction, the design of the application is explained, with special attention to the techniques adapted from botnets. Then the solution will be analyzed and possible variations will be discussed.

5.1.1. Description

The application developed is a pure P2P application, implementing a variation of the Kademlia algorithm, which is an algorithm used by many botnets as well as whitehat applications. The variations allow the network to be resilient to the attack that many botnets and distributed applications are vulnerable to, the Sybil attack. This attack consist of inserting malicious peers in a P2P network in order to make some information unavailable, as described in the chapter three. The technique used to achieve this resilience is adapted from botnets like Conficker, and is based on randomizing the ID meta-information used to locate the proper data inside the network.

5.1.2. Related work

The Sybil attack is a very well known problem in distributed systems, like DHT [24, 61] or sensor networks [59]. Many solutions to remediate this attack have been proposed [47]. Most of them are based either on certification by a central authority, which is absolutely not desirable in a network aiming to be resilient, or node resources testing. A solution for DHT routing has been proposed [21], but it applies to Chord, which uses a different routing algorithm than Kademlia [77].

The proposed solution, based on botnet-like behavior, is a novel approach to solve this problem at least in the realm of the Kademlia DHT. Its application to other scenarios where the Sybil attack is relevant will not be analyzed but it might be possible for some of them.

5.1.3. Uses

The application is aimed as a base for developing other programs on top of it, offering primitive operations on the DHT, in a similar concept as the original Kademlia KadC implementation or GNUnet.

The properties of the application allow a number of uses for it, since it is resilient to both external and internal attacks.

One possible use would be to build a takedown resistant file storage on top of it. Since the network is resilient to attacks and operates without the need of a central server, there is no clear point of attack against it. Any organization trying to disrupt the functioning of the network would have to disconnect each node or have the capacity to pollute the network with enough false nodes to make the data inaccessible. With the proposed design, it would mean to perform a Sybil attack on $1 + 2^{16}$ different nodes per blocked piece of data. In case this attack would be possible, it is trivial to just increment the pool of possible targets adding bits of randomness to the locator information.

Another possible use is on highly unstable networks, such as networks composed of mobile nodes. Due to the extra replication introduced and the inherent replication with each search already present in the Kademlia specification, the data has more chances to survive the high node churn rate present in these kind of networks. The node churn (continuous leaving of present nodes and joining of new ones) can have tangible consequences identical to a Sybil attack. The nodes close to the which had the information may leave the network but their contact data is still stored in other nodes' routing tables. Any search for the data stored at the departed nodes would end in a request which would fail and the data would be unavailable despite being reported as present.

5.2. Design

The application implements a decentralized structured P2P network. This P2P network uses a Distributed Hash Table (DHT). The DHT design is based on Kademlia, with some modifications taken from known botnet techniques used improve the resilience the malicious networks. The improvements are aimed at improving the resilience of the Kademlia network and eliminating any single points of failure, which are problems targeted and solved in botnets. The final result is an application that has the strengths of Kademlia without some weaknesses and allows to develop any whitehat functionality payload on top of it, without worrying about the underlying processes.

The choice of Kademlia as a basic system is motivated by its stability and reliability. It is a decentralized P2P network, which is a basic premise for the application, since a centralized network would offer an obvious attack target. A decentralized network like Kademlia provides a great resilience for the network as a whole against brute-force DoS attacks. In case of a DoS attack, only the nodes affected by the attack are unavailable. The network as a whole, on the other hand, keeps working and after a timeout and patches the hole left by the attacked nodes, resuming its operation normally.

Another reason is that Kademlia, being a DHT protocol, offers a structured organization for the nodes forming the network, which allows efficient access to the stored information. The usage of Kademlia in many P2P file exchange programs and many botnets demonstrate that it is a well tested and well behaving protocol for big networks with up to millions of nodes. This is a very important reason, since its hard to test the real behavior of a protocol with such a high load.

The main problem of Kademlia is its vulnerability to a DoS attack from inside the network called the Sybil attack [24]. Even very famous botnets like the Storm Worm were affected by this vulnerability [39, 11]. The Sybil attack has two main forms: affecting routing or data storage [59].

Since Kademlia uses an iterative routing protocol in which there are parallel queries and the next nodes are selected by the initiator on a distance criteria, the routing attack is not very effective against it. The initiator will always choose the nodes closer to the target and won't accept random redirections from other nodes.

The attack against data however is very successful. The attacker joins the network as multiple nodes with IDs close to the key of the data that is targeted. This way, the former closer nodes will add the new fake nodes to their routing tables, since Kademlia focuses on local knowledge. When the data is stored, the legitimate closest nodes will redirect the storing peer to the attackers. The data will be transferred to them, imitating a normal node's behavior. When later any node in the network tries to access the data, it gets finally redirected to the fake nodes that will claim to have the data, but will not return any valid result.

The proposed solution is a technique the Conficker botnet used in the variation C [25]. It consisted of spreading the attack surface so it becomes impractical to target. In the case of Conficker, the attack surface spreading was to generate each day a pool of 50.000 possible domain names to communicate with the botmaster. In order to receive an update, the malware tried to contact once per day 500 of these domain names. This meant that the malware had only a 1% chance of contacting any individual server that would be registered from the pool, growing almost linearly with the number of servers. This technique reduced the basic reliability of the channel but at the same time prevented anti-Conficker groups from blocking the domain names which would result in a total collapse of the channel.

The design of this work is based on the same concept. Once a pair {key, data} is generated and the store function is called, the data is stored in the original key location but also in a number of other locations, pseudo-randomly generated from the original key. This way, when the data is to be retrieved, it can be found at the expected location but also in other locations throughout the DHT space in case the original key is being blocked by a Sybil attack or unavailable for any other reason.

Other aspects can also be potentially affected by the Sybil attack, like the size assessment of the botnet or the authentication mechanisms. These aspects are analyzed in the next sections.

5.2.1. Store

The STORE message is an iterative evolution of the original Kademlia STORE RPC. The proposed algorithm (Figure 5.1) is the basic Kademlia STORE RPC but iterated a configurable number of times. Each iteration after the first uses a pseudo-random key derived from appending two random bytes to the original key k_0 and taking the SHA1 hash from the result to have k_i, a 160 bit value back, totally different from k_0. This new value k_i is used to perform a new FIND NODE search, find the closest nodes to k_i and store the data in them using k_i as if it was the original key of the data, therefore storing it at the k nodes closest to the new key.

The number of iterations should be balanced to assure a good chance of finding the data when the original hash is being under a Sybil attack. The chances of success depend on both the number of places the data is located and the number of places the data is looked for. Since the storing of data is a very bandwidth demanding operation, it should be limited to a factor much smaller than the search iterations. The values for these parameters

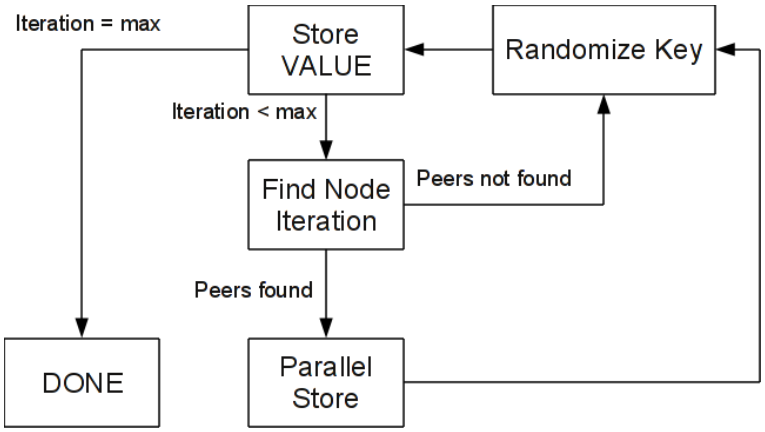


Figure 5.1.: The STORE process is iterated randomizing the key used to store the data.

will be discussed in detail in a further section.

The STORE primitive is the equivalent to the registration of domain names in the case of the botnet. The number of iterations defines the number of places the data will be available in the DHT.

5.2.2. Find Value

The FIND_VALUE message is also modified to make it iterative over a set of pseudo-random keys related to the original key. The process, illustrated in Figure 5.2, generates a pseudo-random key related to the original hash, in the case that the data is not found at the original location or the returned data is not correct. The key generation is the same as in the STORE case, so the data can be found by the FIND_VALUE operation.

First, a regular Kademlia FIND_VALUE search is made for the original key k_0. All nodes that answer with a positive result are stored and then each one of them is contacted if the data has not been correctly retrieved yet.

If this search is unsuccessful, i.e. if no node answers with a positive result or no node is able to send the correct data before a timeout, a new key is generated with the exact same algorithm as in the STORE case. The original data key k_0 is appended two random bytes and the SHA1 hash is calculated to obtain a 160 bits k_i key. Then a new search for data at k_i is performed.

The whole process is repeated a number of times depending on the importance of the information and the type of network. The parameters which are configured for that matter are discussed later in the chapter.

The FIND_VALUE is the exact equivalent to the domain names a bot tries to contact in the original botnet scenario. The more tries are performed, the better the chances to find the data are, but the traffic in the network also is increased.

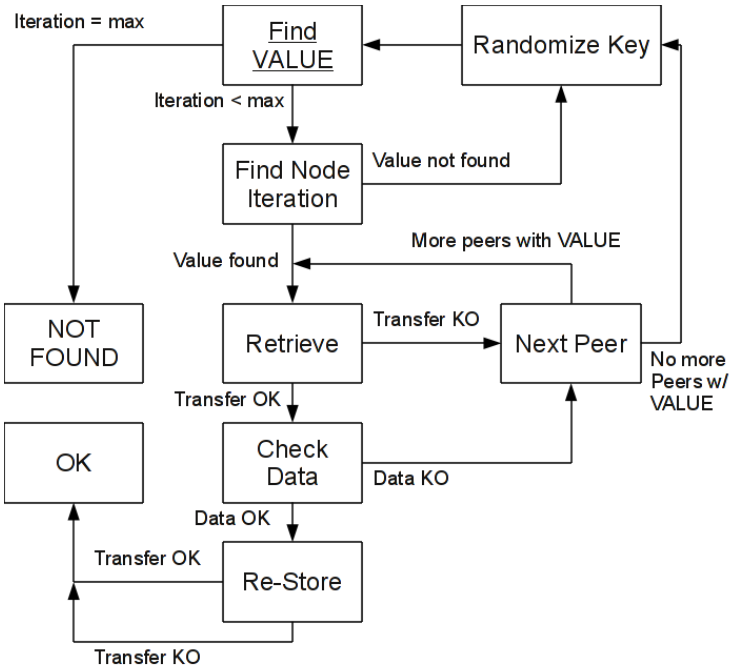


Figure 5.2.: The find process is iterated randomizing the key if data is not found or is not correct.

5.2.3. Iteration values

The number of iterations of both STORE and FIND_VALUE primitives defines the success rate in retrieving the data from an alternative location during a Sybil attack, but they come with a cost. Namely, the incremented number of message iterations causes an extra effort in the network, and an extra storage space is needed to store the replicated data.

These two parameters are equally important and mutually exchangeable. To find one unique backup stored among the proposed 16 bit space, it would be necessary to make an important number of tries to have 50% chance of finding it. The other way around, if the information was stored in 2^{15} places out of 2^{16} possible, one try would be enough to have a 50% chance of finding the information.

The main problem however, is not the number of messages on the network but the redundancy of having the same information stored at many nodes and the publication of the data, which generates much more traffic than a search query. Another reason for this is that store messages are fixed, they have to be repeated every store, but additional find messages are only issued when the key is not found on the nodes close to the original key k_0. For these reasons, it is better so set a high maximum number of find iterations while keeping a moderate number of store iterations, to avoid saturating the network. In intrinsically unstable networks, the value of the store iterations might be raised to compensate the high node churn.

The probability to make a successful attempt when there are p copies stored in the net-

work is:

$$P = p/2^{16}$$

The probability of the opposite, of a failed attempt, is of course $1 - P$. The probability of having all failed attempts in f repetitions is:

$$Q = (1 - P)^f$$

Again, the probability of the opposite is $S = 1 - Q$. Therefore, after f iterations, the probability of a successful search is described by the following equation:

$$S = 1 - (1 - \frac{p}{2^{16}})^f$$

The probability distribution as a function of p and f is illustrated at Figure 5.3.

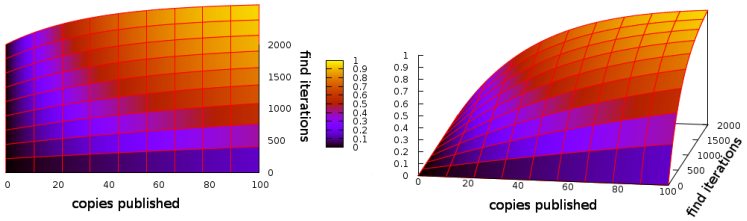


Figure 5.3.: The distribution of probability to find the information with p store iterations and a maximum of f find iterations.

Following the example of the Conficker botnet (50.000 key space, 500 find iterations, $\sim 1\%$ success per active server), but having a 2^{16} possible keys to store the information, and making maximum 2^{10} iterations per search, each storage iteration gives $\sim 1,5\%$ success. With $p = 64$ and $f = 2^{10}$, the chances of finding a Sybil-attacked key are $\sim 63,23\%$. Using $p = 64$ and $f = 2^{11}$, chances improve to a $\sim 86,47\%$. To have a high confidence in the results, parameters must be $p = 128$ and $f = 2^{13}$, which returns results in a $\sim 99,97\%$ of searches that happen under a Sybil attack, and naturally 100% of normal situations.

Since the search is performed in a random fashion, there is no way of guaranteeing a 100% precise result in a case where the original key is attacked. This could be changed by using a different key randomization algorithm for the find operation. Instead of using a random key generator, use a counter and sequentially increment it. The generation of hashes would be the same, i.e. by appending the counter to k_0 and hashing the result with SHA1. This way, performing an exhaustive search on the whole pseudo-random key space can return a definitive result. This has not been implemented but is a very reasonable improvement in the method. The practical gain however would be negligible, since the parameters mentioned make it that only $\sim 0,03\%$ of results are false negatives and with an exhaustive search real negatives would take a very long time and a lot of resources to complete.

In case of absolute need of conclusive results, a possible modification would be to reduce the key randomization space to a more manageable size like 2^{10} and perform exhaustive searches on that space. This of course would have a negative impact on the resilience of the network, since the effort to make a Sybil attack on all the key space would also be lower.

5.3. Size assessment

Due to the nature of the Kademlia protocol, each node has a good knowledge of the nodes close to it in the DHT space and a weaker knowledge of nodes further away from it. This behavior is unchanged in the implementation. Keeping in mind that the IDs are generated randomly and thus are evenly distributed, routing information can be used to make an estimate on the size of the network without an exhaustive crawl of the DHT space. The Sybil attack, as it is based on modifying routing tables of nodes next to the target key, can cause imprecise information to be analyzed and a wrong size being calculated.

First, to calculate the size based on the routing table information, it is necessary to analyze the distribution of the nodes in the network and in the routing tables of each node. The network is populated by nodes with a random 160 bit ID each. Assuming that the IDs are generated by a high quality random number generator (in this case, OpenSSL's RAND_pseudo_bytes()), the distribution of nodes in the key space is uniform. Having an uniform distribution, each node will have an ID with at least the first i bits in common with $n \times 2^{-i}$ nodes of the network, being n the total size of the network's population.

Half of the nodes will have IDs starting with 1 and the other half will have IDs starting with 0. A node with the first ID bit being a 1, will have that bit in common with $n/2$ or $n \times 2^{-1}$ nodes. The same principle applies to each consecutive bit, since the IDs are distributed uniformly.

Also, each node will have exactly i first bits in common with $n \times 2^{-(i+1)}$ nodes. From the $n \times 2^{-i}$ nodes that have at least i first bits in common, half of them will also have at least the next bit in common. So, $n \times 2^{-(i+1)}$ will have at least $i+1$ first bits in common, and the other $n \times 2^{-(i+1)}$ will have exactly i first bits in common with the node. This amount will be called n_i, and is defined as:

$$n_i \simeq n \times 2^{-(i+1)}$$

These relations are used to calculate the total size of the network based on the Kademlia routing table population.

The number of hosts with exactly i first bits in common with the local ID (n_i) is known for all i where $0 < n_i < k$, being k the parameter that defines the size of the k-buckets, and is usually 20. For values where $n_i > k$, the k-bucket implementation truncates n_i to k, so $n_i \leq k$ and thus it cannot be used to estimate the size of the network.

From the explanation above it is possible to deduce that:

$$n \simeq n_i \times 2^{i+1}, \quad \forall n_i \text{ where } 0 < n_i < k$$

Applied to the Kademlia k-bucket structure, where in the k-bucket d a node stores the hosts with a distance d to the local node (k_d), and IDs are 160 bits long:

$$n \simeq k_d \times 2^{161-d}, \quad \forall k_d \text{ where } 0 < k_d < k$$

5.3.1. Local routing table

The first approach is to estimate the size of the whole network based on the information contained only in the local routing table. This has the advantage of not polluting the network with unnecessary traffic.

First, it is important to make sure that the local node has in fact the information about the closest nodes to itself in the network. The original Kademlia design takes care of this by making each node bootstrap by performing a FIND_NODE for the local ID. This way each node either finds the closest nodes to itself at boot time or is found by other nodes when they join the network.

With this information, an estimate to n can be calculated as the weighted average of n_d, which is:

$$n_d = k_d \times 2^{161-d}, \quad \forall k_d \text{ where } 0 < k_d < k$$

The Figure 5.4 illustrates the precision of this estimates for different network sizes. To make sure the routing tables are fully populated, 16 searches are performed before measuring the number of contacts, namely for IDs 0x0000...0000 to 0xF000...0000.

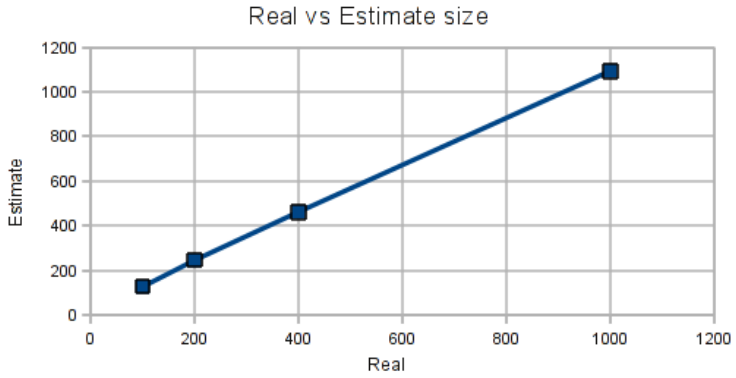


Figure 5.4.: Results from real vs estimated network size from local routing table analysis by a node not near a Sybil attack.

This value is only valid when the assumption of an uniform distribution of IDs in the network holds true. In the case of a Sybil attack, the IDs around the attacked keys are not random and thus the distribution is no longer uniform. A node near an attacked key would have unrepresentatively high values of k_d for low values of d. The size estimate in this case would be impossible and eliminating the Sybil nodes is impractical.

5.3.2. Random search

To allow making a size estimation in a network under a Sybil attack, a node can take advantage of the fact that the Sybil attack is by definition a local attack and affects only nodes with close IDs to the key being attacked. The rest of the network remains largely unaffected by the attack and thus can be used to estimate the size of real nodes in the whole network.

To calculate this estimate a search for a node with a random ID is performed. The result will be the closest node to the random ID with a distance d between them, which would correspond to having a $k_d = 1$ in the routing table. The size estimate in this case would be:

$$n \simeq 2^{161-d}$$

This rough approach can be refined by two methods, either analyzing the nodes in the Kademlia shortList [65] resulting after the search or by taking the results over several searches. Since the latter approach does not need the modification of the FIND_NODE code function, it is the solution chosen to represent the algorithm. In Figure 5.5 it is possible to appreciate the distribution of the results from 256 random searches with different network sizes. The median gives consistently a result twice as big as the real size, since there are always nodes especially close to the random ID and there cannot be results especially far due to the uniformly populated key space. Thus, the graphics are slightly shifted towards the right, and since the scale is binary exponential, the result is two times the expected size.

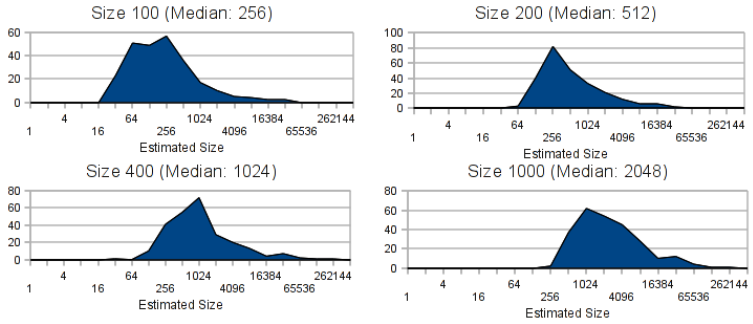


Figure 5.5.: Results from real vs estimated network size from random node searches.

5.4. Authentication

An authentication mechanism is needed in order to prevent Sybil nodes from attacking the Kademlia DHT by returning bogus data. In case of validating a node just by the fact it answers the FIND_VALUE requests with data, it would allow an attacker to feed bogus data to any node requesting the information and preventing them from searching the data at alternative locations inside the DHT, achieving the DoS purpose.

The basic authentication is done by calculating the SHA1 hash of the retrieved data and comparing it to the original key k_0, no matter where the data was stored. As long as the hash is considered secure and no collisions can be forced, this assures that the retrieved data is in fact the one being looked for. It also protects against the extremely rare case of a collision with some other pre-existing data in the DHT. Even in the border case of storing the data in all possible $2^{16} + 1$ locations, the probability of any particular piece of data to share the same hash is $\sim 2^{-70}$.

The need for strong authentication may arise from a resource depletion attack, where the attacker tries to store so much information at a node that the targeted data is discarded and replaced by garbage. Since data can be stored at places distant from its hash due to the key randomization, a node cannot refuse to store data under a key just because the data's hash does not match the key. This attack has two possible countermeasures which are not implemented by default but the primitives for their implementation are in place.

Hash checking The requesting node must provide the original random bytes that originated the secondary hash under which the data is stored. The storing node checks that the hash is in fact related to the data and considering the estimated size of the network it is reasonable to store at its ID.

Since the bigger the network, the closer the keys must be to the node IDs, a Sybil attack against a node to just store garbage would not make much sense, since the effect is the contrary as the desired. The Sybil attack would work against other nodes, making them estimate a much bigger network, and refusing to store other nodes' data. In any case, the random search size estimation gives a foolproof tool to estimate the size of the network.

The downside of this solution is that in this case the collisions do not have to be calculated against the hash size but against the network size, which is much smaller and easily doable since the biggest networks have only a couple million machines, which is not a computational problem at all.

Public key cryptography Applying the standard, usual method is a solution for the authentication problem. This requires to distribute the public modules of the authorized keys and prevents anyone from storing content in the network, since a private key is needed for this. It is also possible to require authentication only over a certain limit per node and below that limit allow anyone to store content in the node.

The OpenSSL RSA functions to create keys, distribute them, sign and check content are available as source code in easy to use wrap functions but they are not used by the network itself as it is a higher level application decision whether to make use of them and under what conditions.

5.5. Cryptography

All the messages the nodes exchange are encrypted with Advanced Encryption Standard (AES) using a 256 bit key. This is done for scrambling purposes since the algorithm does not offer total confidentiality and an attacker with perfect knowledge will be able to read the contents of the network traffic.

For each message, the sender uses a key depending on the recipient, using the recipient's ID to generate the key. Since the cypher key is 256 bits and the hash is only 160 bits, extra 96 bits are needed. Although OpenSSL allows to use shorter amounts of data to generate the key, using only the ID would generate 256 bit keys with only 160 bits of entropy. To complete the missing 96 bits, a common salt value is appended to the hash and the concatenated 256 bit value is used as a key.

When the destination node ID is not known, as it can happen during bootstrap, a 160 bit hash is calculated using the current time, with a minute precision to allow some time drift among nodes. This means that a node must be time-synced with the bootstrap it is using, in order to successfully join the network. The implementation does not include a synchronization mechanism so an external one should be used, for instance Network Network Time Protocol (NTP) [52]. This behavior has been observed in botnets like Conficker, which used a time-based functions and a thread to keep the time synced across the botnet [62].

5.5.1. Confidentiality

In order to access the contents of a message, an external observer needs to know the key used to cypher it, which as previously stated, is composed of a 160 bit hash and a 96 bit salt value.

The salt value has to be common for all nodes, so it must be commonly accessible for all of them. In the case of the proof of concept application it is distributed in the code but it could be dynamically generated based on publicly available information, as discussed for domain name generation in chapter four: meteorological information, newspaper headlines or social network data. An attacker with access to a copy of a node can analyze the functioning of the code and discover the salt value used in the network.

Knowing the salt value and the precise time, it would be possible for the attacker to decipher the bootstrap packet sent by a node, thus discovering the 160 bit ID of that node which will be used to generate the key in all packets send to that node.

Being able to decipher traffic incoming to the node, the attacker can read all data being received by the node to form a parallel contact table, storing the ID, IP address and port of all contacts the nodes communicates with or receives in FIND_NODE reply messages.

With the contact data stored, the attacker is able to read all outgoing communication generated from the node. Since the IP address and destination port are in clear-text in the UDP header of each packet, the attacker can look in the contact table for the ID of the contact at that address and, combining the remote ID and the salt value, obtain the key used for the message.

As explained above, the encryption used does not offer confidentiality against an determined attacker. The attacker however needs to keep track of all communication and know the salt value used by the bot. This is not the case in a typical automated network analysis tool, which makes fingerprinting of the traffic by automated tools much harder.

To achieve confidentiality, public key cryptography must be used. Each node, after generating a key pair at boot would include the public part in each message and include other nodes' keys in the contact table. Each message would then be encrypted with a random AES 256 bit key and the random material to generate the key would be included in the message encrypted with the public key of the destination node. Since this is a well known solution used in industry and botnets [79], it has not been included in the implementation, in order to reduce complexity and computational load. Public key cryptography is a very processor intensive task, and it was desired to obtain a lightweight enough executable to be run in thousands of instances on a single machine in order to test the solution suitability.

5.5.2. Segmentation

The way the cryptography is implemented make all the nodes in the network communicate using the same piece of information - the salt which is concatenated to the 160 bit value to form the key. A change in this salt value in one node would make the node unable to communicate with the rest of the network. The change of the value in a set of hosts to a new value shared by all the nodes in the set would however make them able to communicate inside the set.

This way, changing the salt value in a group of nodes is an effective way to segment the

network, since the nodes from the two groups will be unable to communicate with each other and after a timeout will eliminate each other from the routing tables, substituting them with other nodes from the same segment.

5.6. Conclusion

In this chapter a new solution for the Sybil attack against the Kademlia DHT has been proposed using a solution based on a behavior used by botmasters to improve the resilience of their networks. The metrics related to the new solution have been analyzed and other potential ramifications of the Sybil attack have been addressed. The application of this solution to other scenarios vulnerable to Sybil attacks has not been analyzed and it is left as a possible future work.

6. Conclusion

6.1. Further improvements

As seen in the previous chapter, locator key randomization is a valid protection technique developed first by criminals but with applications in whitehat scenarios. But this is not the only blackhat technique that can be used. Furthermore, it is not even the only technique to improve the resilience of P2P networks that can be used.

One of the few single points of failure that can still be attacked to thwart a network are the bootstrap nodes. Some software uses dynamic per-node generated listings [7], most use rendez-vous points hardcoded in the program and other solutions have been proposed to take the approach of scanning subnets on the Internet that are likely to have P2P nodes [29]. Some of these methods are either inefficient or insecure, since the bootstrap nodes can go offline before the software is first run making it impossible to join the network.

In highly volatile environments like mobile networks, even the approach of saving a list of contacts for the next run can be risky, since the next time the node tries to connect to the network, all the peers can be gone or using a different address and therefore are unreachable.

This problem has also been addressed in botnets by using Fast Flux. A P2P network could use Fast Flux techniques to maintain a real-time up to date pointer to reach the network by updating the DNS records with IP addresses of available nodes. This kind of functionality has been considered for this thesis but finally wasn't implemented and is left for a possible future work.

6.2. Conclusions

Botmasters have to create programs that live in very hostile environments. They constantly monitor whitehat applications and use proven solutions for some of their problems. They develop some other techniques on their own for other requirements. Malware and whitehat software have some common aspects and both communities may use solutions developed by the other side to improve their own products.

In such a dynamic environment as the botnet market, blackhats will certainly keep innovating and creating new approaches for known problems and new problems that might present in the future. Monitoring and adapting solutions from blackhats is a very valid resource that can save time and money by not having to develop solutions from scratch. Therefore, the security community could benefit a lot by keeping an eye on blackhats, not only as a source of problems, but also as a source of solutions.

Bibliography

[1] Magnet-uri project. http://magnet-uri.sourceforge.net/, May 2010.

[2] Twitter uses bittorrent for server deployment. http://torrentfreak.com/twitter-uses-bittorrent-for-server-deployment-100210/, 2010.

[3] Yacy, large scale search engine. http://yacy.net/Technology.html, May 2010.

[4] K Aberer and M Hauswirth. Peer-to-peer information systems: concepts and models, state-of-the-art, and future systems. In *SOFTWARE ENGINEERING NOTES*, volume 26, 2001.

[5] Byron Acohido. An invitation to crime. *Usa Today*, 1:1–2, March 2010.

[6] David S. Anderson, Chris Fleizach, Stefan Savage, and Geoffrey M. Voelker. Spamscatter: Characterizing internet scam hosting infrastructure. In *Usenix Security Symposium*, pages 135–148, 2007.

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

[8] P. Barford and V. Yegneswaran. An inside look at botnets. *Malware Detection*, 27:171–191, 2007.

[9] K Bennett and C Grothoff. gap - practical anonymous networking. In *LECTURE NOTES IN COMPUTER SCIENCE*, pages 141–160, 2003.

[10] Scott Berinato. Hacker economics. Technical report, www.cio.com, October 2007.

[11] Lasse Trolle Borup. Peer-to-peer botnets: A case study on waledac. Master's thesis, Technical University of Denmark, 2009.

[12] T. Brisco. DNS Support for Load Balancing. RFC 1794 (Informational), April 1995.

[13] T.M. Chen. Trends in viruses and worms. *The Internet Protocol Journal*, 6(3):23–33, 2003.

[14] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, page 44. ACM, 2004.

[15] Bram Cohen. *The BitTorrent Protocol Specification*, January 2008.

[16] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of the USENIX SRUTI Workshop*, pages 39–44, 2005.

[17] Luis Corrons. How the butterfly botnet was broken. http://www.zdnet.co.uk/news/security-threats/2010/03/16/how-the-butterfly-botnet-was-broken-40088328/, March 2010.

[18] Luis Corrons. Red de bots mariposa. http://pandalabs.pandasecurity.com/es/red-de-bots-mariposa/, March 2010.

[19] D. Dagon, G. Gu, and C. Lee. A taxonomy of botnet structures. *Botnet Detection*, 36:143–164, 2005.

[20] Dancho Danchev. Cybercriminals hijack twitter trending topics to serve malware. http://www.zdnet.com/blog/security/cybercriminals-hijack-twitter-trending-topics-to-serve-malware/3549, June 2009.

[21] G. Danezis, C. Lesniewski-Laas, M.F. Kaashoek, and R. Anderson. Sybil-resistant DHT routing. *Computer Security–ESORICS 2005*, pages 305–318, 2005.

[22] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, page 21. USENIX Association, 2004.

[23] D. Dittrich and S. Dietrich. P2P as botnet command and control: a deeper insight. In *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*, pages 41–48, 2008.

[24] John Douceur and Judith S. Donath. The sybil attack. In *Peer-to-Peer Systems*, pages 251–260, 2002.

[25] Niall Fitzgibbon and Mike Wood. Conficker.c, a technical analysis. Technical report, SophosLabs, Sophos Inc., April 2009.

[26] Michael J. Freedman. Experiences with coralcdn: A five-year operational view. In *7th USENIX/ACM Symposium on Networked Systems Design and Implementation*, May 2010.

[27] F.C. Freiling, T. Holz, and G. Wicherski. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. *Computer Security–ESORICS 2005*, 3679/2005:319–335, 2005.

[28] Larry Gadea. Large scale server deploys using bittorrent and the bittornado library. http://github.com/lg/murder, February 2010.

[29] C. GauthierDickey and C. Grothoff. Bootstrapping of Peer-to-Peer Networks. In *Applications and the Internet, 2008. SAINT 2008. International Symposium on*, pages 205–208, 2008.

[30] P. Brighten Godfrey and Ion Stoica. Heterogeneity and load balance in distributed hash tables. In *IN PROC. OF IEEE INFOCOM*, 2005.

[31] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[32] J.B. Grizzard, V. Sharma, C. Nunnery, BB Kang, and D. Dagon. Peer-to-peer botnets: Overview and case study. In *Proceedings of the First USENIX Workshop on Hot Topics in Understanding Botnets*, 2007.

[33] Christian Grothoff, Ioana Patrascu, Krista Bennett, Tiberiu Stef, and Tzvetan Horozov. Gnet. Technical report, GNUnet, 2002.

[34] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. Bothunter: Detecting malware infection through ids-driven dialog correlation. In *Proceedings of the 16th USENIX Security Symposium*, pages 167–182, 2007.

[35] P. Gutmann. The commercial malware industry. In *DEFCON conference*, 2007.

[36] S. Hansman and R. Hunt. A taxonomy of network and computer attacks. *Computers & Security*, 24(1):31–43, 2005.

[37] G. Hoglund and J. Butler. *Rootkits: subverting the Windows kernel*. Addison-Wesley Professional, 2005.

[38] T. Holz, C. Gorecki, K. Rieck, and F. Freiling. Measuring and detecting fast-flux service networks. In *Symposium on Network and Distributed System Security*. Citeseer, 2008.

[39] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–9. USENIX Association, 2008.

[40] ipoque. Internet study 2008/2009. http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009.

[41] Vitaly Kamluk. The botnet business. Technical report, Kaspersky Lab, May 2008.

[42] M.G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, page 53. ACM, 2007.

[43] Jimmy Kuo. Storm drain. Technical report, Microsoft TechNet, September 2007.

[44] Dennis KÃ¼gler. An analysis of gnunet and the implications for anonymous, censorship-resistant networks. Technical report, GNUnet, 2003.

[45] Felix Leder and Tillmann Werner. Know your enemy: Containing conficker. Technical report, The Honeynet Project, April 2009.

[46] R. Lemos. Bot software looks to improve peerage. http://www.securityfocus.com/news/11390, May 2006.

[47] B.N. Levine, C. Shields, and N.B. Margolin. A survey of solutions to the sybil attack. *University of Massachusetts Amherst, Amherst, MA*, 2006.

[48] Gurmeet Singh Manku. *Dipsea: A Modular Distributed Hash Table*. PhD dissertation, Stanford University, Department of Computer Science, August 2004.

[49] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[50] Robert McMillan. New russian botnet tries to kill rival. *Computerworld.com*, Feb 9:1, February 2010.

[51] C. Merchant and J. Stewart. Detecting and Containing IRC-Controlled Trojans: When Firewalls, AV, and IDS Are Not Enough. *SecurityFocus.com*, 1:1, 2002.

[52] D. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305 (Draft Standard), March 1992.

[53] P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592.

[54] David Moore, Geoffrey M Voelker, and Stefan Savage. Inferring internet denial-of-service activity. In *Proceedings of the 10th USENIX Security Symposium*. USENIX, 2001.

[55] Corey Nachreiner and Scott Pinzon. Understanding and blocking the new botnets. Technical report, WatchGuard, April 2008.

[56] Yury Namestnikov. *The economics of Botnets*, July 2009.

[57] Yury Namestnikov. Information security threats in the first quarter of 2010. Technical report, Kasperky Lab, 2010.

[58] Jose Nazario. Twitter-based botnet command channel. http://asert.arbornetworks.com/2009/08/twitter-based-botnet-command-channel/, August 2009.

[59] J. Newsome, E. Shi, D. Song, and A. Perrig. The sybil attack in sensor networks: analysis & defenses. In *Proceedings of the 3rd international symposium on Information processing in sensor networks*, pages 259–268. ACM, 2004.

[60] J. Oikarinen and D. Reed. Internet Relay Chat Protocol. RFC 1459 (Experimental), May 1993. Updated by RFCs 2810, 2811, 2812, 2813.

[61] Georg 'oxff' Wicherski, Tillmann Werner, Felix Leder, and Mark Schlösser. Stormfucker: Owning the storm botnet. In *25th Chaos Communication Congress*, December 2008.

[62] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An analysis of conficker's logic and rendezvous points. Technical report, SRI International, March 2009.

[63] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. Conficker c analysis addendum c. Technical report, SRI International, April 2009.

[64] The Honeynet Project. *Know Your Enemy: Fast-Flux Service Networks*, 2007.

[65] XLattice Project. Kademlia: A design specification. http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html, May 2010.

[66] A. Rajab et al. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, page 52. ACM, 2006.

[67] A. Rajab et al. On the impact of dynamic addressing on malware propagation. In *Proceedings of the 4th ACM workshop on Recurring malcode*, page 56. ACM, 2006.

[68] Anirudh Ramachandran and Nick Feamster. Understanding the network-level behavior of spammers. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2006.

[69] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–26, 1978.

[70] Mark Russinovich. Sony, rootkits and digital rights management gone too far. *Technet Blogs*, 1:1, October 2005.

[71] Reporters sans Frontieres. The enemies of the Internet. *URL: http://www.rsf.fr*, 1:1–64, March 2010.

[72] S. Shin, J. Jung, and H. Balakrishnan. Malware prevalence in the KaZaA file-sharing network. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, page 338. ACM, 2006.

[73] M.K. Stahl. Domain administrators guide. RFC 1032, nov 1987.

[74] S. Staniford, V. Paxson, and N. Weaver. How to own the internet in your spare time.

[75] J. Stewart. Sinit P2P trojan analysis. *Web publication. Available at URL: http://www.secureworks. com/research/threats/sinit*, 2003.

[76] J. Stewart. Phatbot trojan analysis. *Retrieved from Secure Works: http://www.secureworks.com/research/threats/phatbot*, 2004.

[77] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, page 160. ACM, 2001.

[78] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the ACM CCS*, November 2009.

[79] S Stover, D Dittrich, J Hernandez, and S Dietrich. Analysis of the storm and nugache trojans: P2p is here. In *USENIX ;login:*, volume 32, December 2007.

[80] H. Thimbleby, S. Anderson, and P. Cairns. A framework for modelling trojans and computer virus infection. *The Computer Journal*, 41(7):444, 1998.

[81] D. Turner, M. Fossi, E. Johnson, T. Mack, J. Blackbird, S. Entwisle, M.K. Low, D. McKinney, and C. Wueest. Symantec global internet security threat report–trends for july-december 07. *Symantec Enterprise Security*, 13:1–36, 2008.

[82] P. Wang, S. Sparks, and C. Zou. An advanced hybrid peer-to-peer botnet. Technical report, University of Central Florida, 2007.

[83] D.J. Weber. *A taxonomy of computer intrusions*. PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 1998.

[84] Julia Wolf. Technical details of srizbi's domain generation algorithm. Url: http://blog.fireeye.com/research/2008/11/technical-details-of-srizbis-domain-generation-algorithm.html, November 2008.

[85] Jeff Yan and Brian Randell. A systematic classification of cheating in online games. In *Proc. 4th ACM SIG-COMM Workshop on Network and System Support for Games*, 2005.

[86] W. Zhang et al. Linux virtual server for scalable network services. In *Ottawa Linux Symposium*, pages 1–10. Citeseer, 2000.

[87] Zonghua Zhang, Ruo Ando, and Youki Kadobayashi. *Information Security and Cryptology*, chapter Hardening Botnet by a Rational Botmaster. Springer Berlin / Heidelberg, 2009.